POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Development of a Decentralized Social Media

Supervisors	Company Supervisors	Candidate
Prof. Antonio J. DI SCALA	Fabrizio MAIOCCO	SeyedHossein JAVADIZAVIEH
	Ismail NASRY	

Academic Year 2022-2023

Acknowledgements

I am deeply grateful to everyone who has been by my side throughout this journey. First and foremost, I extend my heartfelt thanks to my family for their unwavering encouragement, even during the most challenging times when solutions seemed elusive. I am also indebted to Professor Di Scala for his constant availability and valuable clarifications, as well as Professor Bazzanella for instilling belief in me. Without their trust and support over the past two years, this work would not have been possible.

The BitPolito team has been an extraordinary source of inspiration and motivation throughout my journey at Politecnico! Their unwavering belief in our mission as a student team has propelled us to create one of the most remarkable and defining experiences of my time here. Their dedication and support have exceeded my wildest expectations, instilling in me a newfound sense of purpose and determination. I am deeply grateful for their trust and commitment, as it has enabled us to achieve feats I could have never imagined before. BitPolito's impact on my life and the Politecnico community is immeasurable, and for that, I am truly thankful.

I would like to express my sincere appreciation to CGM Consulting S.r.l., particularly Fabrizio Maiocco and Ismail Nasry, for providing me with the opportunity to work on the practical aspect of this thesis. Their introduction to the world of Blockchain and its pragmatic applications has been enlightening and instrumental in my academic growth.

To my friends, I am immensely thankful for making my time in Turin joyful and memorable. These years have been nothing short of wonderful, and I cherish the memories we have created together. To my entire family, your understanding and unwavering support have meant the world to me. Thank you for standing by me and supporting every decision I have made in life. And to MohammadMahdi, you have been the driving force behind my constant pursuit of self-improvement over the past two years. Your presence has revolutionized my life, and I am profoundly grateful for your encouragement.

Summary

During my enriching internship at CGM Consulting S.r.l., I delved deep into the evolving world of Web3 technology, focusing particularly on the NEAR Protocol. This experience was a part of my thesis for my Master's Degree, where I extensively explored the development of decentralized applications (dApps) in the Web3 ecosystem.

My primary responsibilities included developing and integrating various cutting-edge functionalities within dApps. One of the key areas I worked on was the integration of Non-Fungible Tokens (NFTs). This involved creating smart contracts that facilitated the minting, transferring, and managing of NFTs, using Rust programming language. These contracts were essential for handling digital assets securely and efficiently on the blockchain.

Another significant aspect of my internship was incorporating AI Image Generation into the dApp. This innovative feature allowed users to generate unique images based on specific inputs, enhancing the user experience and adding a creative dimension to the application.

Furthermore, I explored the implementation of oracle data, particularly focusing on Chainlink oracles. These oracles acted as crucial connectors between the blockchain and external data sources. My involvement was centered around integrating weather forecast data, showcasing how real-world information can be effectively used in smart contracts and dApps. This integration not only expanded the functionality of our dApps but also demonstrated the practical applications of blockchain technology in accessing and utilizing real-world data.

Overall, my internship at CGM Consulting S.r.l. was a deeply informative and transformative experience. It provided me with a comprehensive understanding of blockchain technology, smart contract development, and the practical implementation of innovative features in dApps. The skills and knowledge gained during this period significantly contributed to my thesis and have equipped me with valuable insights into the potential and application of blockchain technology in various domains.

Development of a Decentralized Social Media

List of Figures

Acronyms

AI	Artificial Intelligence
dApp	Decentralized Application
API	Application Programming Interface
NFT	Non-Fungible Token
DeFi	Decentralized Finance
DEX	Decentralized Exchange
DAO	Decentralized Autonomous Organization
NEP	NEAR Enhancement Proposals
ERC-20	Ethereum Request for Comments 20
EVM	Ethereum Virtual Machine

HTTP	Hypertext Transfer Protocol
HTML	HyperText Markup Language
FTP	File Transfer Protocol
SMTP	Simple Mail Transfer Protocol
AWS	Amazon Web Services
GCP	Google Cloud Platform
IPFS	InterPlanetary File System
P2P	Peer-to-Peer
KYC	Know Your Customer
ICO	Initial Coin Offerings
PoW	Proof-of-Work
PoS	Proof-of-Stake

JWT
: JSON Web Token

CLI
: Command Line Interface

NLP
: Natural Language Processing

Wasm
: WebAssembly

Development of a Decentralized Social Media

Chapter 1

Introduction

The rapid advancement of blockchain technology has revolutionized various industries, offering decentralized, transparent, and secure solutions to long-standing challenges. This Master's Degree Thesis aims to provide an in-depth exploration of the development of a decentralized social media, the combination of NEAR Protocol [1], Non-Fungible Tokens, Artificial Intelligence and Chainlink Oracles. By exploring these topics in detail, I hope to contribute to the advancement of blockchain technology and its transformative impact on various sectors.

To seamlessly bridge the gap between the transformative impact of blockchain technology on various sectors and the introduction to NEAR Protocol, it's essential to highlight the pioneering role of Bitcoin and its underlying blockchain technology.

Bitcoin, introduced in a 2008 whitepaper by an entity known as Satoshi Nakamoto, revolutionized the concept of digital currency by introducing a decentralized blockchain network. This network is notable for its robustness, which is partially attributed to its network difficulty—a mechanism that adjusts the complexity of the mathematical problems solved by miners. This adjustment, which occurs approximately every two weeks, ensures consistent block generation times despite fluctuating mining power [2].

Furthermore, Bitcoin undergoes a process known as "halving" approximately every four years. This event reduces the reward for mining new blocks by half, thereby diminishing the rate at which new bitcoins are created and mimicking the scarcity of precious resources like gold. This feature is critical in controlling inflation and contributing to Bitcoin's value [3].

The inception of Bitcoin's network dates back to January 3, 2009, when the first node started running. This marked the beginning of a new era in digital currency and decentralized finance. The decentralized nature of Bitcoin means that it operates on a peer-to-peer network, where each node, or computer, participates in the validation and relay of transactions. This structure ensures that no single entity, including dictators or centralized institutions, can shut down the network or control the blockchain. This resilience against centralized control is a defining feature of blockchain technology and is crucial in understanding the evolution towards more advanced platforms like NEAR Protocol [4].

Bitcoin's blockchain has set the foundation for subsequent blockchain developments, showcasing the potential for a wide array of applications beyond just financial transactions [5]. Its impact is evident in the emergence of various blockchain platforms, each bringing unique features and improvements. The adaptability and robustness of Bitcoin's blockchain have inspired many innovations in the blockchain sphere, laying the groundwork for advanced platforms like NEAR Protocol, which further enhance the capabilities and efficiency of blockchain technology [6].

NEAR is a layer number one blockchain. Super secure and infinitely scalable! It uses a consensus protocol called "NightShade" [7]. It is a sharded proof-of-stake [8] consensus protocol that enables high throughput and low latency of transactions processing on the network. Faster, Cheaper and more securely. Validators secure the network by locking up NEAR tokens and earning rewards as a result. If the validators try to cheap the system, they lose their stake or the community votes them out. Users can delegate their tokens to a validator to earn rewards too. A more Eco-friendly platform. To make NEAR supper fast, it uses a technology called sharding. Ethereum developers can take advantages of NEAR's capabilities without having to leave the Ethereum Network (Rainbow bridge [9] and Aurora [10]). Octopus Network [11] allows anyone to build their own version of an application specific blockchain and connect it to NEAR's mainnet for speed, scalability and security.

The NEAR platform is also great for users, it simple wallet allows anyone to get started within seconds with minimal fuss no pop-ups, no switching networks, no crypto jargon, just a wallet that is effortless to use. Underneath the slick and simple exterior lies a ton of features that let developers create a seameless user experience, whenever they interact with the NEAR ecosystem. Also it is possible to register a NEAR wallet with a YOURNAME.near (Vanity Address). A vanity address allows you to have a more personalized public address, which can be handy for branding, recognition, or aesthetic purposes.

From NEAR wallets, you are able to manage your tokens and digital collectibles and explore DeFi apps, games, NFTs and more. The wallet is also the gateway to NEAR staking system which allows you to stake your NEAR coins with your validator of choice. By staking, you will not only earn a generous yield on your stake balance but you'll also be helping to secure and grow the NEAR network. After submitting the stake, the rewards will be distributed every 12 hours.

Most blockchains require every node or person stores all the information **ON-CHAIN** and process all transactions. while it makes the network secure, it also makes it slow. It takes time for every node to keep itself up to date! Sharding however doesn't have the same problems. it allows the blockcain to be secure, fast and scalable. Sharding is a way of partiotioning a database into smaller pieces or Shards and store it on different machines.

In the case of blockchain, nodes are devided into smaller groups responsible for one part of the chain's data. These individual groups can process transactions at the same time. Instead of one person verifying the transactions in one block, they are verified in multiple blocks by several people.

The more shards, the faster network goes. In NEAR's case it means it can process 100,000 transactions per second and beyond. Its unique design means that there is no limit to how many transactions it can process.

NEAR's approach to sharding is called "NightShade". It differs from the traditional approach to sharding where each shard produces its own batch of data called a chunk which will be added to other chunks to create a single block. A single validation produces each block by assembling all the created chunks into a block. While other sharded networks are harder to take part, in NEAR is making itself more decentralized.

Chunk only produces a lot of people with fewer NEAR and non-specialized computer have easy access to help keep NEAR secure. Dynamic re-sharding will adjust their number of shards based on user demand. This will allow the network to only pay for the infrastructure and scale at any given time.

NEAR needs a way to easily and securely exchange information and value with other blockchains. It achieves this through the Rainbow Bridge. The Rainbow Bridge allows users and developers to easily move their assets between NEAR and other platforms be a tokens or NFTs. This achieves two main things:

- It allows other chains users to avoid the congestion and high fees suffered by other blockchains by migrating their assets to NEAR. useful for users to access and move tokens between chains when needed.

- Provides developers with a tool that can be used to migrate users from Ethereum to NEAR through an easy to deploy bridge for their project's Token.

Currently, the Rainbow Bridge can be used for transferring between NEAR and Ethereum and supporting a huge range of ERC-20 tokens [12] including public Stable coins, Wrapped Tokens, DEX Tokens, Utility Tokens and more. The Rainbow Bridge will lock your tokens in a smart contract on the origin chain before transferring the same amount on the destination chain. This process takes just 7 minutes to move tokens from Ethereum to NEAR. It is permissionless and trustless.

More than 100 billions dollars worth of digital assets held by popular DeFi protocols. The problem lies within these project operating on the Ethereum blockchain which currently suffers from extreme congestion and exorbitantly high fees posing major road blocks to further adoption. Aurora is NEAR's answer to this problem as an ultra efficient EVM deployed on NEAR protocol. Aurora provides a platform, developers and users can use to operate their applications in a fully Ethereum compatible environments without compromise. Developers can deploy their code on Aurora with zero changes and benefit from NEAR's speed, security and scalability.

Chapter 2

Blockchain Fundamentals

Blockchain technology, as a cornerstone of the digital revolution, has gained immense popularity due to its distributed, decentralized, and immutable ledger system. This system records transactions across a network of computers, with each transaction securely linked to the previous one, thus forming an unbroken chain of blocks. This framework ensures transparency, security, and trust among network participants and negates the need for a centralized authority for validation and ledger maintenance [13].

A notable implementation of blockchain technology is Bitcoin, which transcends its role as a digital currency. While often associated with trading and investment, Bitcoin's design and philosophy offer much more. It serves as an educational tool for enthusiasts to understand economic principles, enabling transactions ranging from purchasing everyday items like medicine and housing to paying application fees. The conception of Bitcoin is rooted in the principle of financial freedom, advocating for a system that is accessible and understandable to all, thereby democratizing economic participation [2].

Furthermore, Bitcoin has paved the way for the emergence of various alternative cryptocurrencies (altcoins), each designed with specific use cases and target audiences. These altcoins aim to facilitate everyday transactions for people globally, effectively covering the living costs and daily expenses of individuals. This expansion signifies the adaptability of blockchain technology to meet diverse needs and preferences.

Bitcoin's multifaceted nature extends to trading and education, reflecting its significance in financial markets. Its design principles, emphasizing decentralization and user autonomy, have significantly impacted financial marketing, challenging traditional paradigms and offering new avenues for economic engagement. The influence of Bitcoin thus exemplifies a broader trend in blockchain technology, which has evolved from its initial financial applications to a robust, versatile tool for various global industries, including supply chain management, healthcare, and voting systems [3].

This evolution of blockchain, spearheaded by Bitcoin, underscores its potential to transform not only the financial landscape but also to educate and empower individuals in their economic decisions, thus fostering a more inclusive and informed society. The principles of decentralization, transparency, and user empowerment inherent in blockchain technology are not only pivotal in the financial domain but also serve as the foundational elements of the next phase of the World Wide Web, known as Web 3.0.

Web 3.0 represents a paradigm shift in how we interact with the internet, championing the values of decentralization and user sovereignty. As we transition from the blockchain's influence on specific industries to its integration within the broader fabric of the internet, it becomes evident how these technologies are converging to redefine digital interaction and ownership. This convergence is at the heart of Web 3.0, which aims to leverage the decentralized, trustless nature of blockchain to create a more equitable and user-centric online experience. In the following sections, we explore the evolution of the Web and how blockchain technology is a driving force behind the transformative vision of Web 3.0. Before delving into its specifics, it's essential to understand its background. The World Wide Web has progressed through three distinct phases:

- **Web 1.0 (1991-2004):** This was the Web's initial phase. During this period, internet access was slow, costly, and dominated by static pages. The majority of users were passive consumers, with only a select few producing content. Key features of this era included limited use of JavaScript, scarce media content, and the absence of social media. Despite its limitations, this generation was marked by its decentralized nature. Users either ran their own servers or relied on various hosting providers. Open protocols, such as HTTP, HTML, FTP, and SMTP, were foundational.

- **Web 2.0 (2004-present):** This era witnessed a substantial shift in internet usage. As the internet became more accessible and affordable, its reach extended to the average person. This influx of users prompted businesses to offer a plethora of online services, from shopping to payments. The web content grew richer and more interactive. A significant shift was users transitioning from mere content consumers to active content creators. The rise of social media, the invention of smartphones, and the web's expansive growth characterized this period. However, this growth also led to increased centralization. This centralization presented challenges, such as issues with digital ownership. Users found that they didn't truly own their digital assets, as centralized entities retained control. This era also saw a compromise in user privacy, as personal data became a currency for many online platforms.

- **Web 3.0 (Future):** This is the forthcoming phase of the Web. It's still nascent, and its exact trajectory remains uncertain. However, some defining characteristics are beginning to emerge:
  - Emphasis on decentralization and privacy.
  - Transition of digital ownership from corporations to individual users.
  - Trustless and permissionless operations.
  - Community-driven governance and tokenomics.

In this dissertation, we shall elucidate how these fundamental tenets of Web 3.0 manifest in practical terms, demonstrating how to construct an application wherein digital assets are decentralized and unequivocally owned by the end users.

2.1 Decentralization in Web Development

2.1.1 Centralized Internet

The present-day Internet is largely centralized, predominantly relying on client-server architecture [14] hosted on major cloud platforms like AWS, Azure, and GCP. Remarkably, 90% of mobile traffic is directed to these cloud services [15], signifying a concentration of control within a few corporations.

2.1.2 File Sharing Protocol & P2P Communication

Decentralization first made its mark with file sharing through the introduction of the BitTorrent protocol. This peer-to-peer (p2p) system promotes genuine decentralization by allowing distributed data storage without central oversight. While it sparked controversies, its principles paved the way for contemporary decentralized storage systems such as IPFS and FileCoin.

The concept of decentralized, cryptographic control of digital data was notably advanced by Stuart Haber and W. Scott Stornetta in their seminal work "How To Time-Stamp a Digital Document" [13]. They proposed a system for time-stamping digital documents, forming a chain of cryptographic proofs that laid the groundwork for blockchain technology. This concept of chaining blocks of data is a foundational element in understanding blockchain's immutability and integrity.

Further improvements in digital time-stamping were proposed by Dave Bayer, Stuart Haber, and W. Scott Stornetta, as outlined in "Improving the Efficiency and Reliability of Digital Time-Stamping" [16]. Their work enhanced the practicality and efficiency of the cryptographic process, contributing to the robustness of blockchain technology.

Adam Back's "Hashcash - A Denial of Service Counter-Measure" [17] introduced the proof-of-work system to combat spam emails, which later became integral to Bitcoin's transaction verification process. This system requires a certain amount of computational work, deterring frivolous or malicious uses of computing power, and is a cornerstone in the functionality of decentralized networks.

Historically, monetary transactions were overseen by central entities like banks. This paradigm shifted with Bitcoin, the pioneering cryptocurrency. Drawing from the principles of BitTorrent and the aforementioned foundational works, Bitcoin utilizes p2p communication for its transaction ledger, preserved in a blockchain. Mainly, the blockchain's architecture solves the double spending problem. By using the proof-of-work consensus mechanism, Bitcoin solves in one shot the double spending problem and minting of coins in a process called "mining".

Building on Bitcoin's success, numerous cryptocurrencies emerged, with Ethereum standing out. Ethereum expanded the blockchain concept to house diverse data types [18], including code. With data and code stored in a decentralized manner, Ethereum introduced the capability to execute this code, leading to the creation of Smart Contracts. We'll delve deeper into blockchains, smart contracts, and their potential for crafting dApps.

2.2 Blockchain basics

To establish a foundational understanding, let us provide a concise overview of the fundamental concepts of blockchain and smart contracts.

2.2.1 Concepts of Blockchain & Smart Contracts

In traditional Web 2.0 applications, the backend framework is fundamentally anchored on two cornerstones: a database for data preservation and a server for executing code. When transitioning to Web 3.0, these foundational elements persist but metamorphose in their characteristics. The conventional database is supplanted by a blockchain, and where we once relied on standard servers, we now leverage the capabilities of smart contracts.

The underlying architecture of a blockchain can be visualized as a connected sequence, or chain, of transactional records. For efficiency, transactions are clustered into blocks. A pivotal aspect of this design is the cryptographic linkage between blocks: each block encapsulates the cryptographic signature of its predecessor. This configuration imparts a crucial trait to the blockchain - any attempt to alter a transaction mandates a change in its cryptographic signature, thereby rendering subsequent transactions in the chain invalid. This inherent tamper-resistance renders the blockchain ideal for decentralized data storage, ensuring easy validation of transactional integrity by any party.

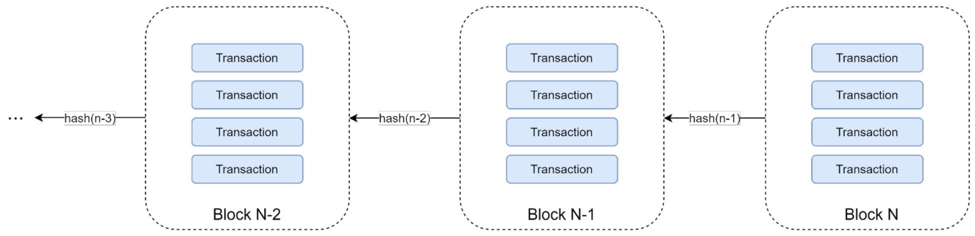


Figure 2.1: The integrity of transactions on a chain

2.2.2 Consensus Mechanism

Blockchain, resembling a decentralized transaction log, continuously grows by adding new transactions only [19]. It can be thought of as a distributed form of the event sourcing pattern, where each transaction represents a distinct event [20]. Operating without a central server, blockchain relies on a consensus mechanism for block creation, data synchronization, and network participation [21]. We'll explore various consensus protocols in more detail in subsection **3.6.4**. Notably, blockchain transactions are transparent, necessitating the encryption of sensitive information before its inclusion.

2.2.3 Blockchain Nodes

So, what's the mechanism for appending transactions to the blockchain? This responsibility rests with the Blockchain Node. Individuals can set up their own nodes, join the blockchain's peer-to-peer network, and propose new transactions. Furthermore, this node grants access to the existing blockchain data.

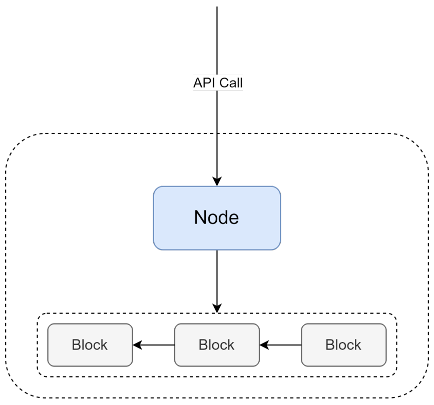


Figure 2.2: Connected Node to the P2P Blockchain network

Blockchain transactions can vary widely, their nature being dependent on the specific blockchain network in question. For instance, Bitcoin, the pioneering blockchain system, primarily managed a ledger of transactions. Its transactions were simple, denoting transfers of funds between entities. Though this design was revolutionary and follows the original goals of Bitcoin creators it is not adapt well to more modern applications to Decentralized Finance (DeFi) Applications. This is where smart contracts step in.

2.2.4 Smart Contract's Unique Attributes

To those familiar with the Web 2.0 framework, think of smart contracts as analogous to serverless functions. Instead of operating on standard cloud servers, they run on blockchain nodes. Yet, smart contracts have unique attributes:

- Pure Operation: They act as pure functions, taking in the current state (recorded on the blockchain) and arguments from the caller, and then outputting an altered state: F*(state, args)* -> state. This implies no off-blockchain calls, like API requests, are allowed. This design choice ensures decentralized consistency, where various network nodes can run the contract and consistently achieve the same outcomes.

- Transparency: Smart contracts are inherently open-source, granting anyone the ability to review the code and validate its operations.

- Permanence: Post-deployment, the code of a smart contract becomes a permanent fixture on the blockchain, resisting any modifications. While there are methods to update them, such techniques are specific to individual chains.

2.2.5 Backend Structure of a dApp

Drawing a real-world parallel, smart contracts are akin to legal contracts: they're permanent, behave predictably, and are open to all involved parties. In essence, smart contracts are digital renditions of these legal agreements, but coded.

A lingering question remains: If we're restricted to initiating transactions, how do we deploy and run smart contracts? There are two primary transaction categories that enable this:

1. Deployment Transaction: Through this, the smart contract's code is deployed, making it a permanent part of the blockchain's data.

2. Function Call Transaction: Triggered with certain arguments, this transaction type activates a smart contract, yielding an updated state.

When a node receives a Function Call transaction, it fetches the contract's code and state from the blockchain, runs the code, and then logs the updated state back onto the blockchain as a new transaction.

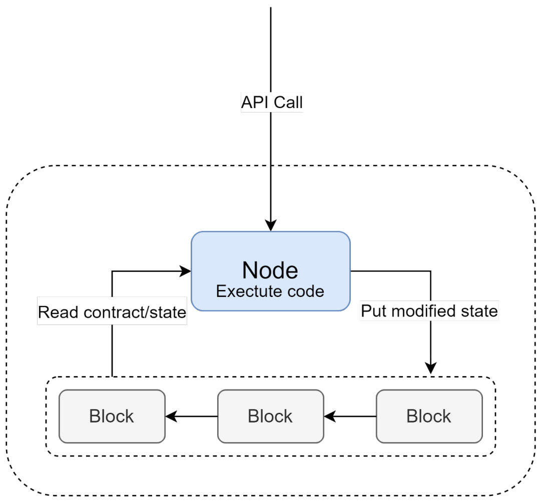


Figure 2.3: Execute the code through the Node (Read & Put states)

While we've delved into the backend structure of a dApp, the client-side remains to be addressed. Using an API interface, dApps can communicate with diverse clients, mirroring the Web 2.0 model, which includes web, mobile, desktop, and even other server platforms. However, user identity in dApps differs significantly from traditional setups.

In standard Web 2.0 systems, servers control and own user identities, dictating access to their services. Conversely, blockchain operates without such centralized controls, allowing open interactions (excluding certain private blockchains which we'll not discuss here).

2.2.6 dApp User's Authentication

A natural question arises: Without a conventional login or registration, how do dApps authenticate users? The answer lies in public key cryptography. In this system, a public key represents a user's identity, akin to a username, while a private key functions similarly to a password. However, instead of typical login methods where servers verify credentials and issue access tokens, users in dApps sign transactions using their private key. Consequently, traditional identifiers like usernames or emails are absent, a crucial aspect for developers to note, especially for applications necessitating Know Your Customer (KYC) processes [22].

Furthermore, due to the complexity of private/public key pairs, they aren't as easily remembered as standard usernames or passwords. To address this, specialized software called wallets are employed. These wallets safeguard users' key pairs, facilitate transaction signatures, and can share keys with other applications when needed.

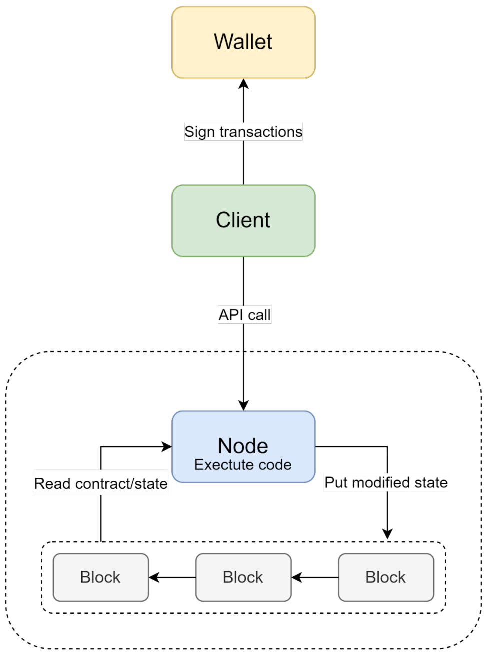


Figure 2.4: Client-Side of dApps and User Identity

2.2.7 Financial Dynamics

In the Web 2.0 paradigm, the financial dynamics are straightforward: users compensate the service providers either directly through monetary payments or indirectly by offering access to their data, sometimes both. Service providers, in turn, bear the costs of the infrastructure.

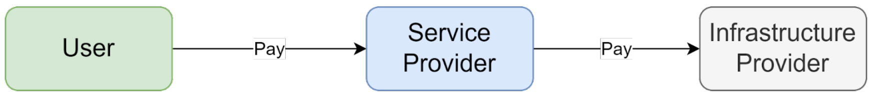


Figure 2.5: Web 2.0 Cost Dynamics

However, the Web 3.0 model revolutionizes this dynamic. Here, users transact directly with infrastructure providers, i.e., the nodes operating on the blockchain, sidelining the traditional service providers.

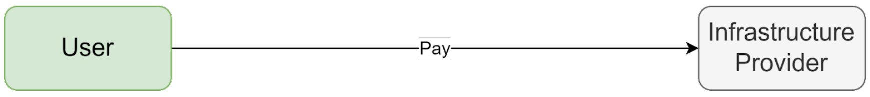


Figure 2.6: Web 3.0 Cost Dynamics

Such a shift brings about profound implications:

- Endurance of Services: Since applications are anchored on the blockchain, service providers lose the authority to terminate or limit their services. The apps, in essence, achieve immortality, persisting as long as the blockchain network thrives.

- Redefining Monetization: Given that users aren't directly compensating the service providers, a novel monetization approach is necessitated. For instance, specific fees might be embedded within smart contracts for certain operations.

- No Hidden Costs: While Web 2.0 services often appeared free, they weren't transparent about costs. In the Web 3.0 realm, though service providers might subsidize initial costs to ease user onboarding, the onus eventually falls on the users to settle the bills.

But how do these payments transpire? Traditional payment mechanisms, like credit cards, don't align with the decentralized ethos of blockchains. The alternative? Cryptocurrencies. Each blockchain network introduces its proprietary digital currency facilitating intra-network transactions.

Consequently, whenever a user triggers an action on the blockchain, say by invoking a smart contract, they must cover both the infrastructural costs and potentially an additional service fee to the provider.

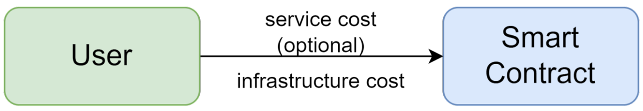


Figure 2.7: Service Costs in Blockchain Transactions

This infrastructure fee, commonly termed as ***"gas"***, encompasses two main components:

1. Computational Cost: This pertains to the computational resources expended to incorporate a transaction into the blockchain.

2. Storage Cost: This relates to the additional storage requisites essential for every transaction.

The lingering conundrum is: How do users procure these cryptocurrency tokens to begin with? One viable route is purchasing from existing token holders using conventional currency or another cryptocurrency. Platforms like Binance facilitate such exchanges. However, this mechanism is contingent on a pre-existing pool of circulating tokens.

Generating and augmenting this token supply hinges on the blockchain's consensus protocol. As highlighted earlier, this protocol serves to incentivize participation within the blockchain network. But how does it operationalize this? Each node engaged in transaction processing is duly rewarded:

$$\textbf{reward} = infrastructureCostReward + coinbaseReward$$

Here:

- **infrastructureCostReward** represents the portion of the infrastructure cost remunerated by users for transactions.
- **coinbaseReward** denotes newly minted cryptocurrency tokens, expressly designed to reward transaction-processing nodes.

