Marlon Frade

learn basic Web3 programming

vol.1

Unlock the power of code and shape the digital future.

"The most profound journeys often begin with a single, uncertain step, guided by the courage to embrace the unknown and the wisdom to learn along the way."

<hello world>

Contents

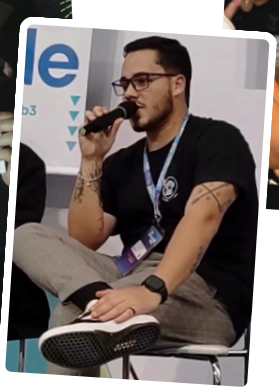
INTRODUCTION

Hi 👋, I'm Marlon Frade.

Hello, and welcome to the "Learn Programming with Web3" series. My name is Marlon Frade, and as a Software Developer, I work on cutting-edge projects for Web3 and create solutions using Artificial Intelligence. My passion for technology has led me to become a mentor in AI at an international hackathon, where I had the privilege of guiding participants from all over the world. By sharing my expertise, I've contributed to some of the most innovative Web3 projects. I firmly believe that through technology, we can build a future with more opportunities and fewer social and intellectual inequalities..

With the Web3 revolution and the increasing demand for skilled professionals in this area, I recognized the need for accessible and practical educational materials. My goal is to guide anyone—regardless of their experience level—on a journey to becoming a proficient programmer, capable of creating innovative solutions for the future of the internet.

This first volume introduces programming fundamentals, laying the groundwork for advanced topics like smart contract development and dApps. It's designed for both beginners and those with prior knowledge, offering a clear, progressive learning experience. Prepare to explore the potential of programming and the Web3 universe together!



CHAPTER 1

Programming Fundamentals

What is Programming?

At its core, programming is the process of giving instructions to a computer to perform specific tasks. These instructions, written in a programming language, form the backbone of every digital tool, application, and system you use, from your smartphone apps to the websites you browse daily. A program, which is a set of these instructions, tells the computer what to do and how to do it.

Computers are powerful but unintelligent machines—they can execute tasks rapidly and efficiently but can only follow the commands given by programmers. Programming, therefore, acts as the bridge between human thought and the computer's mechanical execution. It allows us to translate abstract concepts, ideas, or problems into a language the computer can understand and process.

How Programming Works?

When you write code, you're creating a set of instructions using specific syntax and rules, which vary depending on the programming language you're using. These instructions are then translated (or "compiled" or "interpreted") into binary code—a series of 1s and 0s—that the computer's hardware can read and execute.

Programming languages, like Python or JavaScript, are designed to be human-readable, so developers can focus on problem-solving rather than how the machine operates internally. Behind the scenes, however, the computer uses its central processing unit (CPU) to carry out each instruction, from basic arithmetic to more complex decision-making tasks.

The Purpose of Programming

Programming allows us to automate tasks, solve problems, and create systems that make our lives easier or more enjoyable. Whether it's developing a mobile app, creating a game, or automating data analysis, programming opens up endless possibilities. For businesses, programming is essential for improving efficiency, offering new services, and adapting to the digital age. For individuals, it can mean creating personal projects, contributing to open-source software, or even launching a tech startup.

The Importance of Learning to Code in the Web3 Era

As we move into the Web3 era, the importance of learning to code has never been greater. Web3 refers to the next generation of the internet, where decentralization, blockchain technology, and peer-to-peer interactions are reshaping how we interact with online platforms. Unlike Web1 (the static internet) or Web2 (the interactive, user-generated content era), Web3 aims to return control of the internet to users through decentralization, reducing the need for large intermediaries like Google, Facebook, or Amazon.

For anyone interested in contributing to this new internet era, understanding how to code is essential. Web3 development introduces complex but exciting new challenges, such as:

- **Blockchain Development:** Working with decentralized systems where data is spread across multiple nodes rather than centralized servers.

- **Smart Contracts:** Programs that run on the blockchain and can autonomously execute agreements when specific conditions are met.

- **Decentralized Applications (dApps):** Apps that run on decentralized networks and provide services without a centralized authority.