Effectively, with every processed transaction, a minuscule volume of cryptocurrency is birthed, causing a gradual increment in circulating cryptocurrency over time. Naturally, an initial token volume is essential to jumpstart the network, often achieved via methods like Initial Coin Offerings (ICO) [23], This will be discussed in greater detail in Chapter **3**.

2.3 Dominant Consensus Mechanisms in Modern Blockchain

Currently, two primary consensus mechanisms dominate the blockchain landscape:

- **Proof-of-Work (PoW) [24]:** This is the pioneer consensus model, adopted by trailblazers like Bitcoin. However as we mentioned this consensus was not projected for dApps.
- **Proof-of-Stake (PoS) [8]:** Emerging as a contemporary alternative, PoS operates without the need for intense computational efforts, sparing graphic cards in the process. Here, transaction processing is often termed "validation". Modern blockchains, such as NEAR, have embraced PoS. Even Ethereum is in the midst of transitioning to this model. A significant advantage of PoS is its cost-effectiveness, coupled with swifter transaction processing.

Armed with this foundational understanding, we're poised to delve into the subsequent segment.

2.4 Selecting the Ideal Blockchain to implement dApps

The blockchain landscape is vast and varied, making the task of pinpointing the perfect fit for specific requirements a daunting one. To streamline this process, let's identify key criteria essential for a blockchain:

1. Consensus Algorithm: Proof-of-Stake (PoS) emerges as a more efficient alternative for dApps, with other novel algorithms being comparatively less tested.

2. Transaction/Storage Cost: A lower cost directly translates to user benefits, given that users foot these bills.

3. Transaction Speed: Quicker transaction times lead to enhanced user experiences.

4. Scalability: A network's ability to handle a burgeoning transaction volume is vital. Otherwise, transaction speed and cost might spiral uncontrollably.

5. Development Experience: This pertains to the choice of programming language for crafting smart contracts. While Ethereum championed Solidity [25], newer chains like NEAR have gravitated towards Rust, a seasoned general-purpose language.

Historically, Ethereum blazed the trail by pioneering smart contracts. Yet, as its user base swelled, transaction costs and speeds became prohibitive, revealing its limitations. This gave rise to scaling solutions [26] like layer 2 chains [27], sidechains [28], and plasma chains [29]. Each, however, came with its own set of challenges. Ethereum's switch to a PoS consensus, intended to rectify core issues, remains a work in progress without a definitive completion date.

This backdrop paved the way for next-gen blockchains, engineered to be nimble, affordable, and scalable, having gleaned insights from Ethereum's challenges. Selecting the best among them is intricate. For our purposes, the NEAR blockchain [1] emerges as the front-runner, attributed to its:

- Cost-effective and rapid transactions.

- Inherent design for extensive scalability, ensuring consistent transaction cost and speed.

- Adoption of the PoS consensus mechanism, eliminating the need for resource-intensive mining.

- Preference for Rust as the chief programming language [30], a widely-accepted and cherished language [31], simplifying the recruitment of adept developers.

Subsequent sections will delve deeper into NEAR as the foundational blockchain. Prior to immersing ourselves in the nuances of Web 3.0 transition, a comprehensive examination of NEAR is imperative.

Chapter 3

NEAR Protocol & Smart Contracts

The NEAR Protocol, a significant player in the blockchain space, was launched with a vision to improve upon the scalability and usability issues prevalent in earlier blockchain systems. The project was founded by Alex Skidanov and Illia Polosukhin, who brought together their extensive experience in programming and blockchain technology. The protocol mainnet went live and produced its first block, called the genesis block, on April 22, 2020. Although its validators were run entirely by the NEAR Foundation through Proof of Authority early in the blockchain's life, the "training wheels" were removed in September 2020. According to the developers, this allowed the Foundation to work out issues with the platform before allowing the community to take over as validators to drive decentralization. [32]

Since its inception, NEAR has raised funding through multiple mechanisms to support its development. In August 2020 the team raised over $33 million through an initial coin offering (ICO), and in the first half of 2022 the team raised $500 million from VC firms such as FTX Ventures and a16z. [32]

In terms of network difficulty, NEAR Protocol employs a unique consensus mechanism known as 'Nightshade,' designed to enhance scalability and reduce transaction costs. Unlike Bitcoin, NEAR does not follow a halving schedule. Instead, it utilizes a different approach to control token supply, involving a process known as 'token burning.' In this system, a portion of transaction fees is permanently removed ***('burned')*** from circulation, which helps in regulating the total supply of NEAR tokens over time.

The NEAR Protocol documentation [33] serves as a robust guide for newcomers. This chapter intends to elucidate fundamental concepts, ensuring the subsequent sections are comprehensible even without prior familiarity with NEAR.

3.1 Accounts & Transactions

NEAR's account system stands distinct from contemporaries like Bitcoin or Ethereum. Rather than solely relying on *public/private* key pairs for user identification, NEAR elevates accounts [34] to primary entities, leading to:

- User-Friendly Identification: Instead of public keys, users employ intuitive account names.
- Enhanced Security: Multiple key pairs with varied permissions [35] can be associated with an account. This ensures that losing a single key pair doesn't jeopardize the entire account.
- Organized Account Structure: NEAR supports hierarchical account structures, facilitating the management of multiple smart contracts under a singular parent account.
- Transaction-Based Creation: Accounts and public keys materialize through transactions, as they are blockchain-stored entities.

3.1.1 Transaction (Action) Types

Transactions drive the NEAR Protocol. Though NEAR features a singular transaction type [36], it can encapsulate diverse actions. Typically, a transaction encompasses a single action, and for brevity, we'll use *"action"* and *"transaction"* interchangeably henceforth. Each transaction is associated with a sender and a recipient. Supported transaction (action) types include:

- **CreateAccount/DeleteAccount, AddKey/DeleteKey:** Accounts and key management transactions.
- **Transfer:** Basic operations facilitating NEAR token transfers between accounts.
- **Stake:** Essential for validators in a Proof-of-Stake blockchain. This topic is expanded upon separately [37].
- **DeployContract:** The DeployContract action deploys smart contracts. Each account can host one contract, uniquely identifiable by its account name. Contract updates are initiated if the DeployContract action targets an account with an existing contract.
- **FunctionCall:** The FunctionCall action, critical to the blockchain, facilitates smart contract function invocation.

3.1.2 Smart Contract Invocation Methods

Smart contracts in NEAR are crafted in Rust or JavaScript and are translated into WebAssembly (Wasm) [38]. These contracts encompass methods that can be invoked using the FunctionCall transaction. Each method call comprises the target account id, method name, and relevant arguments.

There exist two mechanisms to call a smart contract method:

1. Initiating a FunctionCall Transaction: This creates a new blockchain transaction, potentially altering the contract's state.

2. Executing a Smart Contract View Call: NEAR's RPC nodes [39] offer an API enabling the execution of state-unchanging ***(readonly)*** methods.

The latter is preferable due to its cost-efficiency, especially when leveraging public nodes that are free. Moreover, view calls don't necessitate an account, proving advantageous for client-side application development.

3.2 Nodes & Validators

3.2.1 Types of Nodes

In the NEAR network, nodes play an integral role in ensuring the smooth operation and security of the blockchain. There are primarily three types of nodes:

- **Validator Node:** These nodes are fundamental to the operation of the NEAR blockchain. They are responsible for participating in the consensus mechanism, producing blocks and chunks. The health of the network heavily relies on the Validator nodes. A real-time overview of these nodes can be viewed on the NEAR Explorer [40].

- **RPC Node:** RPC nodes offer RPC services and are vital for developers. While the NEAR Foundation provides a public RPC endpoint that is freely accessible, participants are also encouraged to set up their own RPC nodes.

- **Archival Node:** These nodes retain the complete blockchain data, constructing archives of historical states. They are particularly valuable for infrastructure providers, chain analysis, and block explorers.

3.2.2 Validators & Network Security

Role of Decentralization

The decentralized nature of the NEAR network ensures its safety through collaboration amongst multiple validators. Validators guarantee that all network transactions are legitimate, thereby preventing malicious activities like money theft.

Consensus Mechanism - Thresholded Proof of Stake

The network operates using a variant of the Proof-of-Stake consensus mechanism, known as Thresholded Proof of Stake [41].

Trust & Staking

In the Proof-of-Stake model, users indicate their trust in specific network validators by delegating NEAR tokens to them, a process termed as ***"staking"***. The premise is that a validator with a significant amount of tokens delegated to them is considered trustworthy by the community.

Validators have two primary roles:

1. Ensure the validation and execution of transactions, subsequently grouping them into blocks.

2. Monitor other validators to prevent the generation of invalid blocks or the creation of alternative chains aiming at double-spending.

Consequences of Misconduct

Misconduct by a validator leads to ***"slashing"***, where a portion or all of their staked tokens are burned. Any malicious attempt to manipulate the chain would necessitate controlling a majority of the validators simultaneously, which is financially risky due to the potential of slashed tokens.

3.2.3 Economy of Validators

Compensation Structure

Validators are compensated for their services to the network. They receive a predetermined amount of NEAR tokens every epoch. This amount is set so that it corresponds to 4.5% of the total supply on an annual basis. Additionally, all transaction fees collected within an epoch, excluding the portion allocated for contract rebates, are burned. Regardless of the fees collected or burned, validators receive their inflationary reward at a constant rate.

Epochs Defined

An epoch represents a fixed duration during which the validators of a network remain unchanged and is quantified in terms of blocks. On both the testnet and mainnet, an epoch spans 43,200 blocks. Although the target duration for an epoch is approximately 12 hours, based on the one-second block creation rate, practical scenarios may see a slightly prolonged duration. This epoch-related information can be accessed via the *protocol_config* RPC endpoint [42] under the parameter ***'epoch_length'***.

Block Retention & Archival Nodes

It's crucial to note that, by design, nodes discard blocks post the completion of 5 epochs, roughly equating to 2.5 days, unless they function as archival nodes [39].

3.2.4 Overview of Validators

Validators [43] shoulder the responsibility of block production and network security.

Hardware Prerequisites

Owing to their role in validating all shards, there are high prerequisites for running a validator node, including an 8-Core CPU, 16GB RAM, and 1 TB SSD storage. The estimated monthly cost for hosting a block-producing validator node is approximately $330.

Active Validators & Seat Price

The current active validators can be accessed on the NEAR Explorer [40]. The minimum seat price for a block-producing validator is based on the 300th proposal. However, if there are more than 300 proposals, the threshold becomes the stake of the 300th proposal, provided it surpasses the minimum threshold of 25,500 $NEAR. The live seat price updates are available on the NEAR Explorer, and any validator nodes staking an amount higher than the seat price can become part of the active validator set.

Future Computational Considerations

Regarding the computational resources, as of now, the NEAR network primarily relies on CPU computations. The consideration of incorporating GPU compute capabilities for validator nodes remains a topic for future discussions.

Chunk-Only Producers

Chunk-Only Producers [43] are a novel addition to the NEAR Ecosystem, designed to promote increased participation and decentralization. These producers focus solely on generating chunks—specific [7] portions of a block from a shard [44]. The design behind Chunk-Only Producers acknowledges the fact that not every participant may have the hardware capabilities to validate the entire network, hence they can operate with an 8-Core CPU, 16GB RAM, and a 500 GB SSD. Economically, they are poised to receive at least 4.5% annual rewards, with potential for more based on the staking percentage of the network's tokens. For comprehensive information about Validators and Nodes, the Dedicated Validator Documentation Site is available [45].

Ensuring Security & Smart Contract Updates

Importantly, the NEAR protocol ensures that validators are insulated from risks associated with vulnerable or malicious dApps. While the onus of securing dApps rests with their developers, the NEAR platform uniquely facilitates smooth updates to smart contracts, ensuring rapid response to any vulnerabilities.

3.3 Gas & Storage

3.3.1 Gas and Its Implications

The computational expense associated with each transaction in the blockchain world is termed ***"Gas"***, quantified in gas units [46]. When a transaction is initiated, it carries a specific gas amount to defray its cost. Simple transactions enable pre-determined gas calculations, but for FunctionCall transactions, an overestimation is customary, with any surplus refunded.

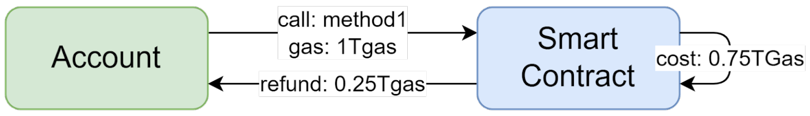


Figure 3.1: Covering Computational Costs

Rather than directly using NEAR tokens, separate gas units are employed to adapt to the fluctuating infrastructure costs. As the network matures, the gas unit's value might shift, yet the gas quantity for a transaction remains consistent.

3.3.2 Cost Storage & Mechanism

Distinct Storage and Gas Systems

In addition to computational costs, many smart contracts demand storage. NEAR's storage system is distinct from its gas mechanism. While gas is relatively affordable, storage incurs a hefty price. This necessitates prudent storage budgeting, with only essential data reserved on the blockchain and supplementary data stored off-chain.

The Staking Model

Contrary to purchasing storage, NEAR adopts a leasing model, termed ***"staking"***. When a smart contract necessitates data storage, the corresponding NEAR token amount is ***"locked"***. Upon data removal, these tokens are released. It's vital to note that these tokens are retained in the smart contract's account, exempting the user from direct payment.

FunctionCall Transactions and Deposits

To facilitate users in bearing storage expenses or any contract-associated fees, NEAR permits token transfers through **FunctionCall** transactions. This feature, known as a deposit, ensures that smart contracts verify the attached token amount, either authorizing actions or reimbursing excessive tokens.

Benefits of Gas Fees and Deposit Attachments

Leveraging gas fees and deposit attachments, contracts can be designed to incur zero developer costs, enduring indefinitely on the blockchain. Intriguingly, 30% of the gas fees expended on contract execution are credited to the contract's account [43]. Though this primarily benefits high-demand contracts, it's a valuable feature.

DeployContract Transactions

Lastly, it's essential to recognize that **DeployContract** transactions, responsible for storing smart contract codes on the blockchain, also incur storage fees.

Given the potential size of smart contracts, their optimization is crucial. Some recommendations include:

- Avoid Rust code compilation on Windows due to its large output. Opt for WSL or other operating systems.
- Prioritize smart contract code size optimization [47].

For a comprehensive understanding of the storage model, the official documentation is recommended [48].

3.3.3 Gas as a Developer Incentive

Unique to NEAR, the gas is not solely utilized for compensating validators. When a transaction calls a contract, the incurred gas fee is divided in a specific manner: 30% is allocated to the smart contract, and the remaining 70% is annihilated. This arrangement provides an incentive for developers to actively participate and innovate within the NEAR ecosystem.

3.3.4 The Concept of Free Transactions

NEAR introduces an unparalleled feature that facilitates the invocation of *read-only* methods in smart contracts without any associated costs. This mechanism is especially advantageous as users do not require a NEAR account for this operation. In such transactions, the validators bear the gas costs, emphasizing the user-friendly nature of the NEAR platform.

3.3.5 Understanding Gas Units & Gas Price

Transactions on the NEAR platform incur a nominal fee in $NEAR, payable upfront. However, this fee isn't directly assessed in $NEAR. Internally, the platform utilizes gas units, which are deterministic in nature. This ensures that a specific operation consistently incurs the same gas cost. The actual transaction fee in $NEAR is ascertained by multiplying the aggregated gas from all operations by a dynamic gas price. This price is recalibrated after each block, contingent on network demand. If the preceding block exceeds half its capacity, the price escalates; otherwise, it diminishes, with a cap on its fluctuation set at 1% per block. The lowest bound for the gas price is currently set at 100 million yocto NEAR.

3.3.6 Correlating Gas to Computational Resources

The determination of gas units has been meticulously executed, resulting in:

- 1 **TGas** (10^{12} gas units) approximating to 1 millisecond of computational time.
- This represents 0.1 ***milliNEAR*** (considering the minimum gas price).

This approximation, albeit rough, offers a useful metric. It's imperative to acknowledge that gas units represent not just computational time but also account for bandwidth and storage. System parameters could undergo modifications in the future, potentially altering the correlation between TGas and computational time.

3.3.7 1S Block Production & Associated Costs

To maintain the network's efficiency, NEAR enforces a maximum gas cap per block, ensuring that a block is produced approximately every second. A table showcasing the cost of prevalent actions in TGas and milliNEAR (considering the minimum gas price) has been provided, elucidating the estimated costs. Transaction types like **AddKey**, **DeleteKey**, and **FunctionCall** inherently possess higher gas costs, necessitating strategic consideration.

Operation	TGas	fee (mN)	fee (N)
Create Account	0.42	0.042	4.2×10^{-5}
Send Funds	0.45	0.045	4.5×10^{-5}
Stake	0.50	0.050	5.0×10^{-5}
Add Full Access Key	0.42	0.042	4.2×10^{-5}
Delete Key	0.41	0.041	4.1×10^{-5}

Table 3.1: Transaction Costs in NEAR

How to buy Gas?

Gas on the NEAR platform functions differently than some might expect from experiences with platforms like Ethereum. Instead of purchasing gas directly, users attach tokens to transactions. Unlike Ethereum, where users can pay extra to expedite their transaction processing, NEAR's gas costs are fixed. Basic operations, such as transfers, have predictable gas amounts that are automatically attached. However, Function Calls are more intricate, necessitating users to specify the gas amount, with a cap currently set at 300 Tgas. Notably, this cap is flexible and can be determined via the ***protocol_config*** RPC endpoint.

Attach extra gas; get refunded!

The actual token cost for these gas units varies. If a user overestimates and attaches more tokens than required for the gas, the excess is refunded. This refund mechanism also applies to basic operations, ensuring users only pay for the gas they use.

Prepaid Gas

To enhance user onboarding, NEAR offers a form of **"prepaid gas"**. Developers can set up their applications so that newcomers can pull gas funds from a developer-controlled account. Once familiarized, these users can then fund their own transactions. This system allows developers to essentially pay for their users' gas fees. By utilizing Function Calls and setting up specific access keys, developers can onboard users without them needing to go through a traditional wallet setup. However, it's crucial to note that while NEAR doesn't limit the use of developer funds for this purpose, developers can set allowances on these access keys, ensuring controlled spending.

3.4 Clients Integration

While we've delved into the client-agnostic execution of smart contracts, real-world applications necessitate Client-Side interactions, be it through web, mobile, or desktop platforms. The NEAR Wallet [49], currently the sole official wallet, facilitates this via HTTP redirects, which is straightforward for web applications (JavaScript SDK is available) but can demand intricate approaches like deep linking for other platforms.

3.4.1 Access Keys

Standard Transaction Flow

Whenever a user wants to post a transaction, the client redirects them to a wallet. Here, the transaction is approved, and the wallet then returns a signed transaction back to the client through a redirect. This method ensures the private key remains hidden from the client, bolstering security.

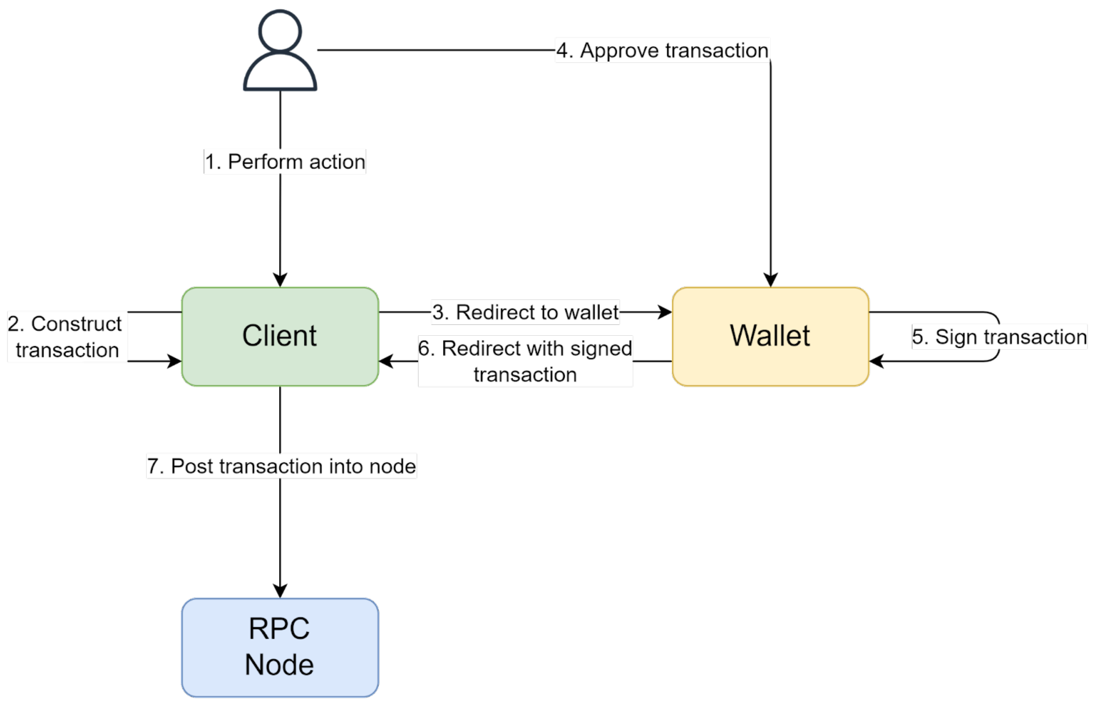


Figure 3.2: Login Flow for Obtaining a Functional Call Key

Challenges

The constant redirection can become cumbersome for users. This is particularly true for transactions that only require minimal gas fees, like calling smart contract functions.

Introduction to Access Keys

To address the above challenge, NEAR introduced two types of Access Keys [35].

Full Access Keys: These keys are versatile and can be employed to sign any kind of transaction. As the name suggests, they provide complete access.

Functional Call Keys: Tailored to enhance user experience, these keys:

- Are specific to a particular contract.
- Possess a dedicated budget for gas fees.
- Cannot be utilized for transactions that transfer NEAR tokens (i.e., payable transactions).
- Are solely designed to cover gas fees, making them less security-sensitive. As a result, they can be securely stored on the client side.

Simplified Signing Flow

Owing to the Functional Call Keys' design, NEAR can offer a streamlined signing process for non-payable transactions. This process begins with a login flow to obtain a Functional Call key.

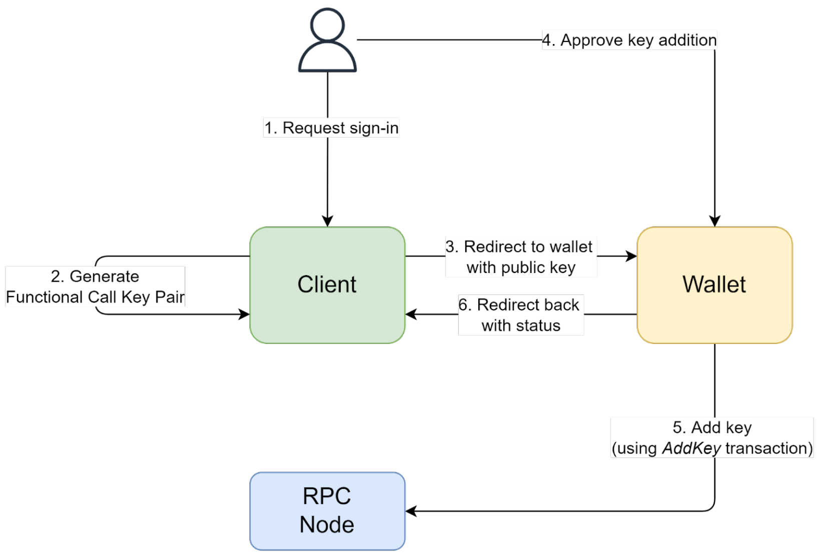

4. Approve key addition
1. Request sign-in
2. Generate Functional Call Key Pair
Client
3. Redirect to wallet with public key
Wallet
6. Redirect back with status
5. Add key (using *AddKey* transaction)
RPC Node

Figure 3.3: Combined Usage of Functional Call Key and Wallet Redirection

The client generates a new key pair and asks a wallet to add it as a Functional Call Key for a given contract. After this, a login session is established and considered alive until the client has the generated key pair. To provide the best user experience usage of both keys is combined - type of signing is determined based on a transaction type (payable or non-payable). In case of a payable transaction, flow with wallet redirection is used, otherwise simplified local signing flow (using a stored Function Call Key) is applied:

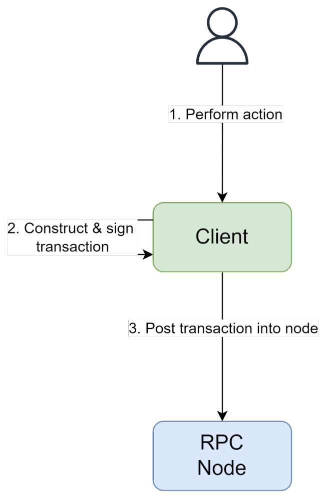

1. Perform action
2. Construct & sign transaction
Client
3. Post transaction into node
RPC Node

Figure 3.4: Generating a Full Access Key

It's important to note that it's possible to generate a Full Access key using the same key addition flow as for the Functional Call key, but this is very dangerous since compromise of such key will give full control over an account. Applications that want to work with Full Key directly should be designed with extreme care, especially in the matters of security.

3.5 Cross-Contracts Calls

The true power is achieved when smart contracts are working in concert and communicating with each other. To achieve this, NEAR provides cross-contract calls functionality, which allows one contract to call methods from another contract. The general flow looks like this:

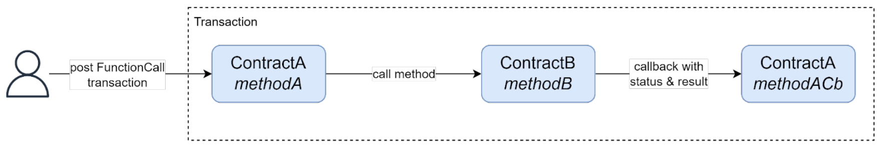


Figure 3.5: Cross-Contracts General Flow

However, certain intricacies arise:

- In order to provide a call status (success or failure) and a return value to the calling contract, a callback method should be called, so there's no single activation of **ContractA**. Instead, an entry method is called first by the user, and then a callback is invoked in response to cross-contract call completion.

- Transaction status is determined by the success or failure of a first method call. For example, if a **ContractB.methodB** or **ContractA.methodACb** call fails, the transaction will still be considered successful. This means that to ensure proper rollbacks in case of expected failures, custom rollback code must be written in the **ContractA.methodACb**, and the callback method itself must not fail at all. Otherwise, smart contract state might be left inconsistent.

- Cross-contract calls must have gas attached by the calling contract. Total available gas is attached to a transaction by a calling user, and distributed inside the call chain by contracts. For example, if 15TGas are attached by the user, **ContractA** may reserve 5TGas for itself and pass the rest to **ContractB**. All unspent gas will be refunded back to the user.

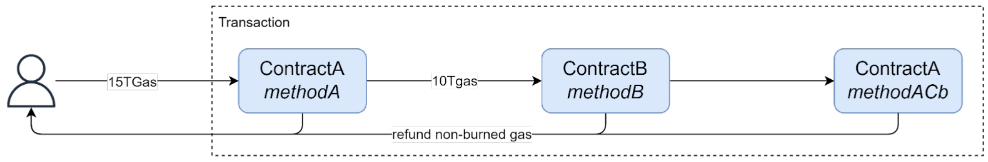


Figure 3.6: Unspent Refunded Gas Flow

- NEAR tokens can also be attached to cross contract calls, but they work differently from the gas. Attached deposit is taken directly from the predecessor account. It means even if a user hasn't attached any deposit, a contract still can attach tokens, which will be taken from its account. Also, since cross-contract call failure doesn't mean transaction failure, there are no automatic refunds. All refunds should be done explicitly in the rollback code.

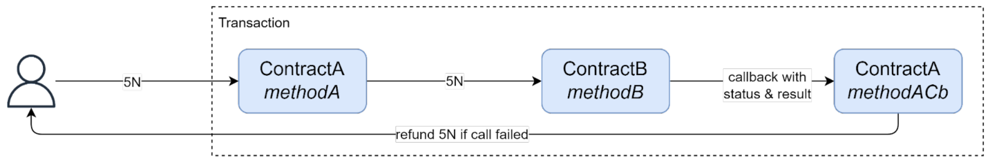


Figure 3.7: Rollback Code Refunds Flow

A few notes on failure modes - since smart contracts run on a decentralized environment, which means they are executed on multiple machines and there is no single point of failure, they won't fail because of environment issues (e.g. because a machine suddenly lost power or network connectivity). The only possible failures come from the code itself, so they can be predicted and proper failover code added.

In general, cross-contract call graphs can be quite complex (one contract may call multiple contracts and even perform some conditional calls selection). The only limiting factor is the amount of gas attached, and there is a hard cap defined by the network of how many gas transactions may have (this is necessary to prevent any kind of DoS attacks on the Network and keep contracts complexity within reasonable bounds).

3.6 Data Management

3.6.1 Data Structures

Delving deeper into NEAR's storage paradigm, I find that the foundational storage mechanism for NEAR smart contracts is a key-value pair system. While this serves basic applications, more complex applications necessitate advanced data structures. The **NEAR SDK** [50] equips developers with enhanced data structures like vectors, sets, and maps [51]. However, there are several considerations:

- Ultimately, they are stored as binary values, which means it takes some gas to serialize and deserialize them. Also, different operations cost different amounts of gas (complexity table [52]). Because of this, careful choice of data structures is very important. Moving to a different data structure later will not be easy and would probably require data migration.
- While very useful, vectors, maps and sets won't match the flexibility and power of classical relational databases. Even implementations of simple filtering and searching might be quite complex and require a lot of gas to execute, especially if multiple entities with relations between them are involved.
- They are limited to a single contract. If data from multiple contracts is required, aggregation should be performed using cross-contract calls or on a client side, which is quite expensive in terms of gas and time.

3.6.2 Relational Databases using Indexers

To support more complex data retrieval needs, data from smart contracts can be transitioned to relational databases using Indexers [53]. Essentially, an Indexer is a specialized blockchain node that processes incoming transactions and puts relevant data into a database. Collected data can be exposed to a client using a simple API server **(e.g. REST or GraphQL)**.

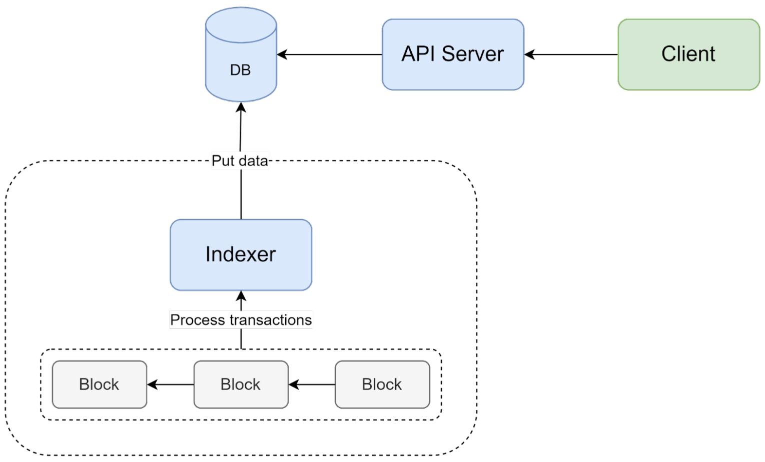


Figure 3.8: Indexer Pertinent Data Flow

To streamline Indexer creation, the NEAR Indexer Framework [54] was introduced. Besides, NEAR Lake Framework [55] is a lightweight alternative to NEAR Indexer Framework that is recommended for use when centralization can be tolerated.

3.6.3 NEAR Data Flow

Understanding the NEAR Protocol blockchain data flow is crucial for developers and users. At a glance, it may seem intricate, but it adheres to clear principles. The blockchain flow can be visualized as an infinite timeline that starts but doesn't end. On this timeline, blocks appear at regular intervals, each referencing the previous block, forming a chain.

Figure 3.9: Chain of Blocks

A unique aspect of the NEAR Protocol is its sharded architecture. There are multiple parallel networks, termed Shards, active simultaneously. Each Shard produces a portion of a block, referred to as a Chunk. When combined, these Chunks from all Shards constitute a Block in the NEAR Blockchain.

To further understand the data flow, imagine a series of tracks similar to those in audio/video editing applications. Each Shard has its own set of tracks, with the top track representing Chunks. These Chunks appear at regular intervals, regardless of any activity on the blockchain.

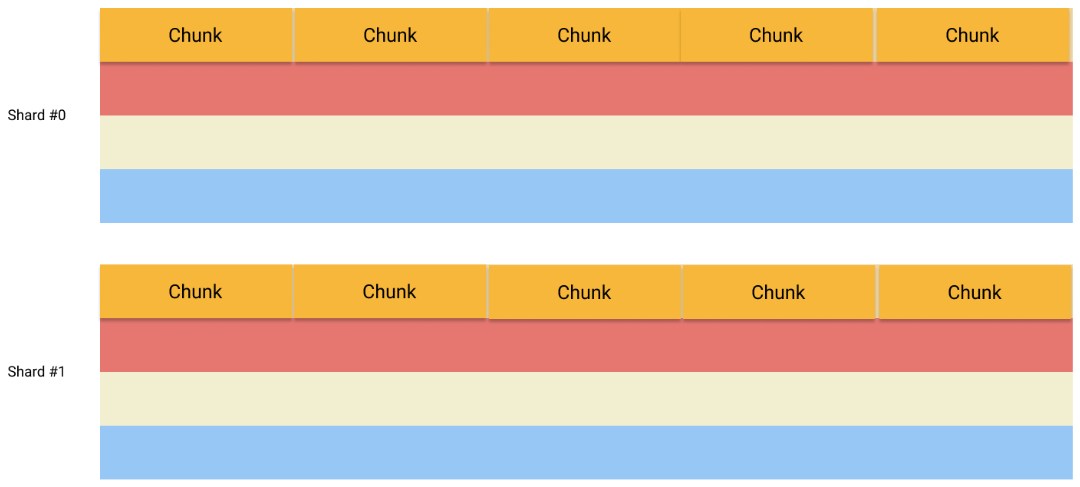


Figure 3.10: Visualization of Shards and Chunks

When activity occurs, it often involves sending a Transaction with change instructions. Once executed, the immediate outcome of a transaction isn't the final result but an internal execution request known as a Receipt. This Receipt can even traverse Shards, making it a powerful asset in the NEAR Protocol.

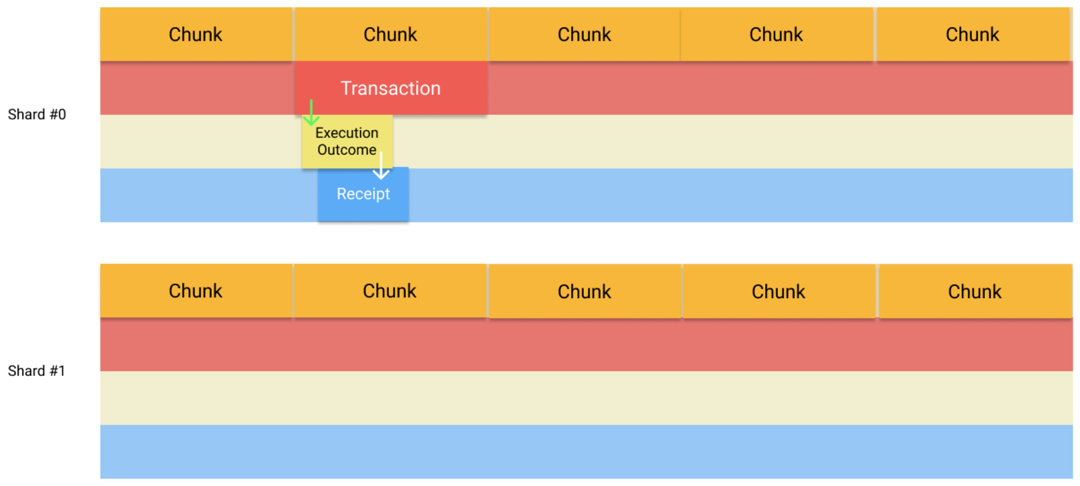


Figure 3.11: Data Flow of Transaction Execution

Let's consider a scenario: two accounts, **alice.near** and **bob.near**, residing on different Shards. If **alice.near** initiates a transaction to send tokens to **bob.near**, the transaction is swiftly executed, producing a Receipt. However, since **bob.near** is on a different Shard, the Receipt moves to **bob.near's** Shard for execution. This system ensures data integrity and flow consistency across Shards.

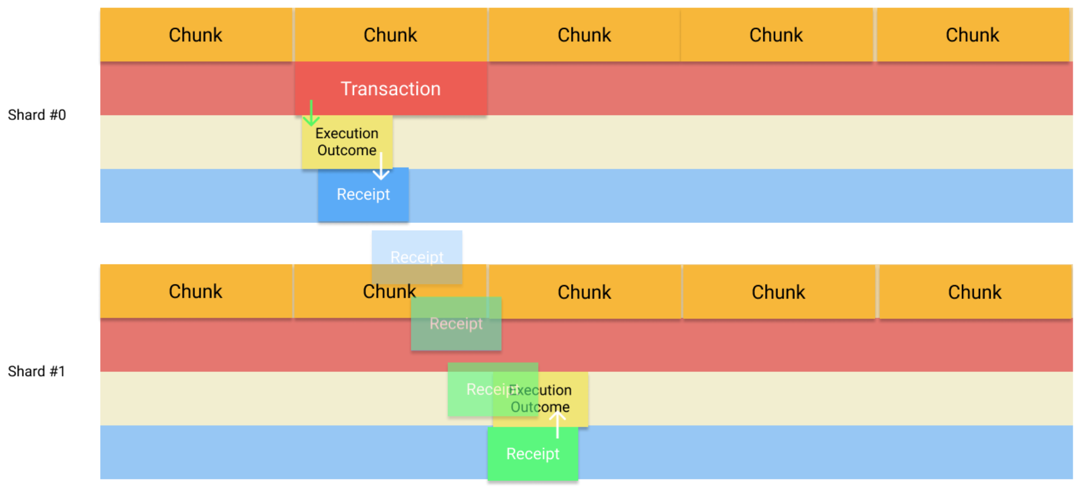


Figure 3.12: Cross-Shard Receipt Execution

It's essential to note that the process might involve more **Receipts** and **ExecutionOutcomes** for complex transactions, especially those with **cross-contract calls**. Additionally, every transaction may include a refund due to the gas mechanism in NEAR Protocol. For instance, if a user attaches more gas than required, they receive a refund.

3.6.4 Event Logging

Extracting transaction data remains challenging due to the unique data structure of each smart contract. To simplify this process, smart contracts can write structured information about outcome into the logs (e.g. in the JSON format). Each smart contract can use its own format for such logs, but the general format has been standardized as Events [56].

This setup bears a resemblance to **Event Sourcing** [20]: the blockchain archives events (transactions) which are then transferred to a relational database through an Indexer. However, challenges such as potential indexing delays, which could last seconds, must be factored into client designs.

For those seeking an alternative to crafting their own Indexer, The Graph [57] offers a solution. With NEAR support currently in beta [58], it operates on the **Indexer-as-a-Service** model and even boasts a decentralized indexing approach.

3.6.5 Introduction to Blockchain Indexers

The Role and Challenge of Indexers

Indexers are pivotal in simplifying the complexities of querying data across multiple blocks in blockchain systems. A blockchain's deterministic nature, structured around serialized writes of blocks, is tailored for *"narrow"* queries, which target specific blocks or accounts. However, aggregating data from multiple blocks, termed *"wide"* queries, becomes challenging due to the need to accumulate results from individual block queries.

Limitations of Smart Contracts

While the inherent nature of blockchains and smart contracts offers a decentralized mode of operations, it's essential to note that smart contracts aren't a direct substitute for backends. Their deterministic structure and lack of interaction with off-chain variables mean they can't access external APIs or conduct tasks common in conventional systems, such as user notifications or third-party integrations.

Practical Examples

To illustrate, consider a dApp selling e-books. The actual act of emailing the purchased e-book to a user can't be executed by a smart contract on the blockchain. A solution involves using an off-chain helper that detects a successful "buy an e-book" transaction and triggers the email dispatch. The NEAR blockchain, for instance, employs a **JSON-RPC endpoint** [39] allowing external entities to interact with it, providing both smart contract calls and data retrieval. This *"pull model"* has its drawbacks, including possible inefficiencies and unavailability of certain data like Local Receipts.

Advantages of Indexers

Enter indexers, which embody the *"push model"*. Instead of constantly querying or pulling data, an indexer passively receives data, facilitating quicker analysis and actions. They're particularly beneficial for *"wide"* queries. For instance, the NEAR Indexer for Explorer captures data streams and stores them in a database, enabling more efficient querying, even for tasks as intricate as recovering multiple accounts linked to a single seed phrase – a feat unattainable with only **JSON-RPC**.

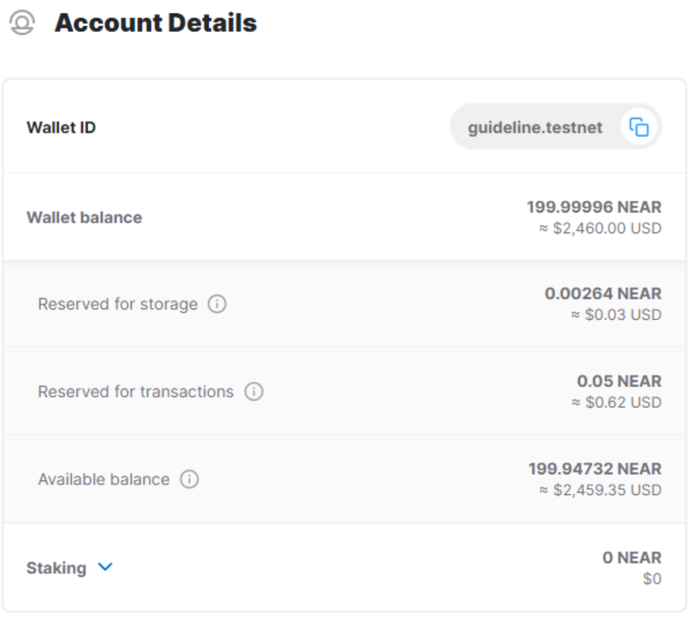
Indexers enhance the interoperability and efficiency of blockchain systems, especially for *"wide"* queries. While this overview provides foundational knowledge, delving into specific projects like the Lake Indexer and other third-party tools, such as The Graph, Pagoda, Pipespeak, and SubQuery, offers practical insights into building and utilizing indexers effectively.

3.6.6 Interaction with NEAR Network

Once we've chosen a network to use, we need a way to interact with it. Of course, transactions can be constructed manually and posted into node's API [42]. But this is tedious [59] and isn't fun at all. That's why, NEAR provides a CLI [60] which automates all of the necessary actions. It can be used locally for development purposes or on build machines for **CI/CD scenarios**.

3.6.7 Account Management

In order to manage accounts on the NEAR network, Wallet [61] can be used. It can show an effective account balance and active keys.



Account Details

Wallet ID	guideline.testnet
Wallet balance	199.99996 NEAR ≈ $2,460.00 USD
Reserved for storage	0.00264 NEAR ≈ $0.03 USD
Reserved for transactions	0.05 NEAR ≈ $0.62 USD
Available balance	199.94732 NEAR ≈ $2,459.35 USD
Staking	0 NEAR $0

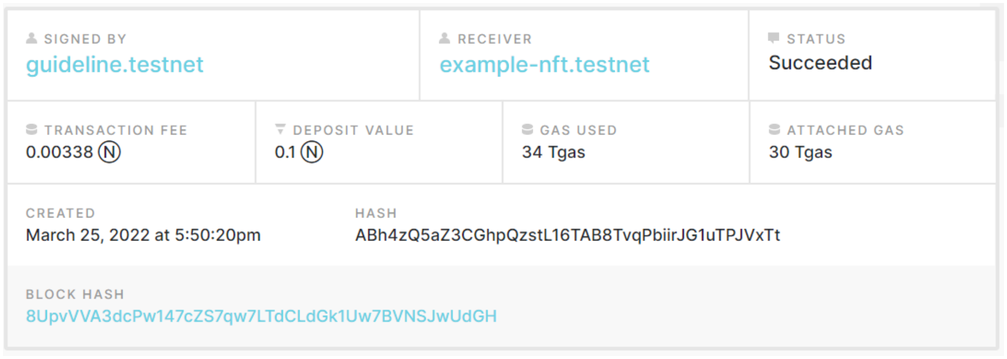
Figure 3.13: Account Details UI

On the image above, *"Reserved for storage"* are tokens locked by a smart contract to cover current storage requirements, and *"Reserved for transactions"* represents the amount of tokens locked to cover gas cost by Functional Call keys.

3.6.8 NEAR Explorer

Last, but not least, blockchain transactions can be viewed using NEAR Explorer [40]. It provides insights into transaction execution and outcome. Let's look at one example [62]. First of all, we can see general transaction information - sender, receiver, status. After this, we can see gas usage information:

- Attached Gas: Total gas provided for the transaction.
- Gas Used: Actual gas spend.
- Transaction Fee: Gas used multiplied to current gas price, shows an actual cost of a transaction in NEAR tokens. Also, Deposit Value shows the amount of NEAR tokens transferred from sender to receiver.



SIGNED BY	RECEIVER	STATUS
guideline.testnet	example-nft.testnet	Succeeded

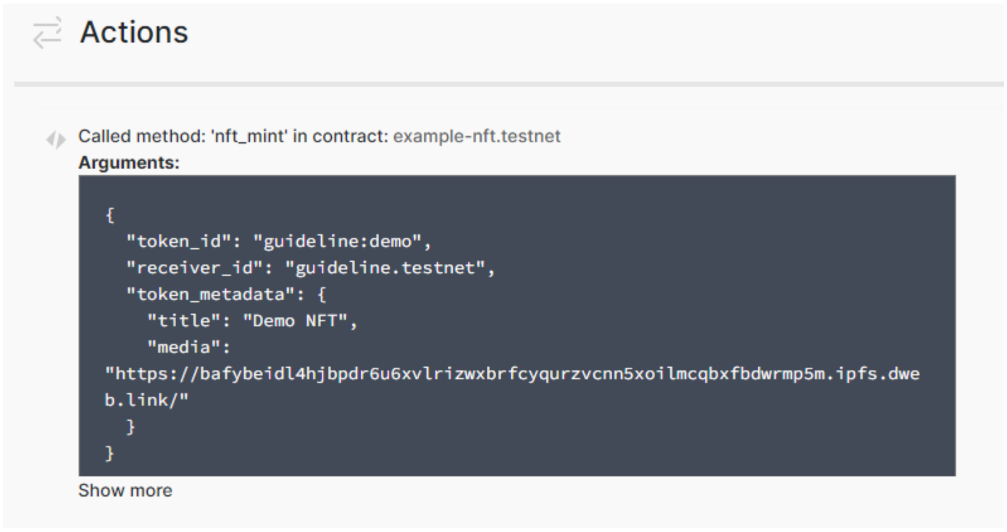
TRANSACTION FEE	DEPOSIT VALUE	GAS USED	ATTACHED GAS
0.00338 Ⓝ	0.1 Ⓝ	34 Tgas	30 Tgas

CREATED	HASH
March 25, 2022 at 5:50:20pm	ABh4zQ5aZ3CGhpQzstL16TAB8TvqPbiirJG1uTPJVxTt

BLOCK HASH
8UpvVVA3dcPw147cZS7qw7LTdCLdGk1Uw7BVNSJwUdGH

Figure 3.14: Transaction Overview

Below this, we can inspect transaction actions (recall, that transactions may have multiple actions). In this case, we have a single FunctionCall action with arguments:



Actions

Called method: 'nft_mint' in contract: example-nft.testnet
Arguments:

```
{
  "token_id": "guideline:demo",
  "receiver_id": "guideline.testnet",
  "token_metadata": {
    "title": "Demo NFT",
    "media":
"https://bafybeidl4hjbpdr6u6xvlrizwxbrfcyqurzvcnn5xoilmcqbxfbdwrmp5m.ipfs.dwe
b.link/"
  }
}
```

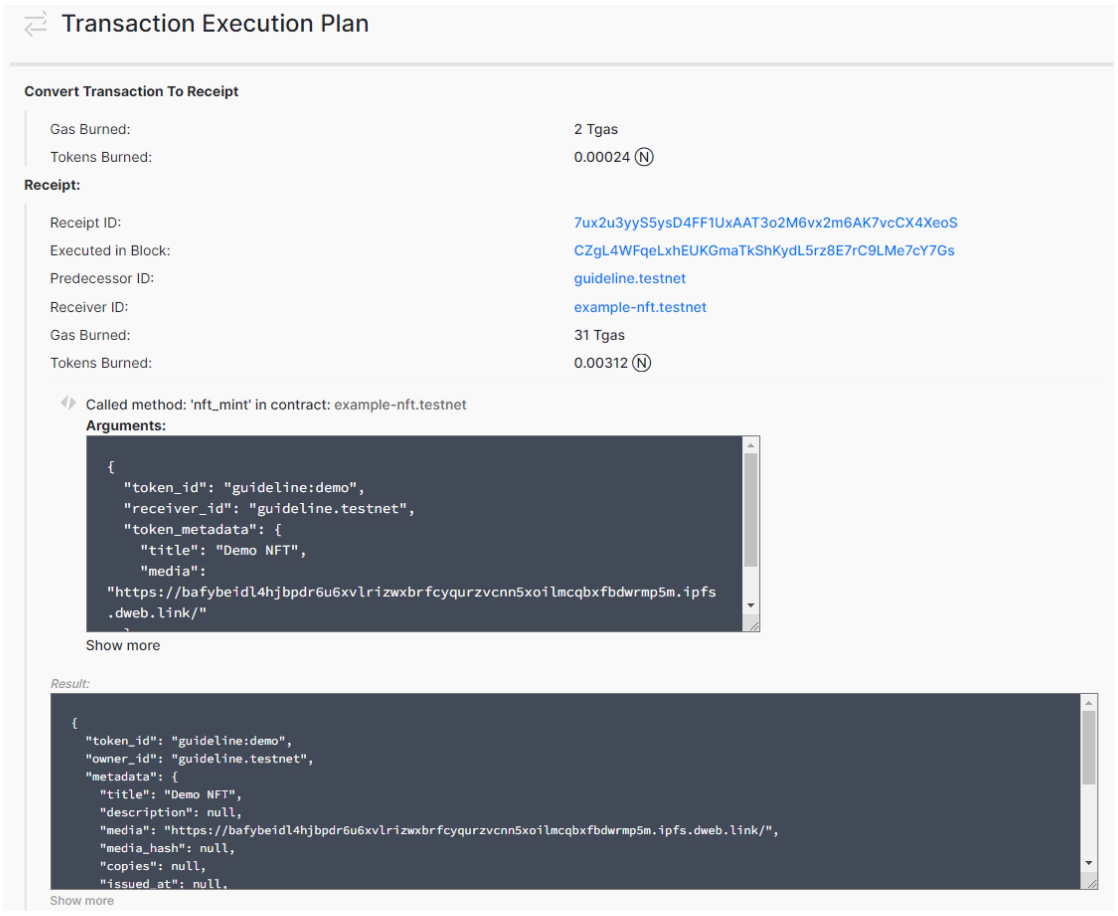
Show more

Figure 3.15: Transaction Actions

At the end, transaction execution details, including *token transfers, logs, cross-contract calls and gas refunds* are provided. One thing that we haven't covered yet is receipts [63]. For most practical purposes they are just a transaction implementation detail. They are quite useful in a transaction explorer to understand how a transaction was executed, but aren't really relevant outside of it.

A Receipt is the only actionable object in the system. Therefore, when we talk about *"processing a transaction"* on the NEAR platform, this eventually means *"applying receipts"* at some point. A good mental model is to think of a Receipt as a paid message to be executed at the destination (receiver). And a Transaction is an externally issued request to create the Receipt (there is a 1-to-1 relationship). There are several ways of creating Receipts:

- Issuing a Transaction
- Returning a promise (related to cross-contract calls)
- Issuing a Refund



Transaction Execution Plan

Convert Transaction To Receipt

Gas Burned:	2 Tgas
Tokens Burned:	0.00024 Ⓝ

Receipt:

Receipt ID:	7ux2u3yyS5ysD4FF1UxAAT3o2M6vx2m6AK7vcCX4XeoS
Executed in Block:	CZgL4WFqeLxhEUKGmaTkShKydL5rz8E7rC9LMe7cY7Gs
Predecessor ID:	guideline.testnet
Receiver ID:	example-nft.testnet
Gas Burned:	31 Tgas
Tokens Burned:	0.00312 Ⓝ

Called method: 'nft_mint' in contract: example-nft.testnet

Arguments:

```
{
  "token_id": "guideline:demo",
  "receiver_id": "guideline.testnet",
  "token_metadata": {
    "title": "Demo NFT",
    "media":
"https://bafybeidl4hjbpdr6u6xvlrizwxbrfcyqurzvcnn5xoilmcqbxfbdwrmp5m.ipfs
.dweb.link/"
```

Show more

Result:

```
{
  "token_id": "guideline:demo",
  "owner_id": "guideline.testnet",
  "metadata": {
    "title": "Demo NFT",
    "description": null,
    "media": "https://bafybeidl4hjbpdr6u6xvlrizwxbrfcyqurzvcnn5xoilmcqbxfbdwrmp5m.ipfs.dweb.link/",
    "media_hash": null,
    "copies": null,
    "issued_at": null,
```

Show more

Figure 3.16: Transaction Execution Plan

3.7 Contract Upgrades

During the development, and sometimes even in production, updates to a contract's code (or even data) are needed. That's why different contract upgrades mechanisms have been created.

3.7.1 Contract Upgrades in Local Development

During the local development, we can just recreate a smart contract's account each time we deploy a contract (**dev-deploy** [64] command in NEAR CLI exists for this). With such an approach, contract data will be purged each time a contract is redeployed [65].

3.7.2 Upgrades in Stable Environments: Code and State

However, once we move to a more stable environment, like testing or production, more sophisticated methods are needed. Redeployment of code is quite simple: we just issue another *DeployContract* transaction, and NEAR will handle the rest. The biggest challenge is to migrate contract state - several approaches are possible [66], but all of them involve some kind of migration code.

3.7.3 Programmatic Updates: Decentralizing the Upgrade Process

But it can take the upgrade strategy one step further. In the previous strategies, developers are fully in control of code upgrades. This is fine for many applications, but it requires some level of trust between users and developers, since malicious changes could be made at any moment and without the user's consent. To solve this, a contract update process itself can also be decentralized - this is called **Programmatic Updates** [67]. The exact strategy may vary, but the basic idea is that the contract update code is implemented in a smart contract itself, and a **Full Access** key to the contract account is removed from a blockchain **(via DeleteKey transaction)**. In this way, an update strategy is transparent to everyone and cannot be changed by developers at will.

Chapter 4

The Economics of Web 3.0: NFTs, FTs, DeFi and DEX

4.1 NEAR Tokens

The Web 3.0 era marks a significant shift not just in technology but also in economics, pivoting on three primary elements: Non-Fungible Tokens (NFTs), Fungible Tokens (FTs), Decentralized Finance (DeFi) and Decentralized Exchange (DEX).

NEAR Enhancement Proposals Each blockchain has its own set of standards, and NEAR Protocol ecosystem is defined by a set of standards known as **NEAR Enhancement Proposals** (NEP) [68], each outlining specific functionalities and guidelines for the network. These standards range from token standards to storage management and more. Below is a brief overview of some of these standards:

NEP-0001 This is the foundational NEP, establishing the purpose and guidelines for all subsequent NEPs. It sets the framework for how NEPs are proposed, reviewed, and implemented.

NEP-0021 A standard interface for fungible tokens allowing for ownership, escrow and transfer, specifically targeting third-party marketplace integration.

NEP-0141 A standard interface for fungible tokens that allows for a normal transfer as well as a transfer and method call in a single transaction. The storage standard addresses the needs (and security) of storage staking. The fungible token metadata standard provides the fields needed for ergonomics across dApps and marketplaces.

NEP-0145 NEAR uses storage staking which means that a contract account must have sufficient balance to cover all storage added over time. This standard provides a uniform way to pass storage costs onto users.

NEP-0148 An interface for a fungible token's metadata. The goal is to keep the metadata future-proof as well as lightweight. This will be important to dApps needing additional information about an FT's properties, and broadly compatible with other tokens standards such that the NEAR Rainbow Bridge can move tokens between chains.

NEP-171 Known as the *"Non-Fungible Token Standard"*, establishes a standard interface for NFTs on NEAR. It introduces functionalities like *nft_transfer*, *nft_transfer_call*, and *nft_token*, providing a comprehensive framework for NFT interactions. This standard also includes extensions for Approval Management and Metadata, enhancing the flexibility and utility of NFTs within the NEAR ecosystem.

NEP-177 Titled *"Non-Fungible Token Metadata"*, focuses on providing a lightweight and future-proof metadata structure for NFTs on NEAR. This standard is crucial for dApps that require detailed information about an NFT's properties, ensuring broad compatibility with various token standards. The metadata standard addresses both the contract level and the individual token level, detailing attributes like title, description, media, and more.

NEP-0178 A system for allowing a set of users or contracts to transfer specific Non-Fungible Tokens on behalf of an owner. Similar to approval management systems in standards like ERC-721.

NEP-0181 Standard interfaces for counting & fetching tokens, for an entire NFT contract or for a given owner.

NEP-0199 Both focus on Non-Fungible Token Royalties and Payouts, providing standards for handling royalties and payouts associated with NFT transactions.

NEP-0245 A standard interface for a multi token standard that supports fungible, semi-fungible, non-fungible, and tokens of any type, allowing for ownership, transfer, and batch transfer of tokens regardless of specific type.