In this environment, learning to code equips you with the tools to develop decentralized apps, write secure smart contracts, and leverage blockchain for new innovations. The Web3 space is still in its early stages, so being able to code will allow you to shape the future of the internet—whether through cryptocurrencies, decentralized finance (DeFi), or new governance models powered by blockchain.

In addition, as industries transition to decentralized technologies, companies will require developers who can understand blockchain, smart contracts, and decentralized systems. Therefore, coding skills tailored for Web3 can position you at the forefront of a fast-growing job market, offering opportunities in everything from financial technology to gaming and social media.

Overview of Common Programming Languages: Python and JavaScript

When starting your programming journey, it's essential to understand the strengths of different programming languages. Two of the most popular and versatile languages today are Python and JavaScript. These languages play a crucial role in traditional software development, as well as in the emerging Web3 space.

Python: The Beginner-Friendly Powerhouse

Python is often praised for its simplicity and readability, making it one of the best languages for beginners to start with. Its syntax closely resembles the English language, which means you can focus more on learning programming concepts without getting bogged down by complex syntax.

Key Features:

- **Simple Syntax**: Python's clean, easy-to-read syntax helps new programmers understand and write code more easily compared to more complex languages like C++ or Java.
- **Versatility:** Python can be used for a wide range of applications, from web development to data science, machine learning, and automation. This flexibility makes it a valuable language to learn.
- **Large Community and Libraries:** Python boasts a massive global community of developers and an extensive ecosystem of libraries (pre-written code) that can simplify tasks, from building web apps to analyzing data.
- **Cross-Platform**: Python code can run on multiple operating systems (Windows, macOS, Linux) without modification, which is crucial for building cross-platform applications.

Python in Web3:

- Python can be used for blockchain development and building decentralized apps (dApps). Although Solidity is the primary language for writing smart contracts on Ethereum, Python plays a critical role in back-end development for Web3 applications and managing interactions with blockchain data.
- Python's libraries such as Web3.py allow developers to connect Python applications to the Ethereum blockchain, making it a handy tool for automating blockchain interactions.

JavaScript: The Language of the Web

JavaScript, on the other hand, is the backbone of modern web development. While Python may be more versatile across different fields, JavaScript dominates when it comes to building interactive websites and dynamic web applications.

Key Features:

- **Essential for Web Development:** Python's clean, easy-to-read syntax helps new programmers understand and write code more easily compared to more complex languages like C++ or Java.
- **Versatility:** Python can be used for a wide range of applications, from web development to data science, machine learning, and automation. This flexibility makes it a valuable language to learn.
- **Large Community and Libraries:** Python boasts a massive global community of developers and an extensive ecosystem of libraries (pre-written code) that can simplify tasks, from building web apps to analyzing data.
- **Cross-Platform**: Python code can run on multiple operating systems (Windows, macOS, Linux) without modification, which is crucial for building cross-platform applications.

CHAPTER 2

Core Web3 Programming

Why This Matters?

Before writing a single line of Web3 code, you must understand the core principles of programming.
In Web3, these principles are not just applied to websites or mobile apps — they are extended to smart contracts, decentralized applications (DApps), and blockchain interactions.

Think of this as your developer survival kit: without these basics, every advanced Web3 concept will feel twice as hard.

Data Types

Data types define what kind of data a variable can store.
Understanding them is crucial because blockchain operations often require strict type definitions (especially in Solidity).

Common Data Types in JavaScript/Python:

- Numbers: 42, 3.14
- Strings: "Hello Blockchain"
- Booleans: true / false
- Arrays / Lists: [1, 2, 3]
- Objects / Dictionaries: { name: "Alice", balance: 100 }

💡 In Solidity, data types are even more explicit: uint256, address, bool, string.

Variables

Variables are containers for storing data. Example in JavaScript:

```javascript
let balance = 100;
const walletAddress = "0x123abc...";
```

Example in Python:

```python
balance = 100
wallet_address = "0x123abc..."
```

Best Practices:

- Use const in JavaScript for values that won't change.
- Use descriptive names (userBalance instead of x).