NEP-0264 This proposal is to introduce a new host function on the NEAR runtime that allows for scheduling cross-contract function calls using a ipercentage/weight of the remaining gas in addition to the statically defined amount. This will enable async promise execution to use the remaining gas more efficiently by utilizing unspent gas from the current transaction.

NEP-0297 Events format is a standard interface for tracking contract activity.

NEP-0330 The contract source metadata is a standard interface that allows auditing and viewing source code for a deployed smart contract. Implementation of this standard is purely optional but is recommended for developers whose contracts are open source.

NEP-0364 This NEP introduces the request of adding into the NEAR runtime a pre-compiled function used to verify signatures that can help IBC compatible light clients run on-chain.

NEP-0366 In-protocol meta transactions allow third-party accounts to initiate and pay transaction fees on behalf of the account.

NEP-0393 **Soulbound Token (SBT)** is a form of a non-fungible token which represents an aspect of an account: soul. Transferability is limited only to a case of recoverability or a soul transfer. The latter must coordinate with a registry to transfer all SBTs from one account to another, and banning the source account. SBTs are well suited for carrying proof-of-attendance, proof-of-unique-human *"stamps"* and other similar credibility-carriers.

NEP-0399 This NEP proposes the idea of Flat Storage, which stores a flattened map of *key/value pairs* of the current blockchain state on disk. Note that original Trie (persistent merkelized trie) is not removed, but Flat Storage allows to make storage reads faster, make storage fees more predictable and potentially decrease them.

NEP-0413 A standardized Wallet API method, namely *signMessage*, that allows users to sign a message for a specific recipient using their NEAR account.

NEP-0418 This proposal is to switch the behavior of the *attached_deposit* host function on the runtime from panicking in view contexts to returning 0. This results in a better devX because instead of having to configure an assertion that there was no attached deposit to a function call only for transactions and not view calls, which is impossible because you can send a transaction to any method, you could just apply this assertion without the runtime aborting in view contexts.

NEP-0448 A major blocker to a good new user onboarding experience is that users have to acquire NEAR tokens to pay for their account. With the implementation of NEP-366, users don't necessarily have to first acquire NEAR tokens in order to pay transaction fees, but they still have to pay for the storage of their account. To address this problem, NEAR proposes allowing each account to have free storage for the account itself and up to four keys and account for the cost of storage in the gas cost of create account transaction.

NEP-0455 Introduce compute costs decoupled from gas costs for individual parameters to safely limit the compute time it takes to process the chunk while avoiding adding breaking changes for contracts.

NEP-0492 This proposal aims to restrict the creation of top level accounts (other than implicit accounts) on NEAR to both prevent loss of funds due to careless user behaviors and scams and create possibilities for future interopability solutions.

NEP-0514 This proposal aims to adjust the number of block producer seats on *testnet* in order to ensure a positive number of chunk-only producers present in *testnet* at all times.

Each NEP plays a critical role in shaping the functionalities, standards, and governance of the NEAR Protocol, contributing to the overall robustness and versatility of the ecosystem. NEP-177 and NEP-171 are two significant standards and I applied these two on our dApp metadata structure.

4.1.1 Non-Fungible Tokens (NFTs)

At the heart of the new Web 3 economy lies Non-Fungible token (NFT) [69]. In a nutshell, it's a way to represent digital ownership in a decentralized way. From a technical perspective, it's just a piece of data on a blockchain. The simplest case of such data is just a *(token_id, account_id)* tuple, where *token_id* uniquely identifies an asset, and *account_id* identifies an owner. A smart contract that owns this data defines a set of allowed operations - like creation of a new token **(minting)** or transfer of a token to another account. An exact set of allowed operations is defined in an NFT standard.

Because NFTs are tied to a specific contract, they mostly make sense only in scope of this contract, and subsequently they are tied to a specific dApp. It's possible to implement transfer of NFTs between contracts, but there's no standard way to do this.

What digital asset is hiding behind a *token_id* is up to the smart contract to decide. There are few common ways how to handle this:

- Store an asset itself in a smart contract alongside the ownership information. This is the most straightforward way, but often is not feasible since storage cost is quite high and many types of digital assets, especially media, are quite big.

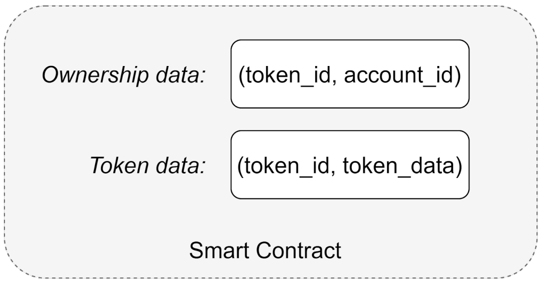


Figure 4.1: Direct Storage Mechanism for NFTs

- Store token data off-chain. Such an approach solves storage cost problems, but requires some level of trust to guarantee that data in the off-chain storage won't be changed or removed.

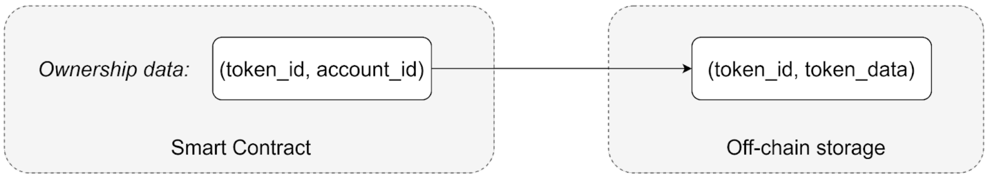


Figure 4.2: Off-Chain Storage for NFTs

- Store asset's metadata and hash on chain, and an asset itself on some off-chain storage. Storing an asset's hash on a chain guarantees data integrity and immutability. On-chain metadata usually includes basic token information, like title, description and media url. It's required to quickly identify an asset without downloading it from the storage. This is the most popular approach to handle NFT's since it combines the best of 2 previous approaches - token is immutable and storage cost is cheap (exact cost depends on the storage solution, but it usually several orders of magnitude cheaper than an on-chain storage)

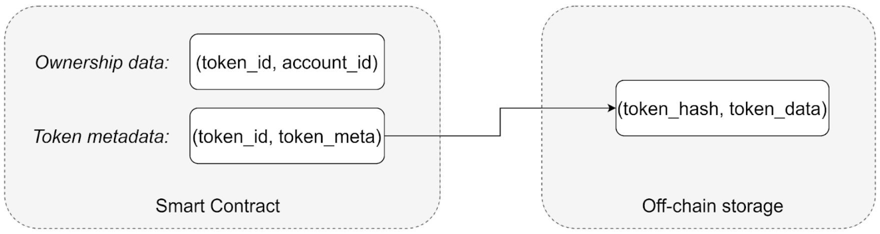


Figure 4.3: On-Chain Metadata and Hash Storage for NFTs

Choosing the right off-chain storage also can be a challenge, in general they can be divided into 2 buckets:

- Centralized storages - traditional Web 2 storage solutions, like relational databases or blob storages. While suitable for some applications, this means NFTs can be destroyed if a central server goes offline, so they aren't the most popular in the Web 3 world.
- Decentralized storages. As we already mentioned, BitTorrent protocol is one of the first examples of such decentralized storage solutions, but in recent years more advanced solutions have appeared - like IPFS, FileCoin and Arweawe. Such solutions are a preferred method to store digital assets, since they are cheap and decentralized, so no-one can destroy or alter NFT assets.

In addition to the NFT standard, NEAR also provides its implementation [70], which can be used by Smart Contract developers to implement NFTs in their smart contract. Implementation itself doesn't dictate assets storage model, so it's up to a developer to decide how and where it will be stored.

4.1.2 Fungible Tokens (FTs)

NFTs changed digital assets ownership model, but by itself they are not enough to build a full digital economy. In the simplest model, NFTs can be sold and bought using main blockchain currency (e.g. NEAR tokens), but this is quite limiting since circulation and price of such tokens is dictated by the blockchain itself. What if, instead of relying on blockchain currency, applications could create their own? For exactly this reason, Fungible Tokens (FT) have been created.

Similarly to NFTs, fungible tokens are also just a piece of data stored in a smart contract, but instead of storing unique token ids, an amount of tokens held by an account is stored.

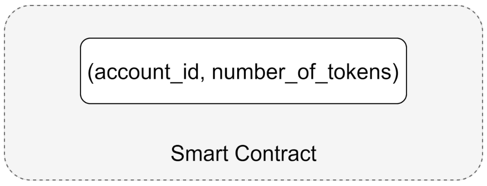


Figure 4.4: Structure of Fungible Tokens in Smart Contracts

Smart Contracts can define allowed operations - like transfer or payment using this token. NEAR defines a standard [71] for fungible tokens and provides a default implementation [70].

Since an application is fully in control over emission and circulation of such tokens, a full fledged application economy can be created. For example, users can earn FTs for performing actions, and spend them to buy or mint new NFTs. Another exciting option is creation of Decentralized Autonomous Organizations (DAOs) [72], in which FTs can be used as a membership (or governance) tool. In such scenarios, tokens are awarded to members and can be used to vote on decisions or participate in community events.

4.1.3 Decentralized Finance (DeFi) & Decentralized Exchanges (DEX)

DeFi [73] stands as a transformative element in Web 3.0, offering an alternative to traditional financial systems. DeFi's core concept revolves around creating financial services, including borrowing, lending, and trading, in a decentralized setup. This is achieved through various platforms and protocols, with Decentralized Exchanges (DEX) [74] being a critical component for trading fungible tokens for other cryptocurrencies or stablecoins. DeFi's integration with both NFTs and FTs demonstrates the extensive economic potential of Web 3.0, blending finance with the new digital asset paradigm. We won't go into details here, but at the core a liquidity pool [75] for a Fungible Token can be created on DEX, which allows trades of this token for other tokens or stablecoins [76]. This opens the door for a new gaming model - Play-to-Earn [77], where players can earn real-life money just by playing a game.

4.2 Integrating NFTs in Web 2 Applications

The transformation into Web 3 begins with decentralizing digital assets ownership through Non-Fungible Tokens (NFTs). This step paves the way for a user-owned economy, where digital assets exchange hands without developer interference. NFTs in Web 3 represent any digital asset, like game characters or skins, existing on the blockchain but require integration with traditional server-based applications.

4.2.1 Blockchain-Enabled Application Architecture

First of all, let's outline a typical architecture of a Web 2 application. In most cases, a classic client-server model is used:

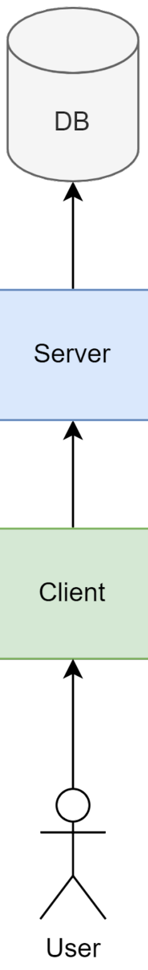


Figure 4.5: Traditional Web 2 Application Architecture

In such architecture, we usually have 3 layers:

- Database - stores application's data. This can be a single database, or several databases of different types, but this is mostly an implementation detail - for our purposes we can view it as a single logical database.
- Server - a centralized web-server. It may be implemented using different architecture patterns (monolithic, microservices, serverless) and technologies, but again, we can consider it as a single logical server.

- Client - client side application user directly interacts with. Different client types are possible: web, mobile or desktop. There is a difference between these clients in regards to blockchain integration, which we'll discuss later.

Now, let's compare it to a dApp architecture:

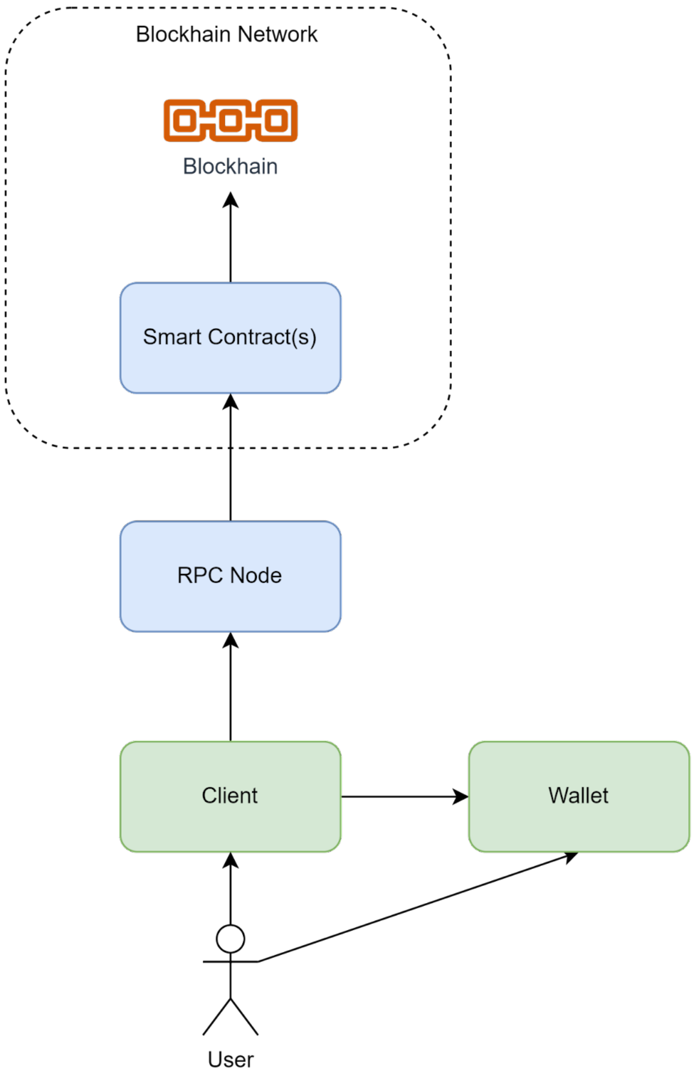


Figure 4.6: dApp Architecture with Blockchain Integration

We can notice that there is a common component in these architectures - the client application. This means we can use it as a junction point to connect them together.

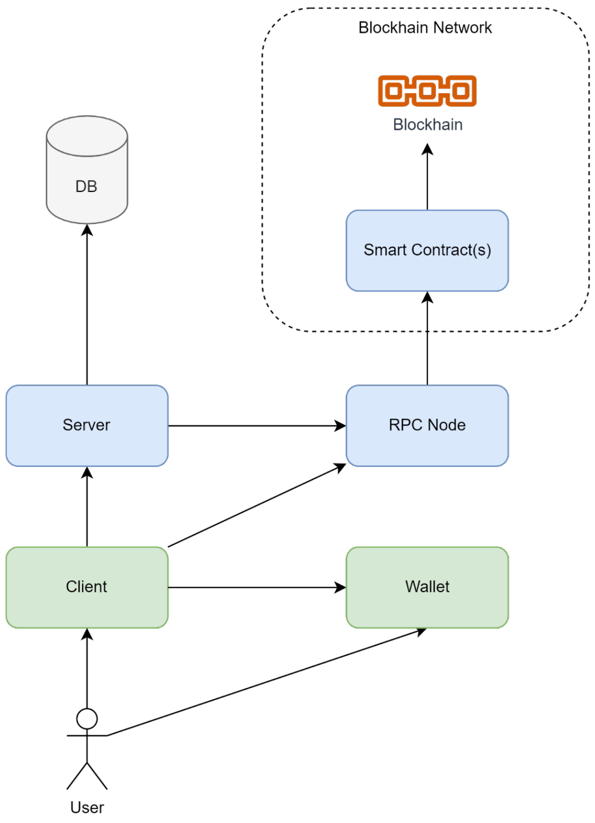


Figure 4.7: Client as a Junction Point in Hybrid Architectures

A keen reader may notice an additional connection between the Server and RPC Node. This is required because in a client-server architecture clients cannot be trusted. That's why every action performed from the client should be validated by a backend server. But in this case everything is complicated by the fact that they essentially have two backends: Web 2 server and a smart contract, so two possible validation flows are possible:

- Client performs an action on a server, which involves blockchain data. In this case the server should talk to the blockchain and verify that valid data is provided.

- Client performs an action on a smart contract, which involves server-owned data. Since the smart contract can't talk to the server directly to verify it, we should use a different way to verify the authenticity of the data. In blockchain terminology, such a server is called an **Oracle** [78]. We'll explore how to implement both of these approaches later.

By now, we've reached the point where the type of the client begins to matter. Specifically, problems arise from the dApps payment model - user's pay for the blockchain infrastructure using gas, so money goes directly to infrastructure providers. Also, users make payments directly on the blockchain, without using any intermediaries, like banks or payment services. This approach is at odds with mobile app stores **(Google Play Store and Apple App Store)** - they don't allow any payments on their respective mobile platforms without their cut. Although some shifts in policy are starting to happen (e.g. Apple vs Epic Games duel [79]), at the time of this writing getting blockchain-enabled applications into the stores will probably get vetoed by reviewers. There are some ways to bypass these limitations, e.g. by not using Play Store on Android, but all of these ways are either sub-par in terms of usability or involve some risk of getting banned by stores. That's why for mobile applications an alternative approach is needed.

Sometimes, to move forward we need to take a step back. In this case, to solve a problem with mobile clients NEAR can return to its initial concept of having two clients - one for blockchain integration, and another one for Web 2 server. Blockchain client can be a usual web application, which isn't subject to any constraints from stores. It can also serve as a connection point between blockchain and their existing application.

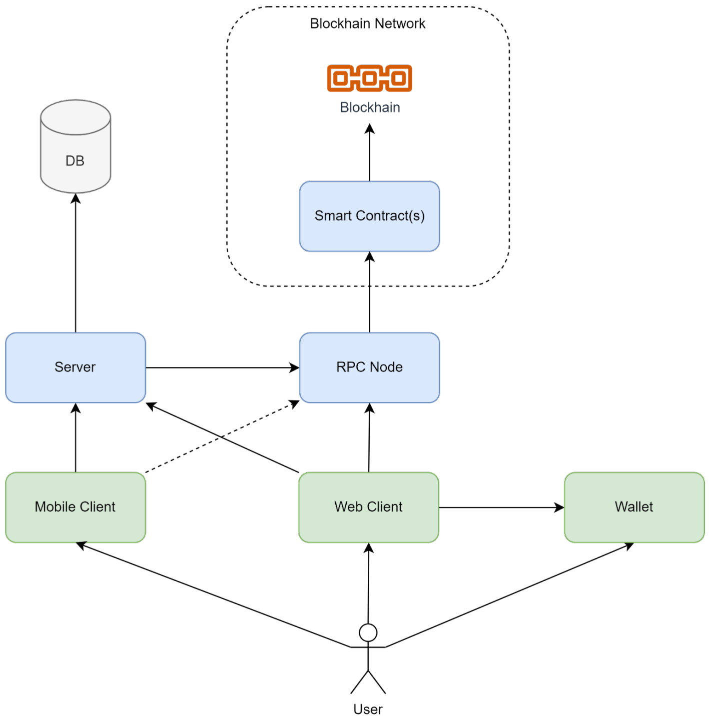


Figure 4.8: Dual-Client Architecture for Mobile App Integration

In this architecture the mobile client is still allowed to talk to the blockchain, but only in a read-only way, which doesn't require wallet connection or any payments. All actions on the blockchain happen on the Web Client instead. Further in this guide we'll use such dual-client architecture, since simpler architecture with a single client can be directly derived from it by merging two clients together.

At this point, NEAR architecture covers almost everything we need to start building our application. However, since we want to build a user-owned economy, we need a marketplace where it'll happen. An obvious choice is to put this marketplace into the web client, but there's one gotcha. If we recall the smart contract's storage model, it's not suitable to serve complex data queries, so an indexer should be used to aggregate data from blockchain into a proper database.

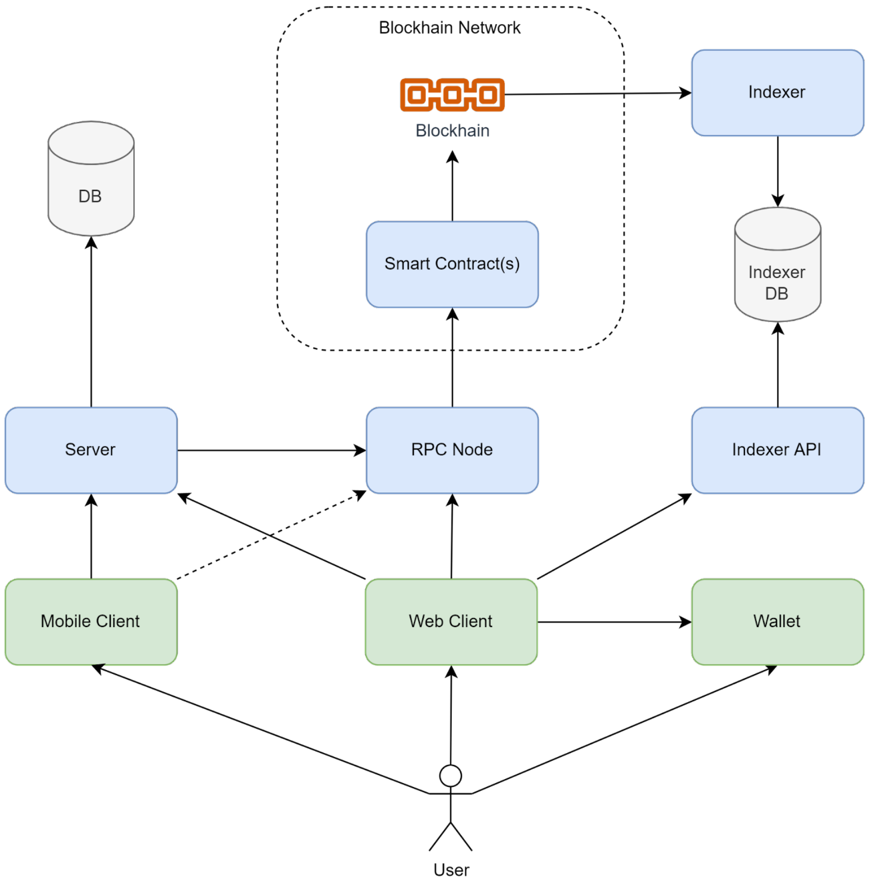


Figure 4.9: Using Indexers for Enhanced Data Querying

By now, every building-block is in place and we can start exploring how to implement this architecture in practice.

4.2.2 NFTs in Web 2 Applications

In order to implement a fully functional application using a hybrid Web 2 - Web 3 architecture, a lot of technological challenges have to be addressed, like authentication and authorization, seamless NFTs usage in client and server, and proper NFT storage model. In the following sections we'll take a closer look at this and describe common patterns and approaches.

Authentication and Authorization

Since our digital assets are represented as NFTs on blockchain, in order to use them in our Web 2 application, a server needs a way to authorize their usage. The basic idea is pretty simple - it can just read data from blockchain by calling a smart contract method and check an owner's account id. For such flow, we have 3 actors:

- Client that wants to use some digital asset (NFT).
- Smart Contract for NFTs should be implemented according to NEAR NFT standards **4.1**.
- Server that verifies ownership of NFT and uses it in its internal logic.

A general flow looks like this:

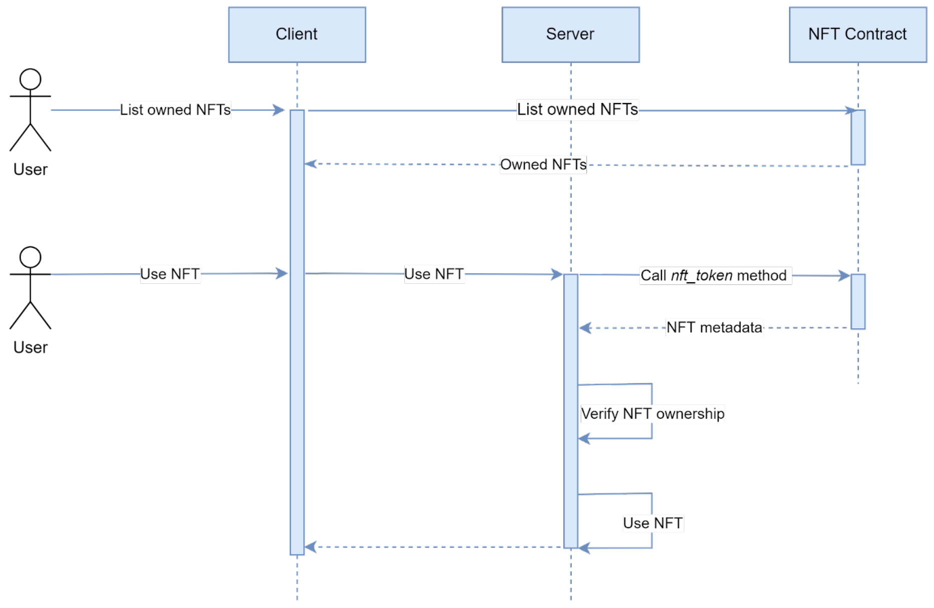
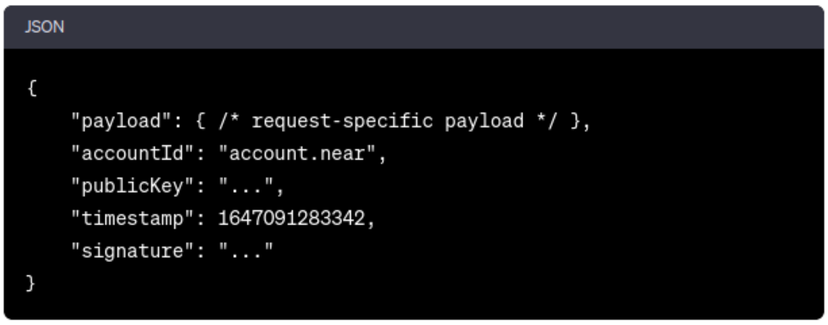


Figure 4.10: Basic NFT Authorization Flow

However, such an authorization process cannot be performed without authentication, so the server also needs a way to authenticate a user.

Recall that the user's identity on a blockchain is represented by a key pair. However, since in NEAR a user may have multiple key pairs and an account is a separate entity, the authentication procedure is a bit more complicated.

To authenticate our requests, we can use public-key cryptography - a client can sign a request using a user's private key, and then a server can verify the signature and key ownership. A typical request with authentication may look like this:



JSON

```
{
    "payload": { /* request-specific payload */ },
    "accountId": "account.near",
    "publicKey": "...",
    "timestamp": 1647091283342,
    "signature": "..."
}
```

Figure 4.11: Typical request with authentication

where:

- *accountId* – user's account id on NEAR.
- *publicKey* - public key of the key pair used for signature, must be either Functional or Full access key for the provided account.
- *timestamp* - current datetime, must be verified on server. It's needed to prevent replay attacks [80]. Alternative to timestamps is usage of nonce [81], but it's more complicated.
- *signature* - signature of the request payload and other fields. Usually, a payload is hashed beforehand.

Depending on the implementation, request body, headers, or other side channels can be used to transfer authentication data - exact implementation depends on used technologies and protocols.

Server can use this data to authenticate a request using the following approach:

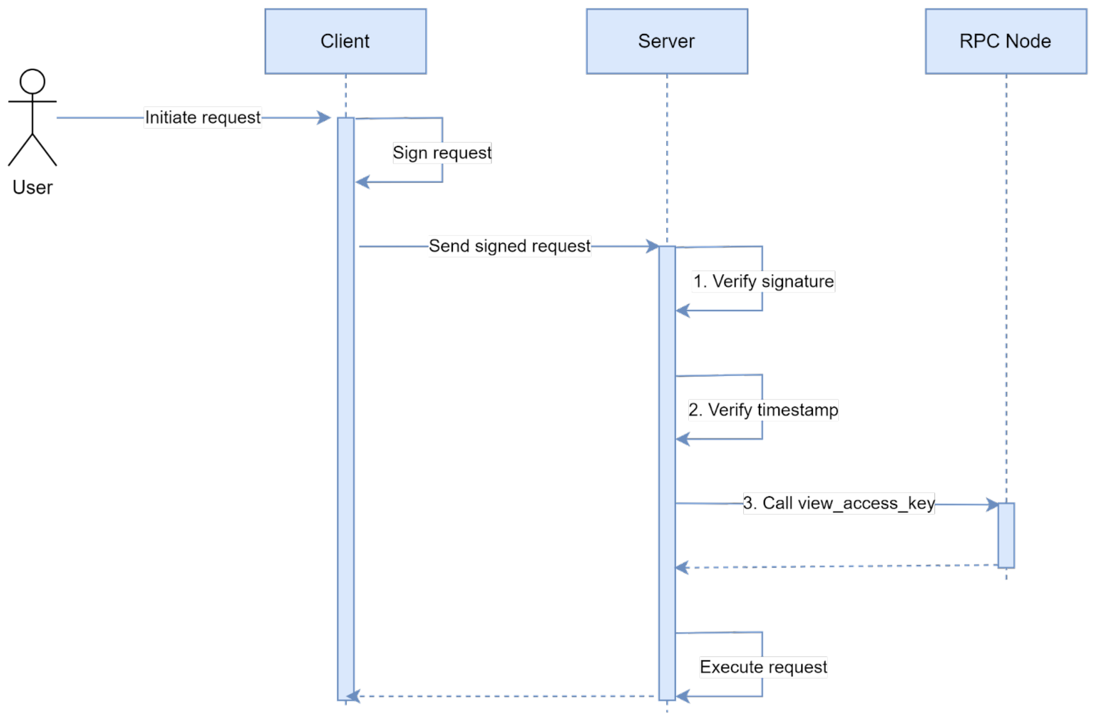


Figure 4.12: Authentication Data Flow in Web 2 Applications

3 authentication steps are performed on the server:

1. Signature verification - if the signature is correct, we are sure that the client really has the private key for the provided public key. Also, this proves that request data hasn't been modified in transit.

2. Timestamp verification - prevents replay attacks. Server can verify that the request's timestamp is not too old (e.g. has been created no more than 10 seconds ago).

3. Public key ownership verification - by calling *view_access_key* method [82], we can make sure that the provided public key is really associated with the provided account.

Such authentication approach is the simplest one, but has a few major drawbacks:

- Performing a **REST API** call to **RPC Node** is quite expensive to do each time from the performance perspective.

- It can't sign requests from the mobile client, since it usually should be disconnected from the blockchain due to store policies, and hence doesn't have a key pair.

- A NEAR account is required in order to start using the application, which complicates the onboarding process.

To solve the first problem, NEAR can simply issue a **JSON Web Token (JWT)** token or authenticate connection in some other way after a successful NEAR account authentication, so the it will serve as *"login"* of sorts:

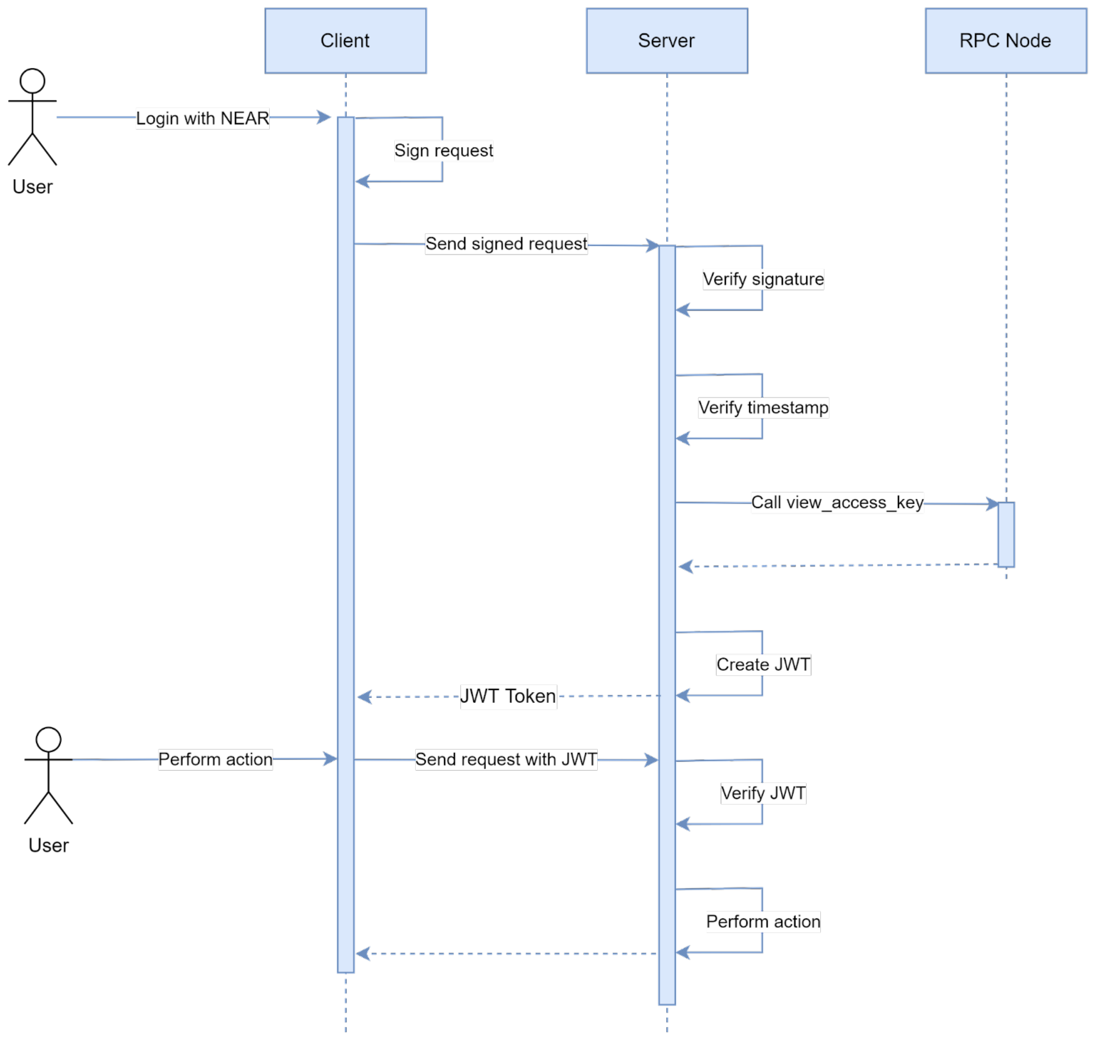


Figure 4.13: Hybrid Authentication Model

While this may be enough for some applications, it doesn't address the last 2 problems. In order to solve all of them, NEAR uses a hybrid authentication approach with 2 accounts:

1. *"Classic"* Web 2 account - all clients can use this account to call a server. For example, this can be a simple *username/password* or **OAuth** 2 login with a **JWT** token.

2. NEAR account - can be used from non-mobile clients only. Instead of performing NEAR account auth each time it needs to use it, it can do it a single time in order to *"connect"* this account to a primary Web 2 account and store Classic-NEAR account connection in a server database. In this way NEAR solve all problems - server doesn't need to authenticate NEAR account each time it wants to perform an authorization, instead it can read an associated NEAR account from its own database.

With such hybrid approach, different authentication methods are used for blockchain and server:

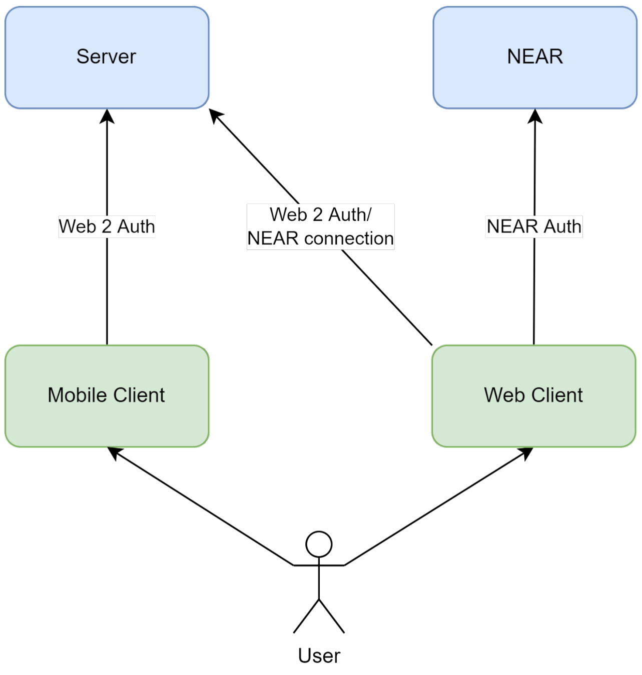


Figure 4.14: NEAR Account Connection Process

NEAR account connection sequence can be implemented in a very similar way to the already described NEAR authentication method, where at the end is stores an authenticated account in a database:

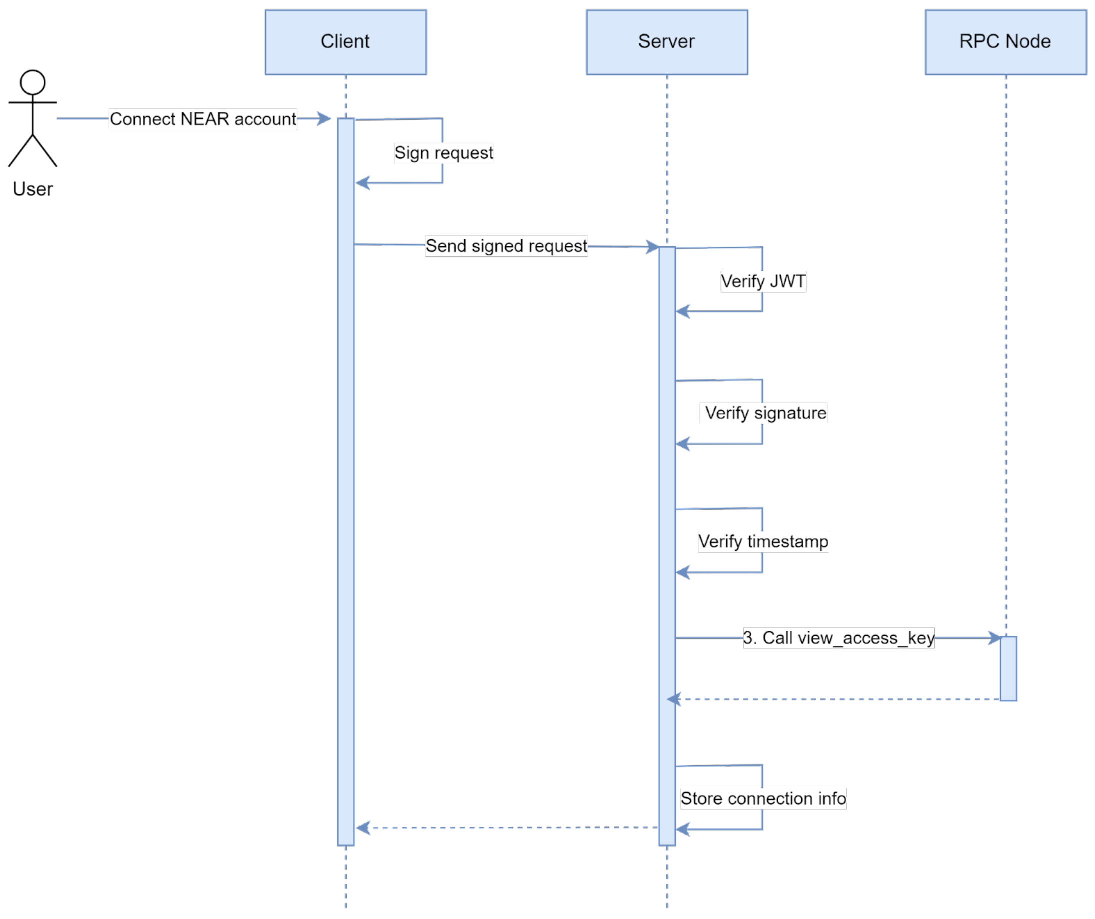


Figure 4.15: Simplified Login Flow with 'Login with Wallet' Feature

There's one more improvement it can make to this flow. Since a Web Client uses both accounts, a user is forced to login using both Web 2 login method (e.g. *login/password*) and NEAR Wallet. This is not ideal from the UX perspective, so NEAR can simplify it by introducing a *"Login with Wallet"* method to a server, which would work when a user already has a wallet connected. It can do this in a similar way to the account connection flow:

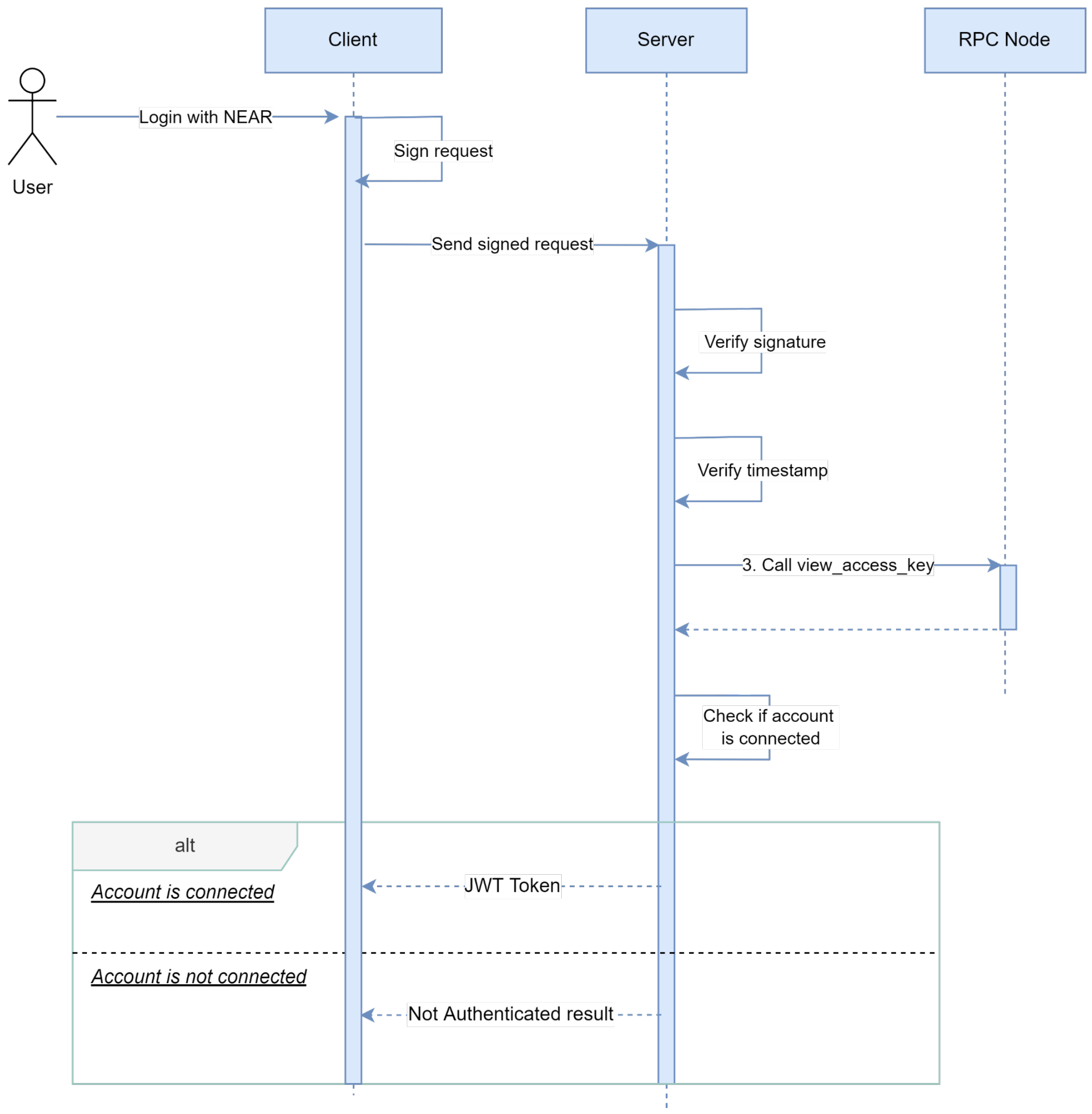


Figure 4.16: Smart Contract Authorization Check

Now, as we've discussed possible approaches for authentication, let's summarize it as an overall login flow for our clients:

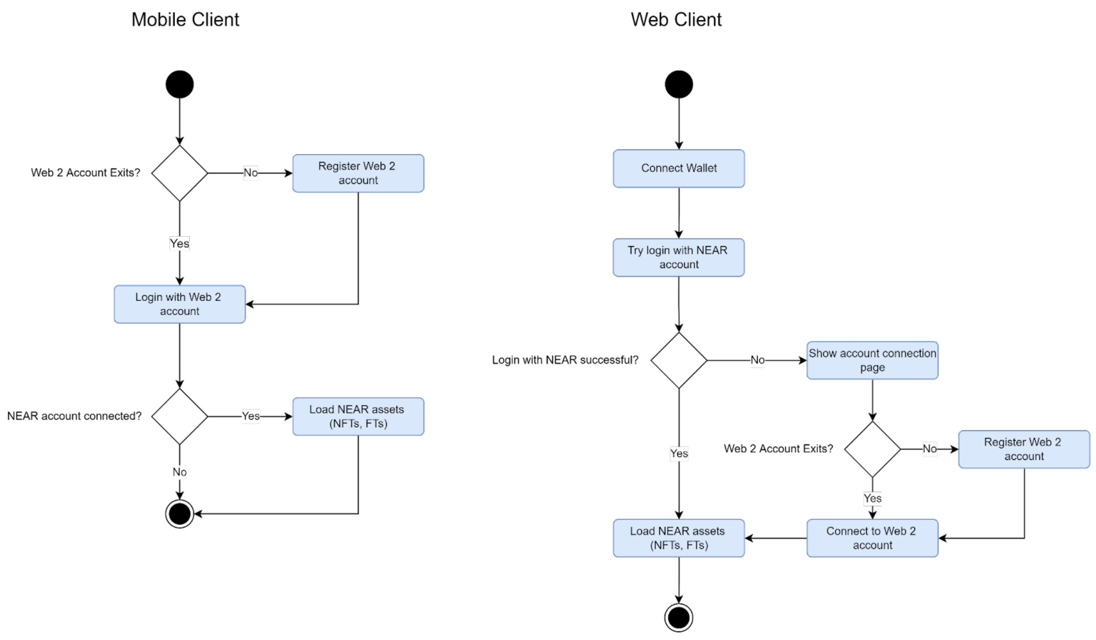


Figure 4.17: Web 2 and Web 3 Hybrid Application Flow

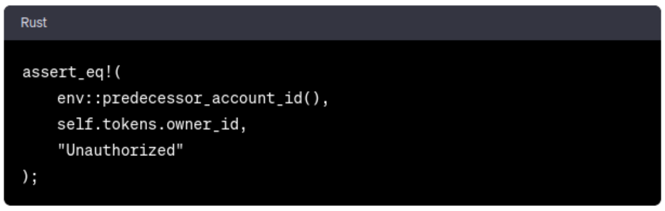
Of course, this is just one possible flow, and a different solution can be assembled from described building blocks. The most important considerations for choosing right authentication flow are following:

- Type of your client - for *web/desktop* clients, or sideloaded Android clients, it's possible to use Wallet as a single authentication method. For mobile clients installed from a store, a hybrid approach with multiple auth methods should be used.
- Target audience - if you target regular users that are not familiar with blockchain technologies, having a hybrid auth method to simplify onboarding might be better than forcing users to learn blockchain before trying your application.

Blockchain Auth & Auth So far, we've discussed authentication and authorization on the Web 2 server's side. But what about Web 3 smart contracts? Everything is much more straightforward in this case.

Since everything is public data on the blockchain, we don't need any authentication for read calls. For transactions, each is signed by an account's private key, and authentication is performed by the network. More details on transaction signing can be found in this documentation [36].

Authorization, on the other hand, must be performed on a smart contract itself, the simplest way is just to check whether caller is allowed to perform an action:



```
assert_eq!(
    env::predecessor_account_id(),
    self.tokens.owner_id,
    "Unauthorized"
);
```

Figure 4.18: Rust predecessor example

NFT usage

After we've learned how to authenticate users and how to authorize NFTs usage, let's find out how we can actually use them in our application.

Since they essentially have two backends in their application - server and smart contract(s), they both can use NFTs for different purposes:

- Server usually uses NFTs for actual functional purposes, e.g. by treating NFT as an in-game character, it can read its properties and stats and apply them using some logic.
- Smart contract is responsible for NFTs ownership, as well as NFTs creation, modification and burning (destruction).

This is the point where the NFT data storage model comes into place. Let's recall, that there are 3 possible options:

1. Store data in a smart-contract **(on-chain)**.
2. Store data in an off-chain decentralized storage [83], like IPFS **(off-chain)**.
3. Store data in an application itself **(in-application)**.

First 2 approaches provide good decentralization, but make NFT harder to work with, especially if we need to modify its properties. Let's consider usage options depending on a storage model used:

1. **On-chain storage**:
    - Server can read data from the blockchain by making an API call. Server can't directly modify data, it should make a smart contract call instead (by issuing a transaction).
    - Smart contract can directly *read/modify* NFT data.
    - Clients can read all data directly from the blockchain.

2. **Off-chain storage**:
    - Server can read data from storage by making an API call. Data on the off-chain storage is usually immutable, so no modifications are possible.
    - Smart contract cannot read data directly, an Oracle should be used. Data cannot be modified from it.
    - Clients should read data from both blockchain and off-chain storage.
3. **In-application storage**:
    - Server can *read/modify* data from its own database.
    - Smart contract cannot read data directly, an Oracle should be used. Data cannot be modified from it.
    - Clients should read data from both blockchain and server.

Depending on a particular use case, any approach, or combination of them, can be used. The simplest case is when we don't have any dynamic NFT data, and we can easily divide data by domains:

- Data that is used by smart contracts is stored on-chain.
- Other data is stored either off-chain or in-application.

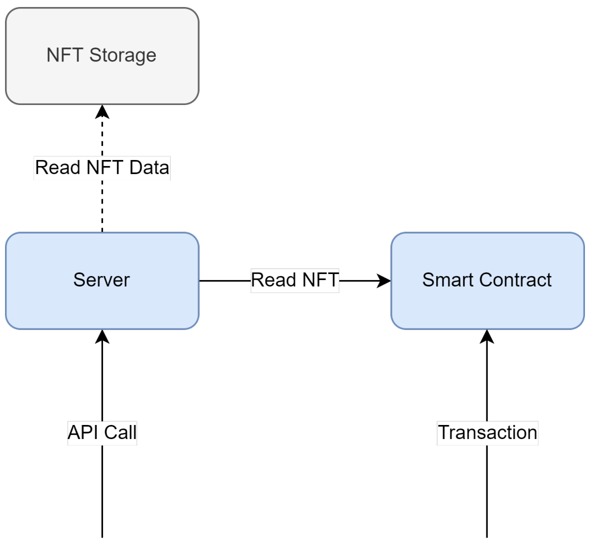


Figure 4.19: NFT Data Storage Model Options

In this approach the server needs to read data from the smart contract, and, optionally, from an off-chain storage (like IPFS or Database).

This will work well for simple use cases, but everything becomes more complicated if we need to have some dynamic data associated with NFTs. E.g we may want to have experience points associated with our game character. Such data can be stored either on-chain or in-application (off-chain storage is also possible, but it's more involved, so we won't discuss it here).

In case of in-application storage, data can be modified by a server without any problems, but there are few drawbacks:

- In order to read this data, clients should make an API call to the server. This adds a centralized point for our NFT, and may not be suitable for all applications.
- If a smart contract requires this data, a server should serve as a **Blockchain Oracle** [78], which complicates things.

If we want our server to serve as an oracle for our smart contract, the easiest way is to cryptographically sign server's data and verify it on the contract's side (server's public key that was used for signing should be stored in a contract in this case).

In order to prevent replay attacks, signed data should include a timestamp, which should also be verified. However, there's one trick to this - smart contracts can't access current time, since it would make them non-deterministic. Instead, transaction signature time can be used - it can be accessed using *env::block_timestamp()* function.

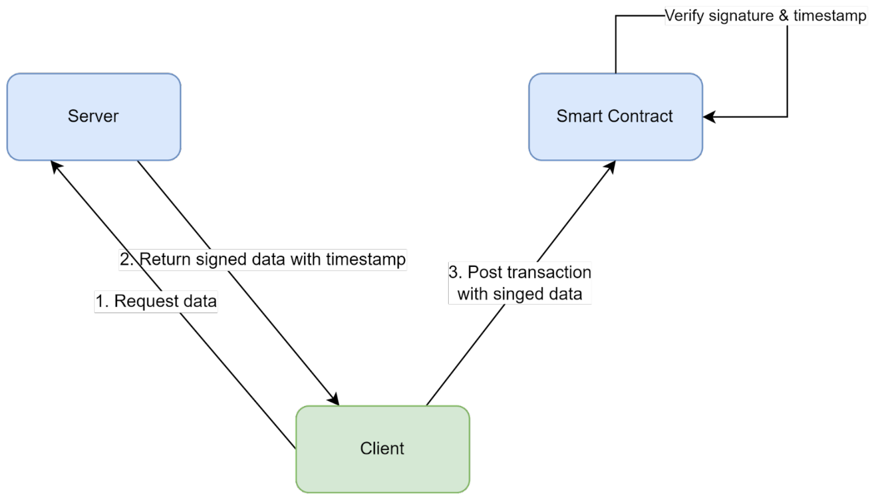

Verify signature & timestamp
Server
Smart Contract
2. Return signed data with timestamp
1. Request data
3. Post transaction with singed data
Client

Figure 4.20: On-Chain Storage Interaction Diagram

In order to avoid all these complications, we can instead store dynamic data on-chain, and use smart contract calls to update it.

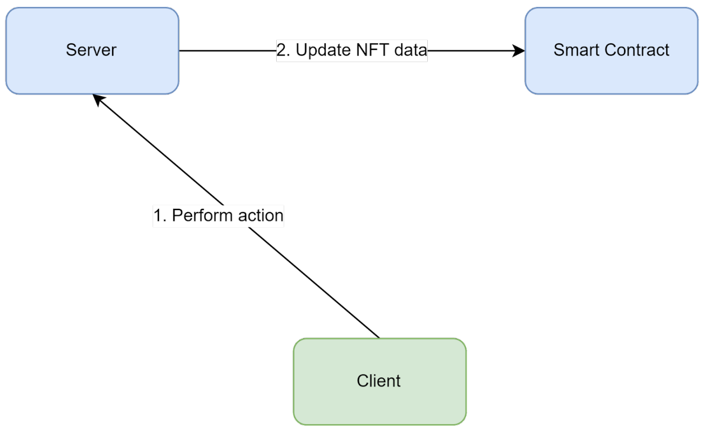


Figure 4.21: Off-Chain Storage Interaction Diagram

Such an approach has one drawback - in order to call a smart contract's method, a transaction should be created by the server, and in order to create a transaction it must be signed using an account's key. That's why a separate NEAR account should be created to be used by the server. Actions on the smart contract can be configured to authorize only this account, so regular users will be disallowed from modifying such data.

Yet another option is to store data on the server-side, but a smart contract can authorize only a server account for calls that rely on this data. As with the previous scenario, the server must have its own NEAR account.

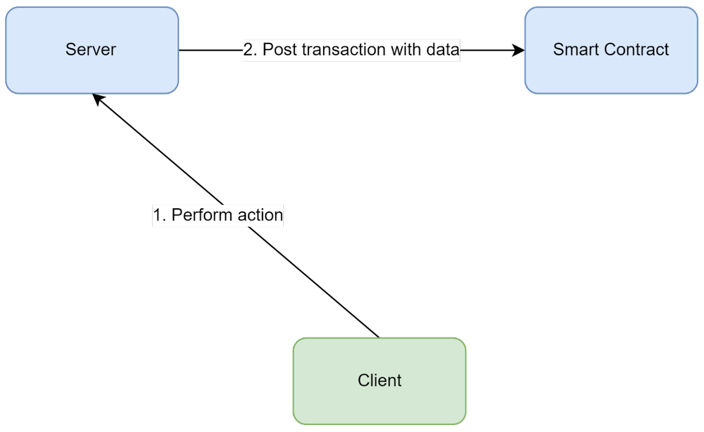


Figure 4.22: In-Application Storage Interaction Diagram

In general, the approach of storing dynamic data on the Smart Contract side is much easier, but an important constraint should be considered - storing data on the blockchain is not cheap, so an appropriate method can be chosen depending on a scenario.

By now, we've covered methods to store and interact with NFTs from our application, an exact strategy should be chosen depending on use cases and constraints. A few things to remember:

- Storing NFTs data in a centralized storage (like an application's database) goes against Web 3 philosophy, and should be used sparingly and with care.
- Storage on the blockchain is not cheap, so decentralized off-chain storages can be used to store large data.
- Storing and using dynamic NFT data is quite tricky, and should be carefully designed. If such dynamic data is needed by smart contracts, it's better to store it inside this contract if possible.

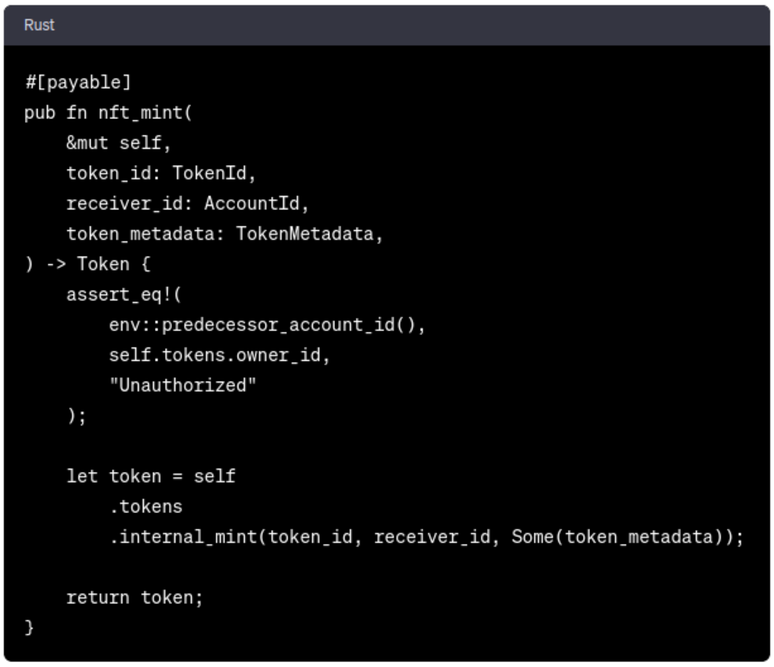
NFT minting

So far, we've discussed only how to use NFTs in the application, but how do they get created?

In the blockchain world, creation of new NFTs is usually called minting. And as with traditional digital assets, there are few ways how to create them:

- Users can mint them directly. This can be done by either allowing creation of NFTs from scratch, or by using more complex processes, like breeding or upgrading. The most famous example of such process is breeding in CrytoKitties game [84] - new NFTs are created by combining existing ones. With this approach users usually have to pay to cover the storage and gas cost of NFTs creation.
- NFTs can be distributed by the developer to a set of users - it is usually called **NFTs airdrop** [85]. Most often this is used as a marketing strategy to kickstart NFTs usage in applications. Storage and gas costs in this case are covered by developers.
- NFTs can be bought on a market or obtained from the lootbox. Depending on an exact strategy, costs can either be paid by a user or by developer. Also, in this case NFTs sometimes can be minted on-demand, to avoid paying upfront costs.

An exact strategy used for NFTs minting depends on application use cases. However, almost always there'll be an *nft_mint_function* defined in a smart contract, which will handle creation of new tokens. This function itself isn't defined in the standard [86] and is up to the application to implement, but the standard library provides a core implementation for it - *mint_internal* [87]. On top of this function an additional logic, e.g. for authorization, can be added:



```rust
#[payable]
pub fn nft_mint(
    &mut self,
    token_id: TokenId,
    receiver_id: AccountId,
    token_metadata: TokenMetadata,
) -> Token {
    assert_eq!(
        env::predecessor_account_id(),
        self.tokens.owner_id,
        "Unauthorized"
    );

    let token = self
        .tokens
        .internal_mint(token_id, receiver_id, Some(token_metadata));

    return token;
}
```

Figure 4.23: Rust nft_mint example

This approach is quite simple, but everything becomes a bit complicated if we want to provide some on-demand minting functionality to avoid paying upfront costs. For example, we may want to create a lootbox with a set of predefined items appearing with some probability.

One approach is to handle this logic on a server side, in this case the server will call *nft_mint* function with computed parameters. However, in this case developers will have to pay the cost of minting. If we want to avoid this, lootbox logic can be moved into the smart contract itself. If users want to open a lootbox, they can call a smart contract function and pay for this action (e.g. by using NEAR or Fungible Tokens). Developers would only need to pay for a lootbox configuration costs, like possible items and their probabilities.

4.2.3 Blockchain Onboarding

Before designing an onboarding strategy, the target audience should be carefully analyzed. As we briefly mentioned before, users can be divided into two broad buckets:

1. Users that are already familiar with blockchain, have their own wallets and understand cryptocurrency basics.

2. *"Casual"* users that aren't familiar with blockchain and don't know much about it.

If only the first category is targeted, then everything is quite simple - users are already familiar with main concepts, and will have no problem connecting their own wallet or creating a new one. However, if we want to target the second category of users as well, a strategy has to be developed to make onboarding into the blockchain world as smooth as possible. While a lot relies on proper UX and is very application-specific, a few architectural patterns and technologies exist to simplify this process: custodial wallets, NEAR drops, Prepaid Gas and Implicit Accounts.

Custodial Wallet [88] is a wallet which is managed by a third party. In this case, a wallet can be created and stored on a server side, and all blockchain operations could be done using the server as a proxy.

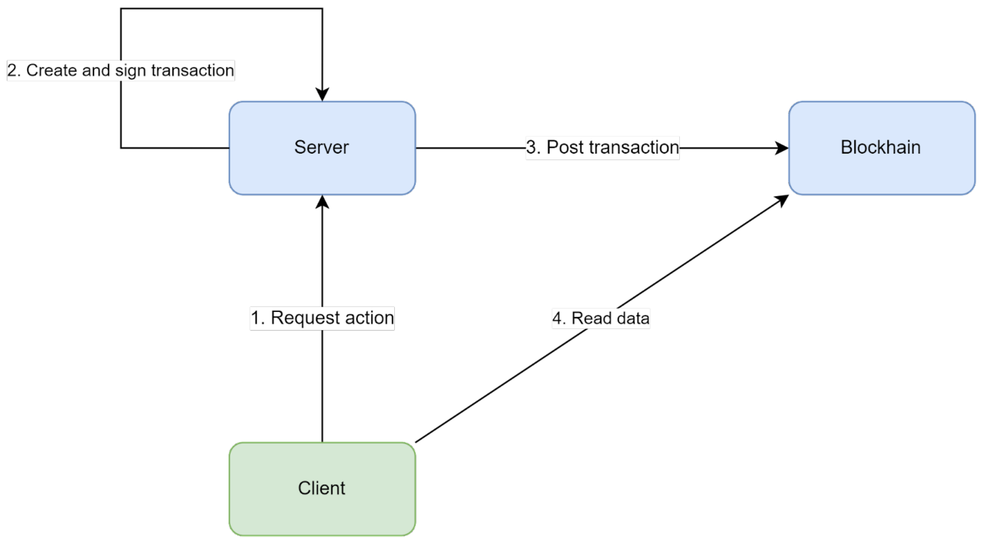


Figure 4.24: Custodial Wallet Concept

In this way, users can remain unaware about the intricacies of blockchain until they are comfortable enough to claim ownership of this account. Once they are ready, the server can transfer the account's ownership and remove it from the server. However, despite simplifying UX for the users, such approach has a few significant drawbacks:

- Users should trust NEAR application to manage their accounts.
- Accounts creation is not free, so unless developers want to pay for it, funds should be transferred from a user to cover this cost. Traditional payment methods can be used, like *PayPal or Apple/Google Pay.* However, such an approach should be used with care for mobile applications due to app stores policies. Alternatively, NEAR Implicit Accounts can be used to avoid paying for account creation.
- Unless they want to leave a custodial wallet as the only supported wallet type, they need to support both types of wallets (custodial and non-custodial) in application. This will increase implementations complexity, since they need to support 2 transaction types:
  - Server-signed transactions in case of custodial wallet.
  - Client-signed transactions in case of non-custodial wallet.

As we mentioned above, **Implicit Accounts** [89] can be used to avoid paying account creation costs. This is especially useful for custodial wallets, since it allows us to create a NEAR Account free of charge. Basically, they work like an Ethereum/Bitcoin-style account by using a public key as an account id, and later can be converted to a full NEAR account. However, they have drawbacks as well. First of all, human-readable account names cannot be used. Also, if you want to convert it to a proper NEAR account, which can support Functional Call keys, the account creation fee still has to be paid.

While being very powerful, custodial accounts are quite complex and tricky to implement. An alternative approach to ease users onboarding is to simplify creation of a wallet itself. In NEAR, you can do this using **NEAR Drops** [90]. It allows us to generate a link that guides users through a quick wallet creation process. However, the same problem as for the custodial accounts applies - creation of an account is not free. That's why, such a link has NEAR tokens attached to it to cover account creation cost and to serve as an initial balance for a newly created wallet. And as with custodial accounts, funds should be transferred from a user to cover this cost using traditional payment channels.

Another option to simplify onboarding is usage of the **Prepaid Gas** concept [91]. For example, you can issue a Functional Call key that allows users to interact with blockchain without having an account created. In this case funds will be drawn from the developer's account. This can be used for demo purposes, or to allow users without a NEAR account to perform some smart contract actions.

4.2.4 NFT Marketplace

At this point, we've covered in detail how to integrate NFTs into our Web 2 application, but we've stayed away from the economy part. The essential part for having a functioning economy is a marketplace where users can freely trade and exchange their NFTs. Such a marketplace usually consists of a smart contract and a client application. This smart contract is closely integrated with a NFT's smart contract using the cross-contract calls. The reason for having a separate smart contract is two-fold:

- This provides a better separation of concerns - we can modify and upgrade our marketplace independently from the NFT contract.
- Multiple marketplaces can be used - e.g. we can have an internal marketplace, and in addition to it users can list their NFTs on external marketplaces. This is possible because a common NFT standard exists that all marketplaces can rely on.

General flow of a simple marketplace integration can look like this:

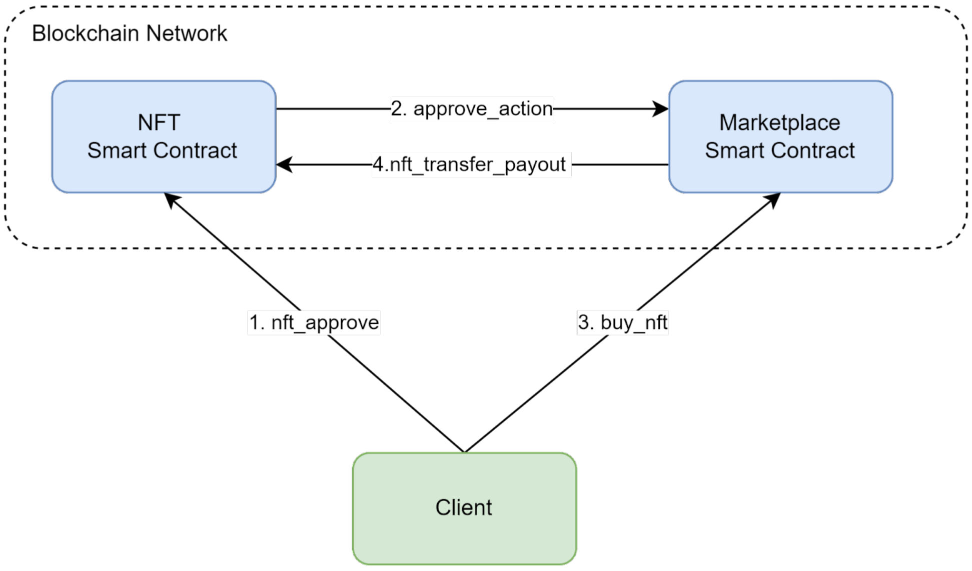


Figure 4.25: Simple Marketplace Integration Flow

1. Client calls the *nft_approve* method [92] on the NFT smart contract. This will approve Marketplace Smart Contract to sell this NFT.

2. After approving an account, NFT smart contract issues a cross-contract call to the Marketplace to create a sale object. Arguments for this call are provided as part of the *nft_approve* call.

3. Another user wants to buy the NFT on sale, so he issues a call to the marketplace contract offering to buy it. An exact call signature for such action is not standardized and depends on marketplace implementation.

4. If an offer to buy a NFT is valid, Marketplace issues an *nft_transfer_payout* [93] call to transfer the NFT and return payout information. This information is used by the Marketplace to distribute profits from the sale between recipients. In the simplest case, all profits go to a seller.

Such flow looks relatively simple, but a few important details are missing. First of all, in order to create a sale, storage needs to be paid for. Usually, the seller is the one who needs to pay for it, but other models are possible - e.g. marketplace or application developers could cover the cost. If we want users to pay for a sale, an approach with storage reservation can be used:

- Before approving NFT for sale, a user should reserve storage on the Marketplace contract to cover sale storage requirements.

- After the NFT is bought or delisted, the user can withdraw storage reservation (remember, that in NEAR storage staking model is used, so data can be deleted and locked tokens refunded).

While this model is relatively straightforward, it's not ideal from the UX perspective - users must make a separate action to reserve storage if they want to sell their NFTs. To improve this, we can combine *nft_approve* call with storage reservation, and automatically refund back the storage cost after the sale is removed.

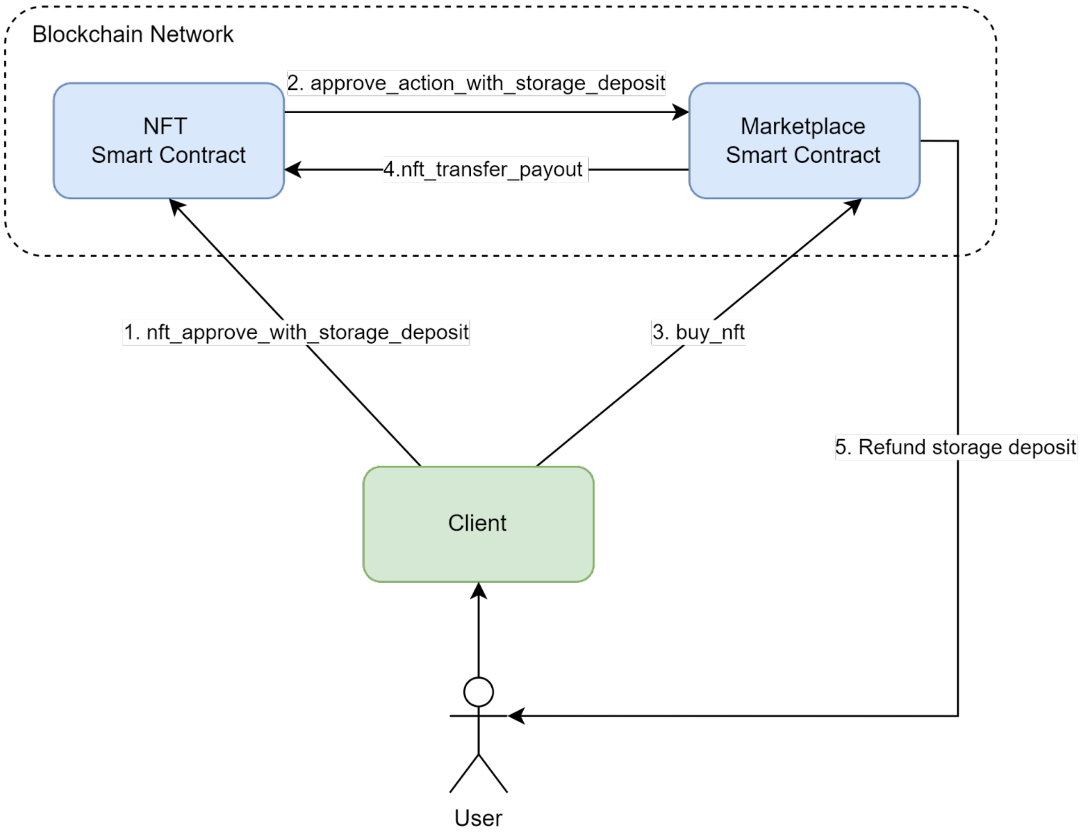

Blockchain Network
NFT Smart Contract
Marketplace Smart Contract
2. approve_action_with_storage_deposit
4.nft_transfer_payout
1. nft_approve_with_storage_deposit
3. buy_nft
5. Refund storage deposit
Client
User

Figure 4.26: Storage Reservation Process in NFT Sales

Another missing thing is how a client can read data about available sales. Of course, sales information can be read directly from a smart contract, but available data structures are not optimized for searching or filtering. Also, we would have to join data from the NFT and Marketplace contracts on the client side, which isn't efficient. In order to solve these problems, an indexer can be used to aggregate data into a suitable database, where data can be stored in a way optimal for retrieval (e.g. a relational database or an ElasticSearch index can be used).

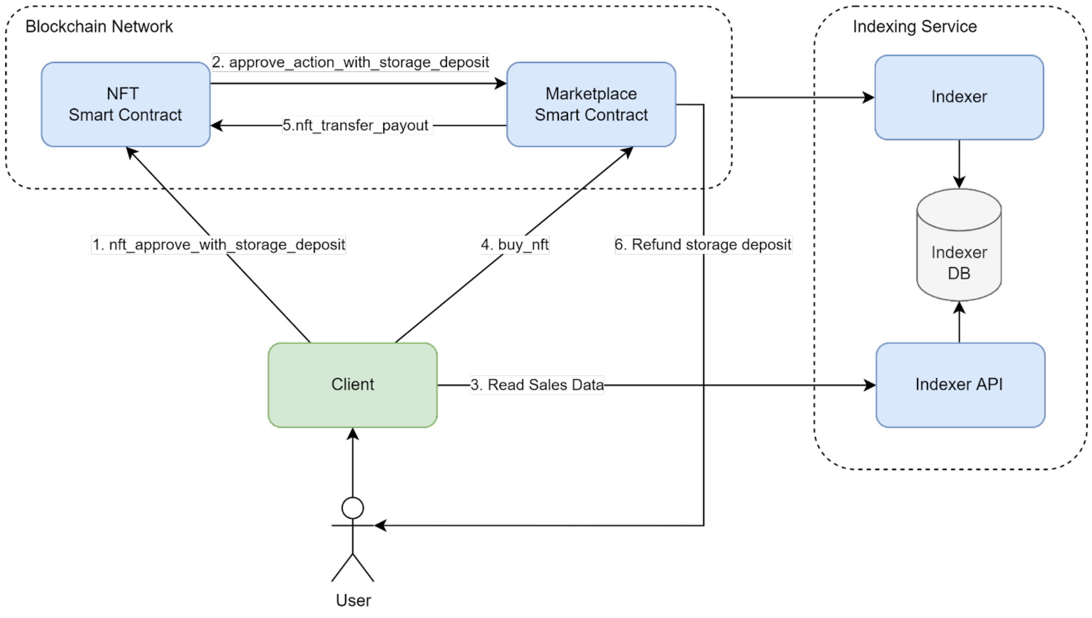


Figure 4.27: Indexing and Data Retrieval in NFT Marketplaces

This is just one example of how a marketplace can be designed, but with it we've covered all basic concepts and problems. Most important points to remember:

- It's better to implement a marketplace as a separate contract.
- Storage management should be carefully designed, with UX in mind.
- In order to implement a proper searching/filtering functionality, a separate indexing service is needed.

A more sophisticated marketplace may allow purchases with Fungible Tokens as payment.

4.2.5 Implementing Components

Now, let's explore NEAR choice of libraries, frameworks and third-party solutions that can be used to implement their architecture.

Client & Server

First of all, how can they interact with blockchain from their clients? If they need read-level access only, they can simply use the **REST API** [42], so it can be integrated into any language and technology without any problems. But everything becomes more complicated if they need to post transactions from a client. Remember, that transaction should be signed with a private key which is stored in a wallet:

- In case of a **Functional Call key**, it can be obtained from the wallet and used directly by the client.
- In case of a **Full Access key**, the user should be redirected to the wallet to approve a transaction.

A **JavaScript API** [94] exists to cover all of these scenarios. It has all of the necessary functionality to integrate Web/Node.JS applications with blockchain. This SDK is a perfect choice for the Web-based clients, but it's not suitable for desktop or mobile based clients. Other libraries can be used for them:

.NET Client [95] - suitable for Unity or Xamarin. **Swift** [96], textbfPython [97] and **Unity** [98].

Same SDKs and libraries can be used for servers. The only difference is that a server cannot interact with a Wallet, so it must have access to a Full Access key, which should be stored and accessed in a secure way. Also, another solution is available if a server uses a technology that doesn't have NEAR SDK available for - we can create a separate (micro)service using the Node.js, which would handle all blockchain interactions:

Contracts

As we discovered in a previous section, for our application we need two smart contracts: for NFT and for Marketplace. There are two options on how to get them - use in-house implementation or some *third-party/SAAS* solution. Both options are viable and have different *pros/cons.*

If we want to create our own contract, we are fully in control and can implement anything we want. An obvious drawback, of course, is that it will take time and money to build it. Third-party solutions, on the other hand, are limited in their functionality and often cannot be easily extended. Also, they usually have some upfront costs and/or usage fees.

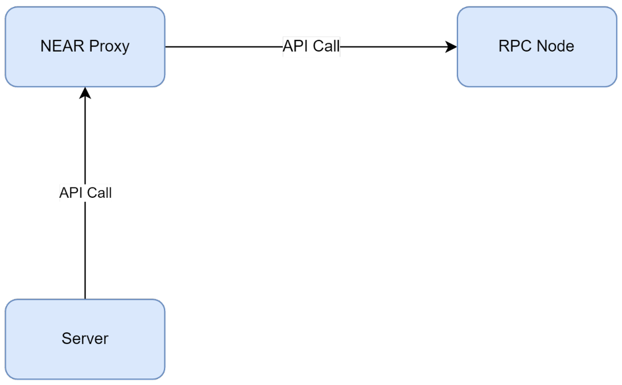


Figure 4.28: Proxy Server for Blockchain Interactions

For an in-house NFT contract implementation a few resources can be used as a starting point. First of all, a **Rust library** [99] is available which implements most of the standard. Another option is to build an entire contract from scratch, a good guide on how to do this is available by this link.

Implementing an own Marketplace contract is more involved since there is no standard implementation.

As for third-party solutions, the most complete one is **Mintbase** [100], which provides a full suite of components for NFTs integration - including contracts, indexer, API and a web client:

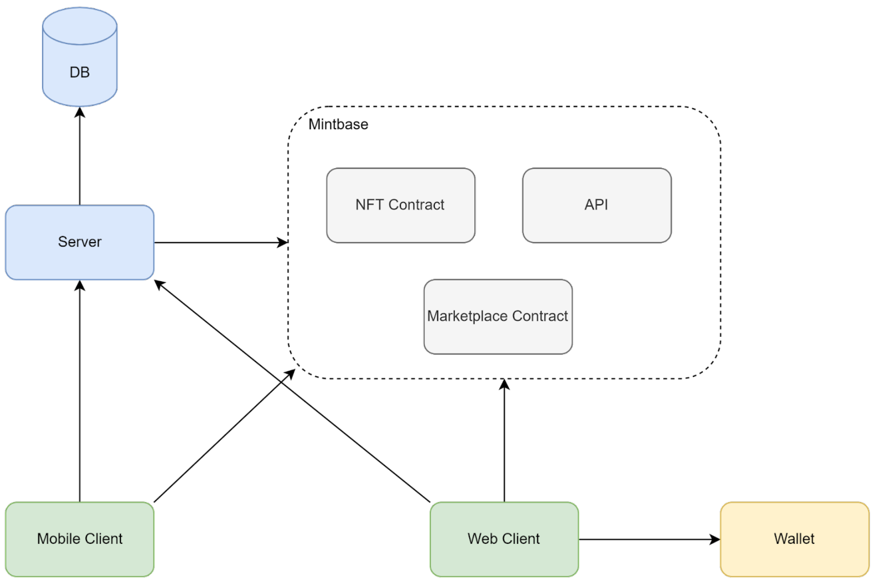


Figure 4.29: Mintbase Suite for NFT Integration

Another option is to roll-out an own NFT contract and integrate with one of the third-party marketplaces, e.g. with **Paras** [101].

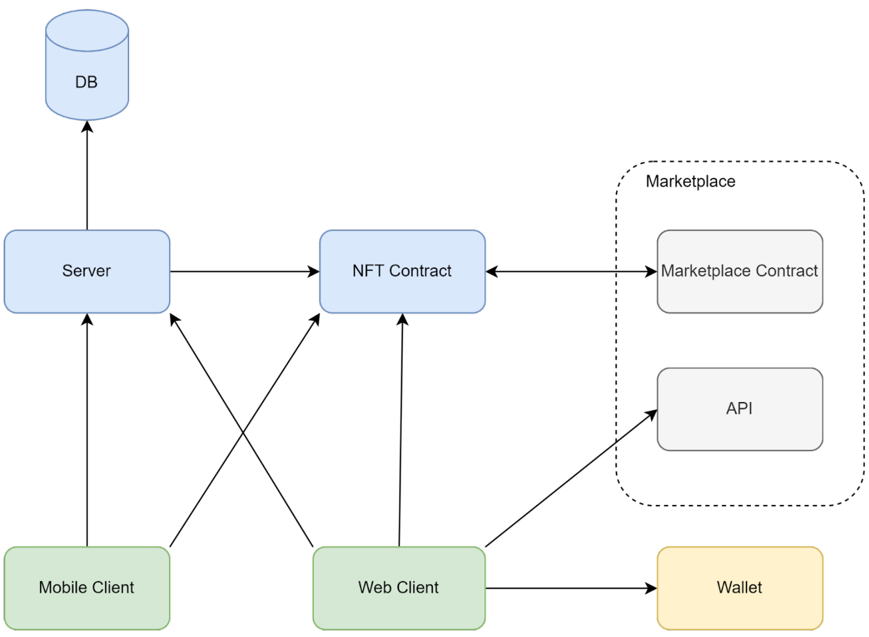


Figure 4.30: Integration with Third-Party Marketplaces

The major advantage of an external marketplace is the fact that they usually run their own indexer and expose collected data via an API, so we don't have to implement these components. However, they usually have their fee for providing them, so a cost-benefit analysis should be conducted before using them.

Off-chain storages

Previously, we've discussed that storage on the blockchain is not cheap, so in most cases some decentralized storage solution should be used. A few options are available:

- **IPFS** [102] - one of the first decentralized storage solutions, which is widely used in the blockchain world. However, in order to make sure that data is preserved on the network, an IPFS node(s) should be maintained.
- **Arweawe** [103] - blockchain-based decentralized storage.
- **Filecoin** [104] - another blockchain-based storage.

- **nft.storage** [105] - a free service built on top of the IPFS and Filecoin. In our case, we have decided to use nft.storage for our dApp, more details can be found in chapter **5.6.2**.

A more in-depth overview of such solutions is available in the docs. In general, there's no *"silver bullet"*, so different solutions should be evaluated and the most suitable chosen. The main concerns while choosing a solution are availability guarantees, and cost.

Indexer

As we already determined, an indexing service is needed in order to support marketplace functionality. It usually consists of 3 components:

- Indexer - processes transactions from a NEAR network and puts extracted data into a database.
- Database - database of choice to store extracted data.
- Indexer API - an API layer on top of the database.

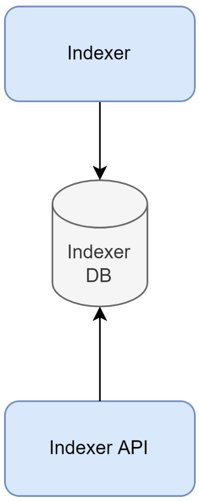


Figure 4.31: Components of an Indexing Service in NFT Marketplaces

While any technology of choice can be used to implement Database and API, an indexer itself is usually implemented using Rust, since a framework is available [106] for this language. Guide how to implement your own indexer can be found here [53].

Usually, an indexer works by extracting data from **Events** [56], which are basically just structured log messages written during contract execution.

The Graph [57] is an alternative to building an indexer from scratch. This is an Indexer-as-a-Service solution, which simplifies their creation and deployment.

Automated Testing

Automated testing of the code is one of the pillars of modern software development. But how do we test our dApp?

Recall that a smart contract is a pure function, which can be easily tested using Unit Tests. Guide on how to write them is available here [107]. Another important kind of tests that is supported by NEAR are E2E tests, they can be executed either deploying contract code to either the local network environment (more info here [108]), or directly to ***testnet*** (more info here [109]).

Having both types of tests is equally important to ensure continuous quality of smart contracts, especially since contract upgrades usually aren't easy to perform (remember, that in DAOs upgrade itself might be governed by a community vote).

4.3 Non-Functional Concerns

Last, but not least, let's cover important non-functional concerns for our architecture.

4.3.1 Security

The most important thing to remember during the entire development is security, and especially the security of smart contracts. Since their code is public and an upgrade procedure is not trivial, it should be carefully audited for security issues and logical exploits.

Another important thing that should be kept secure is a user's private key. In most cases, only Functional Call keys should be directly accessed from a client, and Full Access keys should be kept in a wallet. However, in some cases a Full Access key might have to be used directly (e.g. in case of server transaction signing scenarios). In such a case, it must be kept in a secure location and treated as a most sensitive secret, since its compromise might lead to a full account takeover.

In general, before deploying an application to the NEAR mainnet, it's a good idea to conduct a full security audit.

4.3.2 Scalability and Availability

Another concern is scalability and availability of a solution. There are a lot of ways to scale traditional servers, but how do we scale NEAR blockchain and make sure it's always available?

Since blockchain is decentralized, it provides us with high-availability by design, and NEAR provides a great scalability by employing Proof-of-Stake consensus and sharding. However, in order to interact with a network, it needs an RPC Node. NEAR maintains publicly available nodes for its networks (**listed here [110]**), but it doesn't provide any performance or availability guarantees for them. So, in order to make sure their architecture is scalable and fault tolerant, NEAR maintains their own cluster of RPC nodes, typically behind a load balancer.

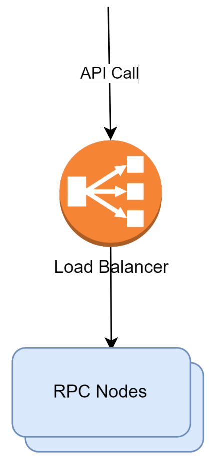


Figure 4.32: RPC Nodes and IPFS Integration

Information on how to set up an RPC node is available here [111].

Also, to guarantee availability and scalability of a whole system, all used third-party services should be reviewed as well. For example, if IPFS is used as a storage for NFTs, pinning nodes and IPFS gateway should be scalable and fault tolerant.

4.3.3 Costs

When building a Web 3 application, it's important to remember that cost calculation is somewhat different from Web 2 applications. Additional costs can be broken down into several categories:

1. Smart Contracts deployment costs. While deploying on NEAR testnet or local environment, it's essentially free of charge. However, when deploying

into the mainnet, developers will be charged for storage and gas cost. Gas cost for a contract deployment transaction is relatively small (around 0.04$ at the time of writing). On the other hand, storage costs can be quite substantial, e.g. a 150KB contract (compiled) will cost around 20$.

2. Smart Contracts usage cost. In Web 3, users pay for smart contract calls, so in order to make sure users aren't discouraged to interact with a contract due to a high cost, it should be optimized to incur the lowest cost possible. This is especially important for storage costs, since gas is relatively cheap.

3. If we want to use a privately hosted RPC node for better availability, its operational costs should be taken into account as well. Cost breakdown can be found here [112], a rough estimation is about 290$ per node per month (and remember that it needs at least 2 nodes for redundancy).

4. Cost of a privately hosted indexer (if it's used). More information can be found here [54], a rough estimation for the costs is about 100$ per month.

5. Third party services costs.

Chapter 5

Setting Up the Development Environment

This chapter focuses on the comprehensive setup required for our dApp. Our dApp is designed to create unique NFTs that reflect userMood, userQuotes, userCity, and userWeather. It also facilitates user authentication and manages a marketplace. To achieve this, the dApp harnesses the power of NEAR wallets for blockchain operations, supplemented by external APIs.

In the development of our NEAR Protocol-based dApp, we strategically employ a variety of APIs to enhance functionality and user experience. This integration is critical for realizing the full potential of our application. Indexer, NFT.STORAGE, Google Map, OpenAI and AccuWeather APIs are instrumental in providing essential real-time data integration for incorporating into NFT attributes.

These APIs, in conjunction with the NEAR Protocol, form a powerful combination that allows for the creation of a sophisticated, feature-rich dApp. They facilitate a broad range of functionalities, from blockchain interactions and data storage to creative media generation and real-time data incorporation, thereby enriching the overall user experience and expanding the capabilities of our dApp beyond the conventional blockchain framework.

5.1 Foundations of NEAR Protocol Development

Developing on the NEAR Protocol encompasses understanding the use of ReactJS for frontend JavaScript library for building user interfaces based on components, Rust for smart contract development, and various APIs for extended functionality. The setup process equips developers with a suite of tools to write, test, and deploy smart contracts and create interactive user interfaces for dApps.

5.2 Prerequisites for NEAR dApp Development

The development journey begins with the installation of Node.js and npm, foundational tools for JavaScript execution and package management. Using the apt package manager, the installation commands are executed, followed by verification steps to ensure successful setup.

Node.js and npm setup commands:

```
sudo apt update
sudo apt install nodejs npm
```

Check the installation with ***node -v*** and ***npm -v***.

For improved package management and developer experience, Yarn and Nodemon are installed globally. Yarn offers efficient dependency management, while Nodemon provides a live development server that watches for file changes.

Yarn and Nodemon installation commands:

```
sudo npm install -g yarn nodemon
```

5.3 Rust and WebAssembly Toolchain Installation

Rust is favored for smart contract development due to its emphasis on safety and performance. The **rustup** tool facilitates Rust installation and management. After installation, the wasm32-unknown-unknown target is added to compile Rust into Wasm, enabling smart contracts to run on the NEAR Protocol.

Rust installation commands:

```
curl –proto ’=https’ –tlsv1.2 https://sh.rustup.rs -sSf | sh
source $HOME/.cargo/env
rustup target add wasm32-unknown-unknown
```

5.4 NEAR CLI and near-api-js Library

The NEAR Command Line Interface (CLI) is indispensable for direct interaction with the NEAR network. Alongside, the near-api-js library provides a JavaScript interface to connect with NEAR nodes.

NEAR CLI and near-api-js installation commands globally:

```
sudo npm install -g near-cli
sudo npm install -g near-api-js
```

5.5 Starting Development with NEAR

Two pathways for initiating NEAR development are presented: creating a new project using *create-near-app* for a "Hello World" example, or cloning the *nft-tutorial-frontend* repository for a pre-configured NFT project with wallet login functionality.

Commands for the "Hello World" NEAR application setup:

npx create-near-app your-awesome-project

cd your-awesome-project

yarn install

yarn dev

This sets up a local server, and developers can start editing the ***contract/src/lib.rs*** file to write smart contracts in Rust or the ***src/*** directory for frontend changes in ReactJS.

Commands for cloning and setting up a more complex starting point, clone the ***nft-tutorial-frontend*** repository, which provides a simple NEAR login with wallet functionality:

git clone https://github.com/near-examples/nft-tutorial-frontend.git

cd nft-tutorial-frontend

yarn install

yarn dev

The repository provides a structured template to build NFT applications on NEAR, complete with login mechanisms and contract interactions. Also this sets up a local server, and developers can start editing the ***contract/src/lib.rs*** file to write smart contracts in Rust or the ***src/*** directory for frontend changes in ReactJS.

5.6 Environment Configuration and API Integration

The ***.env.local*** file is recommended for managing environment variables securely. API credentials for services like Indexer, NFT.STORAGE, Google Map, OpenAI and AccuWeather are added to the file, enabling seamless interaction within the dApp.

APIs play a crucial role in the development of a dApp, extending its capabilities beyond the blockchain. This section guides through the setup and integration of various APIs within the NEAR dApp environment.

5.6.1 INDEXER.XYZ API for NEAR

The Indexer API, typically used for querying blockchain data efficiently, is vital for fetching information about transactions, accounts, and smart contracts. Also we use the Indexer API to populate the marketplace with current NFT data and transaction history.

To set up, we obtain an API user and key from a service like indexer.xyz for Explorer.

5.6.2 NFT.STORAGE API for Decentralized Storage

nft.storage API facilitates the decentralized storage of NFT assets on IPFS, ensuring that the data persists immutably. We use this API to store and retrieve NFT metadata, media, and other associated content.

We sign up at nft.storage and generate an API key.

5.6.3 Google Maps API for Geolocation Services

Google Maps API integrates mapping and location-based services, allowing users to visualize geographical data within the dApp. We obtain an API key from the Google Cloud Platform. In the dApp, we use this API to display minted NFT in each city by targeting the userMoods on a map or to show a city's weather forecast.

5.6.4 OpenAI API for Generative Models

The OpenAI API connects to powerful AI models capable of generating images, text, or other data based on inputs, moderated to ensure compliance with content policies. We create an account at OpenAI and receive your API key. We also create a new moderation to prevent the acceptance of names of famous people, names of cities and historical monuments.

We leverage this API to create images from user-provided moods and quotes, forming a unique aspect of the minted NFTs.

In future releases, we will try to build a Natural Language Processing (NLP) model and adjust it like an endpoint to safely prevent NFT media generation according to our policies.

5.6.5 AccuWeather API for Weather Data

AccuWeather's API provides accurate and timely weather forecasts, essential for incorporating real-time weather conditions into NFT attributes. We register for the AccuWeather API and obtain our key.

We use the AccuWeather API to fetch weather data for a given location, enhancing the user experience by offering personalized NFTs that reflect current meteorological conditions.

5.7 Managing Dependencies

The package.json file details a comprehensive suite of dependencies integral to the development and functionality of our dApp. Among these, a significant focus is on the NEAR Protocol and Web3 technologies, facilitated through various packages.

Key NEAR Protocol dependencies include ***@near-wallet-selector/core*** and ***@near-wallet-selector/wallet-utils***. These packages are pivotal in integrating NEAR wallet functionalities into the dApp, allowing for seamless user authentication and transaction signing. The wallet selector modules enable users to choose their preferred wallet interface, enhancing the user experience.

The Web3 technology stack, particularly through the ***web3.js*** package, plays a vital role in our dApp's blockchain interactions. This package provides the necessary tools for interacting with the Ethereum blockchain, including sending transactions, interacting with smart contracts, and handling blockchain-related data. A notable use case in our application is converting cryptocurrency values from Wei to more readable formats, such as ***Mether***. For instance, we employ ***Web3.utils.fromWei(nft.price_str, 'mether')*** to display NFT prices in a user-friendly format, especially when fetching data from indexers like **indexre.xyz**, where prices are often listed in **Wei** [113].

Additionally, we utilize the ***openai*** package, a key element for integrating AI-driven features from OpenAI into our dApp. This package is primarily employed to generate images using OpenAI's DALL-E, a state-of-the-art image generation model. By incorporating this capability, our dApp is able to produce unique, AI-generated visuals based on user inputs or specific data points. This integration not only enriches the user experience but also demonstrates the potential of combining blockchain technology with advanced AI applications in creative ways.

The mathematical precision in our dApp is maintained by two important libraries: ***bignumber.js*** and ***bn.js***. ***bignumber.js*** is utilized for precise arithmetic operations, especially crucial in financial applications where accuracy is paramount. ***bn.js*** is another library that aids in handling big numbers, especially when dealing with blockchain-related calculations where standard JavaScript number precision is insufficient.

Together, these packages form the backbone of our dApp, ensuring robust functionality, user-friendly interactions, and precise operations essential in the blockchain and AI-driven landscape.

5.8 Creating and Managing a NEAR Account

Developers require a NEAR account to deploy contracts and perform transactions. The account creation process is facilitated through the NEAR Wallet website [114]. Additionally, developers can create sub-accounts using the CLI with the following command:

To log in to your NEAR account, use the command below and follow the in-browser steps:

```
near login
```

It is important to note that while smart contracts can be deployed on master accounts, there are scenarios where creating additional testing accounts is beneficial to test specific functionalities. In such cases, it is preferable to deploy contracts on a sub-account. It's vital to understand that each account or sub-account can maintain only one smart contract at a time:

```
near create-account <sub-account.your-testnet-account>.testnet –masterAccount <your-testnet-account>.testnet
```

5.9 Deploying Smart Contracts

After account setup, smart contracts can be compiled to Wasm and deployed on the NEAR network using the NEAR CLI with commands tailored to the project's specific contract and account details.

```
near deploy –wasmFile <path-to-contract>.wasm –accountId <sub-account.your-testnet-account>.testnet
```

5.10 Handling Errors and Debugging

In case of deployment errors, use tools like *env-cmd* help manage environment variables during runtime and *parcel-bundler* to manage environment configurations and bundle applications, respectively. Keep Node.js updated for the best compatibility using:

```
sudo npm cache clean -f
sudo npm install -g n
sudo n stable
```

This chapter serves to document the technical environment and toolchain setup in the creation of the NEAR dApp, providing insights into the integration of blockchain technology with external APIs to deliver a comprehensive user experience.

Chapter 6

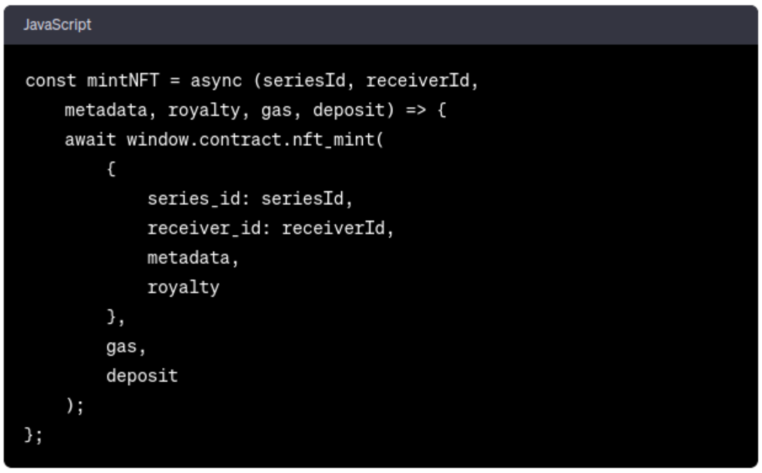
Integrating NFTs in Web Apps and User Interaction Flows

6.1 Smart Contract Interaction

Developing dApps on NEAR protocol involves a seamless integration of smart contracts with a user-friendly frontend. The smart contracts written in Rust provide the immutable logic and rules that govern the behavior of the dApp. On the other hand, the frontend allows users to interact with the smart contracts through a graphical user interface.

6.1.1 JavaScript Functions for Smart Contract Methods

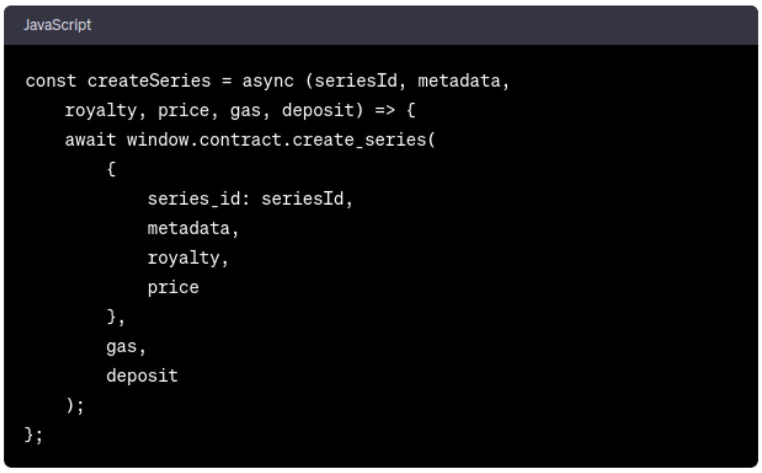
The smart contract's capabilities are made accessible on the frontend through JavaScript functions, which communicate with the contract's methods. Here are the JavaScript counterparts for the Rust functions:



```
const mintNFT = async (seriesId, receiverId,
    metadata, royalty, gas, deposit) => {
    await window.contract.nft_mint(
        {
            series_id: seriesId,
            receiver_id: receiverId,
            metadata,
            royalty
        },
        gas,
        deposit
    );
};
```

Figure 6.1: JavaScript function for nft_mint

- **nft__mint**: This function allows approved users to create new NFTs within a series. Each NFT inherits metadata and royalty information from the series it belongs to. The function enforces any limits on the number of copies and handles financial transactions necessary for minting.

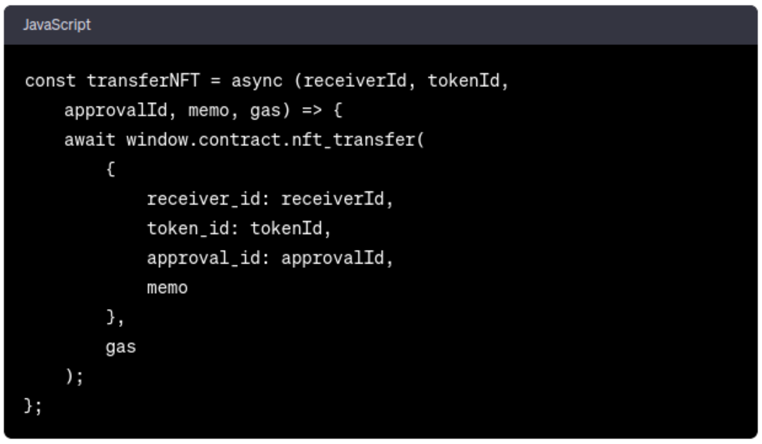


JavaScript

```javascript
const createSeries = async (seriesId, metadata,
    royalty, price, gas, deposit) => {
    await window.contract.create_series(
        {
            series_id: seriesId,
            metadata,
            royalty,
            price
        },
        gas,
        deposit
    );
};
```

Figure 6.2: JavaScript function for create__series

- **create__series**: Approved creators can use this function to establish new NFT series, setting metadata, royalty, and pricing information. The function ensures uniqueness of series and manages storage costs.

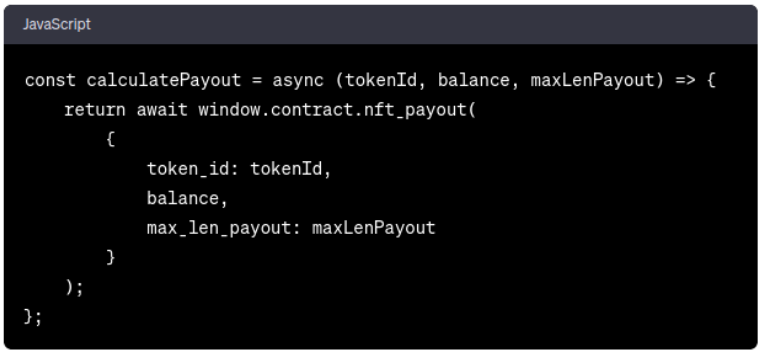


JavaScript

```javascript
const transferNFT = async (receiverId, tokenId,
    approvalId, memo, gas) => {
    await window.contract.nft_transfer(
        {
            receiver_id: receiverId,
            token_id: tokenId,
            approval_id: approvalId,
            memo
        },
        gas
    );
};
```

Figure 6.3: JavaScript function for nft__transfer

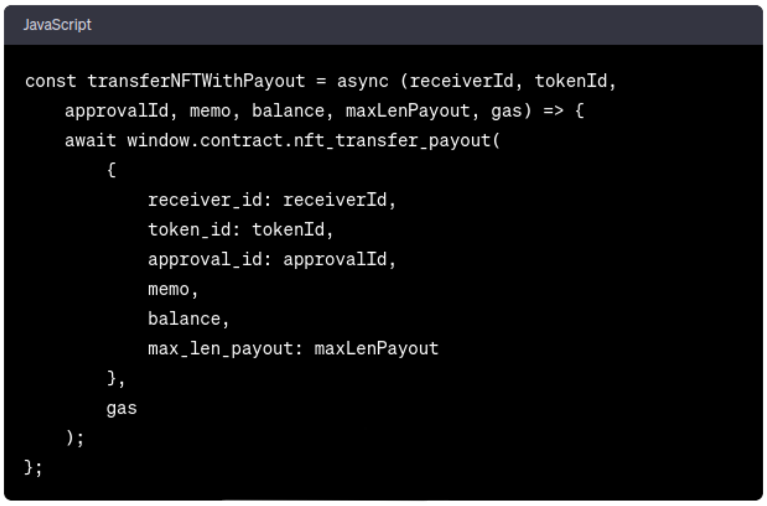
- **nft__transfer**: It enables the transfer of NFT ownership from one user to another, ensuring that only authorized accounts can initiate the transfer.


JavaScript

```javascript
const calculatePayout = async (tokenId, balance, maxLenPayout) => {
    return await window.contract.nft_payout(
        {
            token_id: tokenId,
            balance,
            max_len_payout: maxLenPayout
        }
    );
};
```

Figure 6.4: JavaScript function for nft_payout

- **nft_payout**: This function calculates the payout distribution for an NFT based on its royalty information when a sale occurs.


JavaScript

```javascript
const transferNFTWithPayout = async (receiverId, tokenId,
    approvalId, memo, balance, maxLenPayout, gas) => {
    await window.contract.nft_transfer_payout(
        {
            receiver_id: receiverId,
            token_id: tokenId,
            approval_id: approvalId,
            memo,
            balance,
            max_len_payout: maxLenPayout
        },
        gas
    );
};
```

Figure 6.5: JavaScript function for nft_transfer_payout

- **nft_transfer_payout**: Similar to **nft_transfer**, this function not only transfers the NFT but also returns a payout object detailing the royalty distribution.

6.1.2 Frontend Integration with Smart Contracts

The frontend of our dApp is built using JavaScript and interacts with the smart contract through NEAR's API. The following are some of the methods used in the frontend to communicate with the smart contract:

- **View Methods**: These are read-only calls to the blockchain that do not modify the state and typically do not require gas. Examples include:
  - **get_series**: Retrieves details of a specific NFT series.
  - **get_series_details**: Fetches metadata of a series.
  - **nft_tokens**: Lists NFT tokens available in the contract.
  - **nft_tokens_for_owner**: Lists all NFTs owned by a specific account.
- **Change Methods**: These methods can alter the state on the blockchain, such as transferring tokens or minting new ones. These operations require gas to perform. Examples include:
  - **nft_mint**: Mints a new NFT.
  - **create_series**: Creates a new series for NFTs.

6.2 Frontend User Interaction Flow

The following subsections will guide through the frontend application's key pages, detailing the user interaction flow and integration with the smart contracts.

6.2.1 Login Page

The dApp begins with a login page accessible to all, including new users. Here, users can authenticate using their Gmail or Facebook account, also NEAR wallet login which connects them to the NEAR blockchain and enables interaction with the smart contracts. Notably, even without signing in, new users can view the main page, market page and user profiles but cannot mint or transfer NFTs.

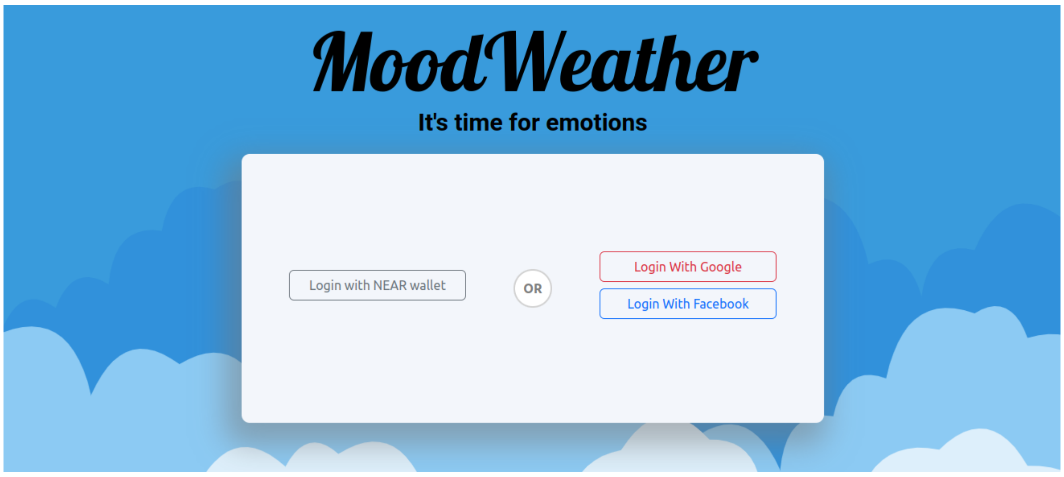
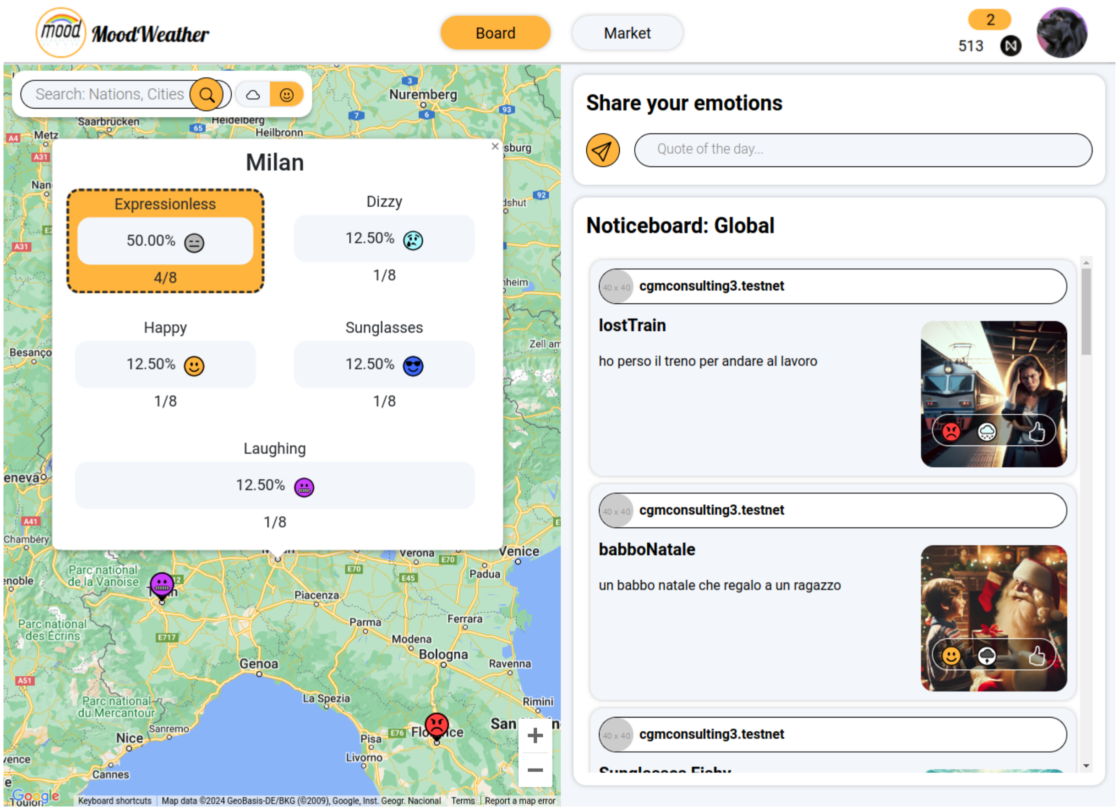


Figure 6.6: Showcasing the authentication options

6.2.2 Interactive Main Interface

Upon visiting the main interface of the dApp, even non-registered visitors are greeted with a dynamic and interactive overview. Central to this page is the ability to search for cities, allowing users to immediately gauge the collective mood of a selected location. The mood data is represented as a percentage, visualizing the emotional climate of the city through expressive icons and color-coding. Additionally, users are treated to an integrated weather forecast feature, sourcing its data from AccuWeather to provide accurate and current meteorological conditions alongside the mood representation.



Milan

Expressionless	Dizzy
50.00%	12.50%
4/8	1/8

Happy	Sunglasses
12.50%	12.50%
1/8	1/8

Laughing
12.50%
1/8

cgmconsulting3.testnet
lostTrain
ho perso il treno per andare al lavoro

cgmconsulting3.testnet
babboNatale
un babbo natale che regalo a un ragazzo

cgmconsulting3.testnet

Figure 6.7: Interactive mood visualization and search functionality on the dApp's main interface

This seamless integration of mood metrics and weather forecasts offers users an immersive snapshot of the city's current atmosphere, combining both emotional and environmental data. The main page serves as a gateway, offering insights into the latest NFT collections, enabling direct transactions within the marketplace, and fostering a sense of community through shared sentiments.

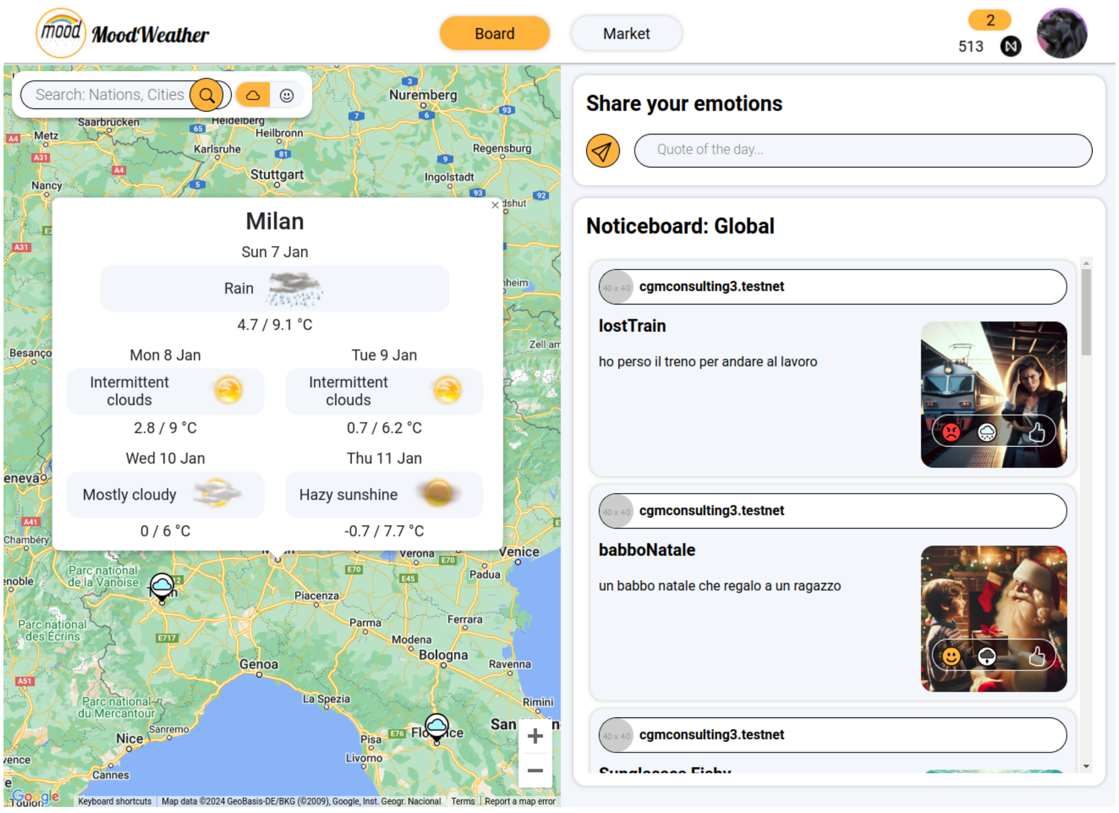


Figure 6.8: Integrated weather forecast feature showing current conditions and predictions

6.2.3 Minting NFT Pop-ups

The user interaction flow on the frontend is designed to provide a smooth and intuitive experience. Users can perform actions such as creating new NFT series, minting NFTs, transferring ownership, and viewing their NFT collection. Here's a typical flow for a user:

1. The user logs in to the dApp using their NEAR wallet and the first step allows users to generate an image using OpenAI, based on their mood and a text prompt.
2. In the second step, users provide an NFT title and select an existing collection or create a new one with specified metadata and royalty details.
3. The final step involves sending data to NFT.storage, approving the minting process.
4. Viewing the newly minted NFT on the main page. Explore the tx here [115].

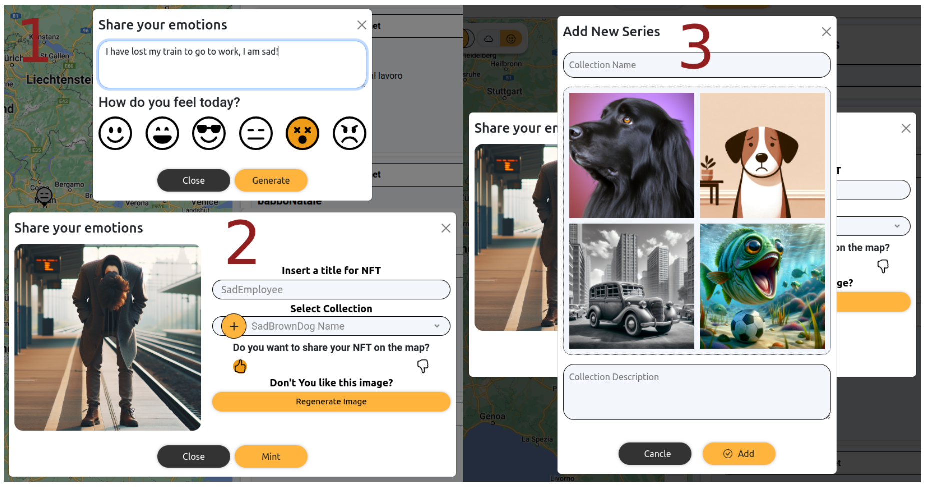
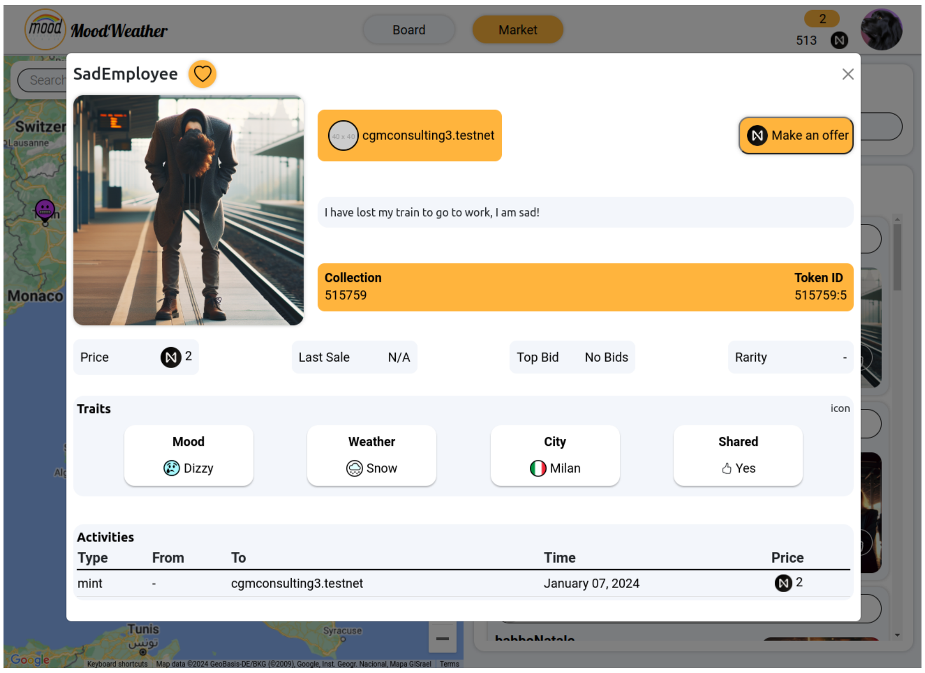


Figure 6.9: Generate an image using Generative AI model, NFT title and select a collection and Sending data to nft.storage, approving the minting process

6.2.4 NFT Details Popup

Clicking on an NFT triggers a popup that displays detailed attributes such as user mood, quote, city, and weather — all stored off-chain on nft.storage.



cgmconsulting3.testnet

I have lost my train to go to work, I am sad!

Collection 515759 — Token ID 515759:5

Price 2 | Last Sale N/A | Top Bid No Bids | Rarity -

Traits

Mood	Weather	City	Shared
Dizzy	Snow	Milan	Yes

Activities

Type	From	To	Time	Price
mint	-	cgmconsulting3.testnet	January 07, 2024	2

Figure 6.10: NFT details popup, highlights the off-chain stored attributes

6.2.5 Market Page

The market page serves as the central hub for economic interactions within the dApp. Here, users can browse, purchase, or place bids on NFTs offered by others. The page is designed to facilitate easy navigation and interaction with the diverse range of NFTs available.

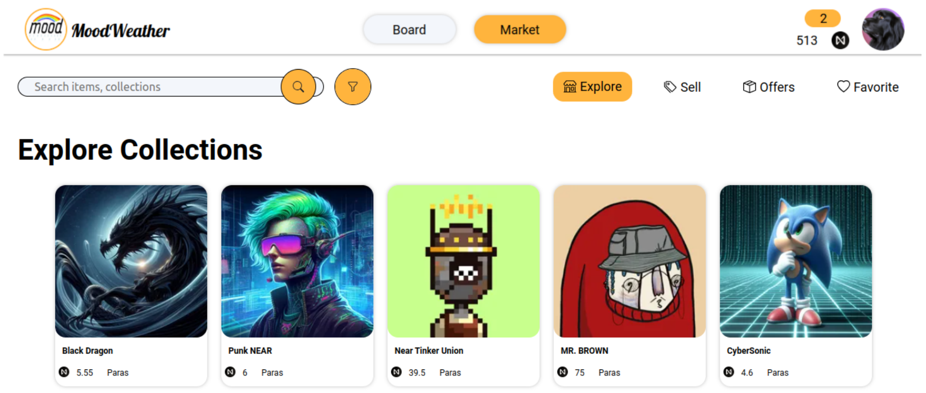


Figure 6.11: Showcasing the marketplace functionality and available NFTs

6.2.6 Collection Page

Each user can visit their collection page to view the NFTs they've created or acquired. This page acts as a personal gallery within the dApp.

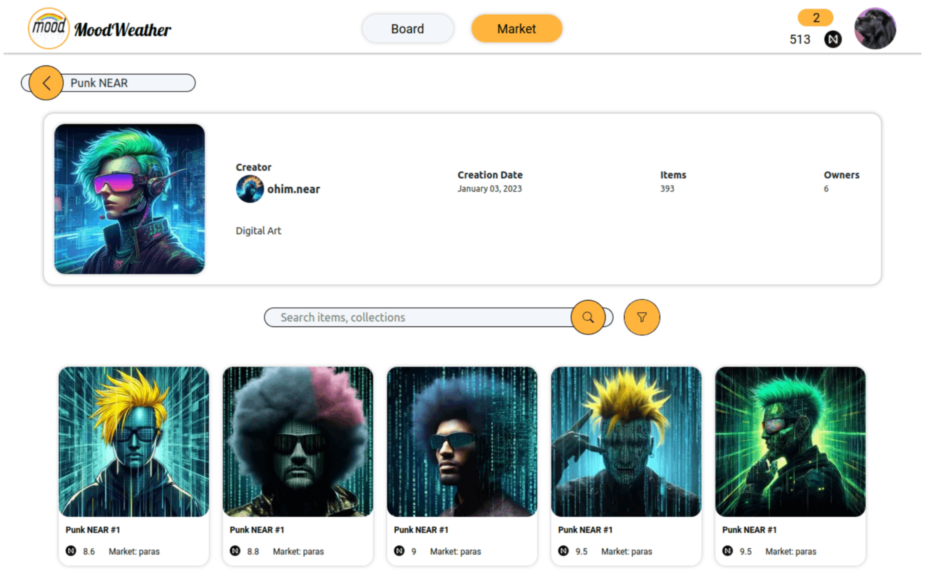
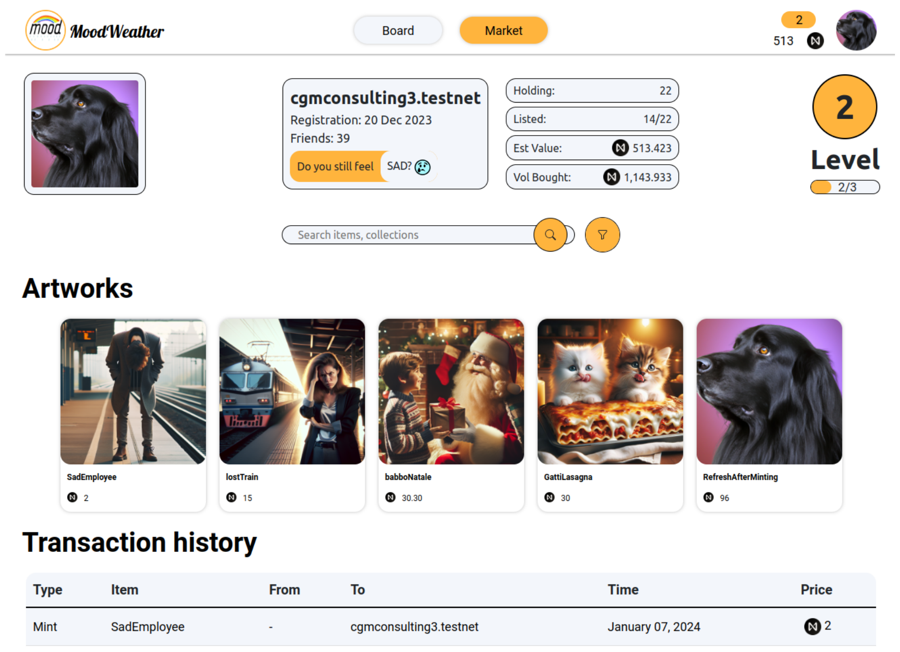


Figure 6.12: User's collection page, displays the variety of NFTs owned by the user

6.2.7 User Profile

The user profile page offers a detailed overview of an individual's engagements within the dApp. It includes their NFT holdings, transaction history, and allows for the transfer of NFT ownership. This page is personalized and reflects the unique activity and preferences of the user.



cgmconsulting3.testnet
Registration: 20 Dec 2023
Friends: 39
Do you still feel SAD?

Holding:	22
Listed:	14/22
Est Value:	513.423
Vol Bought:	1,143.933

2
Level
2/3

Artworks

SadEmployee	2
lostTrain	15
babboNatale	30.30
GattiLasagna	30
RefreshAfterMinting	96

Transaction history

Type	Item	From	To	Time	Price
Mint	SadEmployee	-	cgmconsulting3.testnet	January 07, 2024	2

Figure 6.13: User profile page, demonstrates the features available to the user regarding their NFT interactions

6.3 Conclusion

6.3.1 Reflecting on the Internship Journey

As this thesis reaches its conclusion, it's crucial to reflect on the journey that led to the development of our dApp, a fusion of blockchain technology and creative expression. The internship period was marked by rigorous learning, innovative problem-solving, and the successful integration of NEAR protocol smart contracts with a dynamic, user-centric frontend. Our endeavor to streamline the NFT creation and management process has not only been a technical challenge but also an insightful experience into the evolving world of blockchain technology.

6.3.2 Future Updates and Enhancements

Looking ahead, our team is committed to continual improvement and innovation. Key areas we aim to focus on in upcoming updates include:

1. **Migration Method for Smart Contracts:** We will develop robust methods for smart contract migration, ensuring our dApp remains adaptable and up-to-date with blockchain advancements.

2. **Dynamic NFTs:** Our plan includes designing dynamic NFTs (dNFTs) that represent real-time weather conditions, starting with Italian cities. This innovative approach will add a unique, interactive dimension to the NFTs.

3. **Advancing Content Moderation with NLP:** Our next phase involves improving content moderation by developing an advanced moderation endpoint with a custom Natural Language Processing (NLP) machine learning model. This model aims to more effectively filter out prohibited content, particularly excluding famous personalities' names and notable city monuments in image generation, to prevent copyright infringements and better adhere to content policies.

4. **Optimization:** Continuous efforts will be made to optimize the smart contract's efficiency and reduce the frontend's footprint, aiming for a leaner, more performant application.

5. **Enhancing User Experience:** We intend to revamp the frontend, focusing on intuitive navigation, aesthetic appeal, and responsive design, to elevate the overall user experience.

6.3.3 Encouragement and Community Engagement

As we embark on these exciting developments, we immensely value the blockchain community's support and feedback. The enthusiasm and insights from users and enthusiasts have been instrumental in shaping our roadmap. We invite everyone to continue their support and engagement, as it is the driving force behind our innovation and dedication to enhancing the NFT landscape.

6.3.4 Final Reflections

This internship has been a transformative journey, blending learning and application in the ever-evolving blockchain space. The insights gained and the progress made have set a strong foundation for future endeavors. As we look forward to the next chapter, our commitment to innovation, user engagement, and pushing the boundaries of blockchain technology remains stronger than ever. We are excited about the future of our dApp and its potential to continue evolving and inspiring the blockchain community.

Bibliography

[1] *NEAR protocol.* URL: `https://near.org/` (cit. on pp. 1, 15).

[2] *Nakamoto, S. (2008). Bitcoin: A Peer-to-Peer Electronic Cash System.* URL: `https://bitcoin.org/bitcoin.pdf` (cit. on pp. 1, 5).

[3] A. M. Antonopoulos. «Unlocking Digital Cryptocurrencies. O'Reilly Media». In: *Mastering Bitcoin.* 2014 (cit. on pp. 1, 6).

[4] Narayanan A. Bonneau J. Felten E. Miller A. Goldfeder S. «A Comprehensive Introduction». In: *Bitcoin and Cryptocurrency Technologies.* 2016 (cit. on p. 2).

[5] Casey M. J. Vigna P. «How Bitcoin and the Blockchain Are Challenging the Global Economic Order». In: *The Age of Cryptocurrency.* 2015 (cit. on p. 2).

[6] Tapscott A. Tapscott D. «How the Technology Behind Bitcoin Is Changing Money, Business, and the World». In: *Blockchain Revolution.* 2016 (cit. on p. 2).

[7] *NightShade.* URL: `https://pages.near.org/papers/nightshade/#nightshade/` (cit. on pp. 2, 22).

[8] *Proof-of-stake - Newer Consensus Mechanism.* URL: `https://www.investopedia.com/terms/p/proof-stake-pos.asp` (cit. on pp. 2, 14).

[9] *Rainbow Bridge.* URL: `https://rainbowbridge.app/about/` (cit. on p. 2).

[10] *Aurora Pass.* URL: `https://doc.aurora.dev/` (cit. on p. 2).

[11] *Octopus Network.* URL: `https://oct.network/` (cit. on p. 2).

[12] *ERC-20 Token Standard.* URL: `https://ethereum.org/en/developers/docs/standards/tokens/erc-20/` (cit. on p. 3).

[13] W.S. Stornetta S. Haber. «How to time-stamp a digital document». In: *Journal of Cryptology, vol 3, no 2, pages 99-111.* 1991 (cit. on pp. 5, 7).

[14] *Client-Server Architecture.* URL: `https://en.wikipedia.org/wiki/Client%E2%80%93server_model` (cit. on p. 7).

[15] *Distribution of global cloud.* URL: https://www.statista.com/statistics/292840/distribution-global-cloud-and-non-cloud-traffic/ (cit. on p. 7).

[16] Stuart Haber Dave Bayer and W. Scott Stornetta. «Improving the Efficiency and Reliability of Digital Time-Stamping». In: 1992 (cit. on p. 7).

[17] *Adam Back. (2002). Hashcash - A Denial of Service Counter-Measure.* URL: http://www.hashcash.org/papers/hashcash.pdf (cit. on p. 8).

[18] *Buterin, V. (2013). Ethereum White Paper: A Next-Generation Smart Contract & Decentralized Application Platform.* URL: https://ethereum.org/en/whitepaper (cit. on p. 8).

[19] *Transaction log.* URL: https://en.wikipedia.org/wiki/Transaction_log (cit. on p. 9).

[20] *Event Sourcing pattern.* URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing (cit. on pp. 9, 33).

[21] *Consensus Mechanisms in Blockchain.* URL: https://www.investopedia.com/terms/c/consensus-mechanism-cryptocurrency.asp (cit. on p. 9).

[22] *Know Your Customer.* URL: https://en.wikipedia.org/wiki/Know_your_customer (cit. on p. 11).

[23] *Initial Coin Offerings.* URL: https://www.investopedia.com/terms/i/initial-coin-offering-ico.asp (cit. on p. 14).

[24] *Proof-of-work - Original Consensus Mechanism.* URL: https://www.investopedia.com/terms/p/proof-work.asp (cit. on p. 14).

[25] *Solidity.* URL: https://docs.soliditylang.org/en/v0.8.12/ (cit. on p. 15).

[26] *Scaling Solutions.* URL: https://ethereum.org/en/developers/docs/scaling/ (cit. on p. 15).

[27] *Layer 2 Scaling.* URL: https://ethereum.org/en/developers/docs/scaling/#layer-2-scaling/ (cit. on p. 15).

[28] *Sidechains.* URL: https://ethereum.org/en/developers/docs/scaling/sidechains/ (cit. on p. 15).

[29] *Plasma chains.* URL: https://ethereum.org/en/developers/docs/scaling/plasma/ (cit. on p. 15).

[30] *Most Commonly used Programming Language.* URL: https://insights.stackoverflow.com/survey/2021#most-popular-technologies-language-prof (cit. on p. 16).

[31] *Rust is the most loved language.* URL: https://insights.stackoverflow.com/survey/2021#most-loved-dreaded-and-wanted-language-love-dread (cit. on p. 16).

[32] *NEAR Bitstamp Learn Center.* URL: https://www.bitstamp.net/learn/cryptocurrency-guide/what-is-near-protocol-near/ (cit. on p. 17).

[33] *NEAR Protocol documentation.* URL: https://docs.near.org/concepts/welcome (cit. on p. 17).

[34] *Account Model.* URL: https://docs.near.org/concepts/basics/accounts/model (cit. on p. 18).

[35] *Access Keys.* URL: https://docs.near.org/concepts/basics/accounts/access-keys (cit. on pp. 18, 27).

[36] *Transactions.* URL: https://docs.near.org/concepts/basics/transactions/overview (cit. on pp. 18, 59).

[37] *Staking on NEAR.* URL: https://near-nodes.io/validator/staking-and-delegation (cit. on p. 18).

[38] *WebAssembly.* URL: https://developer.mozilla.org/en-US/docs/WebAssembly (cit. on p. 19).

[39] *Node Types.* URL: https://near-nodes.io/intro/node-types#rpc-node (cit. on pp. 19, 21, 34).

[40] *NEAR Explorer Selector.* URL: https://explorer.near.org/nodes/validators (cit. on pp. 19, 21, 35).

[41] *Thresholded Proof Of Stake.* URL: https://near.org/blog/thresholded-proof-of-stake (cit. on p. 20).

[42] *Node Setup.* URL: https://docs.near.org/api/rpc/setup (cit. on pp. 21, 35, 71).

[43] *NEAR Protocol's Economics.* URL: https://near.org/blog/near-protocol-economics (cit. on pp. 21, 23).

[44] *Sharding Design: Nightshade.* URL: https://pages.near.org/papers/nightshade/ (cit. on p. 22).

[45] *Intro to NEAR Protocol's Economics.* URL: https://near.org/blog/near-protocol-economics (cit. on p. 22).

[46] *Gas.* URL: https://docs.near.org/concepts/basics/transactions/gas (cit. on p. 22).

[47] *Reducing a contract's size.* URL: https://docs.near.org/sdk/rust/contract-size (cit. on p. 23).

[48] *Storage Staking.* URL: https://docs.near.org/concepts/storage/storage-staking (cit. on p. 23).

[49] *NEAR Wallet.* URL: https://wallet.near.org/ (cit. on p. 26).

[50] *SDK for smart contracts.* URL: https://github.com/near/near-sdk-rs (cit. on p. 30).

[51] *Rust Collection Types.* URL: https://docs.near.org/concepts/storage/data-storage#rust-collection-types (cit. on p. 30).

[52] *Big-O Notation.* URL: https://docs.near.org/concepts/storage/data-storage#big-o-notation-1 (cit. on p. 30).

[53] *Indexer for Explorer.* URL: https://docs.near.org/tools/indexer-for-explorer (cit. on pp. 30, 74).

[54] *NEAR Indexer Framework.* URL: https://docs.near.org/concepts/advanced/near-indexer-framework (cit. on pp. 31, 77).

[55] *NEAR Lake Framework.* URL: https://docs.near.org/concepts/advanced/near-lake-framework (cit. on p. 31).

[56] *Events Format.* URL: https://nomicon.io/Standards/EventsFormat (cit. on pp. 33, 75).

[57] *The Graph Website.* URL: https://thegraph.com (cit. on pp. 33, 75).

[58] *Building Subgraphs on NEAR.* URL: https://thegraph.com/docs/en/cookbook/near/ (cit. on p. 33).

[59] *Constructing transactions on NEAR.* URL: https://github.com/near-examples/transaction-examples (cit. on p. 35).

[60] *NEAR CLI.* URL: https://docs.near.org/tools/near-cli (cit. on p. 35).

[61] *Creating a NEAR Wallet.* URL: https://wiki.near.org/overview/tokenomics/creating-a-near-wallet (cit. on p. 35).

[62] *Explorer Example.* URL: https://explorer.testnet.near.org/transactions/ABh4zQ5aZ3CGhpQzstL16TAB8TvqPbiirJG1uTPJVxTt (cit. on p. 35).

[63] *Transaction Receipt.* URL: https://docs.near.org/concepts/basics/transactions/overview#receipt-receipt (cit. on p. 36).

[64] *Deploy Testnet Smart Contract.* URL: https://docs.near.org/tools/near-cli#near-dev-deploy (cit. on p. 38).

[65] *Rapid Prototyping.* URL: https://docs.near.org/sdk/rust/building/prototyping (cit. on p. 38).

[66] *Migrating the State*. URL: https://docs.near.org/develop/upgrade#migrating-the-state (cit. on p. 38).

[67] *Programmatic Update*. URL: https://docs.near.org/develop/upgrade#programmatic-update (cit. on p. 38).

[68] *NEAR NFT Standard*. URL: https://nomicon.io/Standards/NonFungibleToken/ (cit. on p. 39).

[69] *Non-Fungible Tokens*. URL: https://en.wikipedia.org/wiki/Non-fungible_token (cit. on p. 42).

[70] *Module* $near_contract_standardsforFTs$. URL: https://docs.rs/near-contract-standards/latest/near_contract_standards/fungible_token/index.html (cit. on pp. 44, 45).

[71] *Fungible Token Core*. URL: https://nomicon.io/Standards/FungibleToken/Core (cit. on p. 45).

[72] *Decentralized Autonomous Organizations*. URL: https://near.org/use-cases/dao/ (cit. on p. 45).

[73] *Decentralized Finance*. URL: https://www.investopedia.com/decentralized-finance-defi-5113835 (cit. on p. 45).

[74] *Decentralized Exchanges*. URL: https://en.wikipedia.org/wiki/Decentralized_exchange (cit. on p. 45).

[75] *Liquidity Pools*. URL: https://academy.binance.com/en/articles/what-are-liquidity-pools-in-defi (cit. on p. 45).

[76] *Stablecoin*. URL: https://en.wikipedia.org/wiki/Stablecoin (cit. on p. 45).

[77] *Blockchain game*. URL: https://en.wikipedia.org/wiki/Blockchain_game (cit. on p. 45).

[78] *Blockchain oracle*. URL: https://en.wikipedia.org/wiki/Blockchain_oracle (cit. on pp. 48, 61).

[79] *Epic Games v. Apple*. URL: https://en.wikipedia.org/wiki/Epic_Games_v._Apple (cit. on p. 49).

[80] *Replay Attack*. URL: https://en.wikipedia.org/wiki/Replay_attack (cit. on p. 52).

[81] *Cryptographic nonce*. URL: https://en.wikipedia.org/wiki/Cryptographic_nonce (cit. on p. 52).

[82] *Access Keys*. URL: https://docs.near.org/api/rpc/access-keys (cit. on p. 53).

[83] *Decentralized Storage Solutions.* URL: https://docs.near.org/concepts/storage/storage-solutions (cit. on p. 59).

[84] *CryptoKitties.* URL: https://www.cryptokitties.co/ (cit. on p. 63).

[85] *NFTs airdrop.* URL: https://www.investopedia.com/terms/a/airdrop-cryptocurrency.asp (cit. on p. 63).

[86] *Non-Fungible Tokens.* URL: https://nomicon.io/Standards/Tokens/NonFungibleToken/ (cit. on p. 64).

[87] $internal_{m}int_{w}ith_{r}efundmethod$. URL: https://docs.rs/near-contract-standards/latest/near_contract_standards/non_fungible_token/core/struct.NonFungibleToken.html#method.internal_mint (cit. on p. 64).

[88] *Custodial Wallets.* URL: https://www.coindesk.com/learn/custodial-wallets-vs-non-custodial-crypto-wallets/ (cit. on p. 65).

[89] *Implicit Accounts.* URL: https://docs.near.org/concepts/basics/accounts/account-id#implicit-accounts-implicit-accounts (cit. on p. 66).

[90] *NEAR Drops.* URL: https://near.org/blog/send-near-to-anyone-with-near-drops (cit. on p. 66).

[91] *Prepaid Gas.* URL: https://docs.near.org/concepts/basics/transactions/gas#what-about-prepaid-gas-what-about-prepaid-gas (cit. on p. 66).

[92] *Approval with cross-contract call.* URL: https://nomicon.io/Standards/Tokens/NonFungibleToken/ApprovalManagement#2-approval-with-cross-contract-call (cit. on p. 68).

[93] *Royalties and Payouts.* URL: https://nomicon.io/Standards/Tokens/NonFungibleToken/Payout (cit. on p. 68).

[94] *NEAR JavaScript API.* URL: https://docs.near.org/tools/near-api-js/quick-reference (cit. on p. 71).

[95] *NearClient.* URL: https://github.com/good1101/NearClientApi/tree/master/NearClient (cit. on p. 71).

[96] *near-api-swift.* URL: https://github.com/near/near-api-swift (cit. on p. 71).

[97] *near-api-py.* URL: https://github.com/near/near-api-py (cit. on p. 71).

[98] *near-api-unity.* URL: https://github.com/near/near-api-unity (cit. on p. 71).

[99] *Rust Crate near$_c$ontract$_s$tandards*. URL: `https://docs.rs/near-contract-standards/latest/near_contract_standards/index.html` (cit. on p. 72).

[100] *Mintbase, NEAR NFT Launchpad*. URL: `https://www.mintbase.xyz/` (cit. on p. 72).

[101] *Paras Social NFT Marketplace*. URL: `https://paras.id/` (cit. on p. 73).

[102] *IPFS Website*. URL: `https://ipfs.io/` (cit. on p. 73).

[103] *Arweawe Website*. URL: `https://www.arweave.org/` (cit. on p. 73).

[104] *Filecoin Website*. URL: `ipns://filecoin.io/` (cit. on p. 73).

[105] *NFT STORAGE Website*. URL: `https://nft.storage/` (cit. on p. 74).

[106] *NEAR Indexer*. URL: `https://github.com/near/nearcore/tree/master/chain/indexer` (cit. on p. 74).

[107] *Unit Testing*. URL: `https://docs.near.org/develop/testing/unit-test` (cit. on p. 75).

[108] *Setting Up Testing*. URL: `https://docs.near.org/develop/testing/introduction` (cit. on p. 75).

[109] *Integration Test*. URL: `https://docs.near.org/develop/testing/integration-test` (cit. on p. 75).

[110] *NEAR Mainnet RPC Status*. URL: `https://rpc.mainnet.near.org/status` (cit. on p. 76).

[111] *RPC Nodes*. URL: `https://near-nodes.io/rpc` (cit. on p. 76).

[112] *Hardware Requirements for RPC Node*. URL: `https://near-nodes.io/rpc/hardware-rpc` (cit. on p. 77).

[113] *Wei*. URL: `https://academy.binance.com/en/glossary/wei` (cit. on p. 82).

[114] *NEAR Testnet Wallet*. URL: `https://wallet.testnet.near.org/` (cit. on p. 83).

[115] *NEAR Testnet Minted Transaction*. URL: `https://explorer.testnet.near.org/transactions/2LeFqiUFXmqV8Sc8E9XW947RikCpcwjkdW4fVUNDpwYm/` (cit. on p. 89).