Functions

Functions let you package reusable blocks of logic.

JavaScript example:

```javascript
function sendPayment(amount) {
  console.log(`Sending ${amount} tokens...`);
}
sendPayment(50);
```

Python example:

```python
def send_payment(amount):
    print(f"Sending {amount} tokens...")
send_payment(50)
```

In Web3, functions are also used in smart contracts to define how users and other contracts interact with the blockchain.

Control Flow

Control flow structures decide what code runs and when.

Conditionals:

```javascript
if (balance > 0) {
  console.log("You have funds!");
} else {
  console.log("Balance is empty.");
}
```

Loops:

```javascript
for (let i = 0; i < 5; i++) {
  console.log("Transaction", i);
}
```

In blockchain, you often loop through arrays of addresses, transactions, or events.

Objects & Structuring Data

Objects group related data together.

JavaScript example:

```javascript
let user = {
  name: "Alice",
  address: "0x123abc...",
  balance: 100
};
```

Python example:

```python
user = {
    "name": "Alice",
    "address": "0x123abc...",
    "balance": 100
}
```

On the blockchain, similar structures exist in smart contracts using structs.

Events & Logging

Logging is essential for debugging and monitoring.

Javascript:

```javascript
console.log("Transaction sent!");
```

Python:

```python
print("Transaction sent!")
```

In Solidity, events are the blockchain equivalent of logs, allowing external applications to track on-chain actions.

Putting It All Together – Mini Exercise

Let's write a simple script that:

- Stores a user's wallet address and balance
- Sends a "payment" if balance is enough

```javascript
let user = {
  address: "0x123abc...",
  balance: 150
};

function sendPayment(amount) {
  if (user.balance >= amount) {
    console.log(`Sending ${amount} tokens to ${user.address}`);
    user.balance -= amount;
  } else {
    console.log("Insufficient balance.");
  }
}

sendPayment(50);
console.log("Remaining balance:", user.balance);
```

CHAPTER 3

Introduction to Blockchain

From Web 2.0 to Web 3.0

The evolution of the internet has taken us from:

- Web 2.0 – Centralized applications controlled by companies like Facebook, Google, and Amazon.
- Web 3.0 – A decentralized web built on top of blockchain networks.

In Web3:

- Applications run across multiple computers (nodes) around the world.
- No single company owns the network.
- Data and logic are transparent and verifiable by anyone.

What is a Blockchain?

A blockchain is essentially a distributed ledger — a tamper-resistant record of transactions stored across many nodes.

Key points:

- Decentralized – Every node has an identical copy of the ledger.
- Consensus Mechanism – Nodes agree on the current state of the ledger through algorithms like Proof of Work or Proof of Stake.
- Transparency – Anyone can inspect the data.
- Immutability – Once data is recorded, it cannot be altered without consensus.

The idea existed before Bitcoin, but Bitcoin made it mainstream by implementing it for digital money.

Ethereum took it further by allowing developers to write programs (smart contracts) that run on the blockchain.

Ethereum & the EVM

Ethereum introduced the Ethereum Virtual Machine (EVM), which allows running smart contracts in a decentralized way.

- Each node runs the EVM.
- Smart contracts deployed to Ethereum run identically on all nodes.
- This enables DApps (Decentralized Applications) where the backend logic is public, verifiable, and distributed.

Smart Contracts

A smart contract is code stored and executed on the blockchain.

- Most Ethereum contracts are written in Solidity.
- Other blockchains may use different languages (e.g., Rust for Solana, Move for Aptos).
- Smart contracts define rules for transactions, store state, and emit events.

Frontend + Blockchain Interaction

A complete DApp usually has:

1. On-chain backend – Smart contracts on Ethereum (Solidity).
2. Off-chain frontend – A web app in JavaScript/TypeScript (React, Next.js).
3. Blockchain connection – Libraries like web3.js or ethers.js to communicate with the blockchain.

Why We Use Development Tools

Writing smart contracts directly on a live blockchain is slow, expensive, and risky.

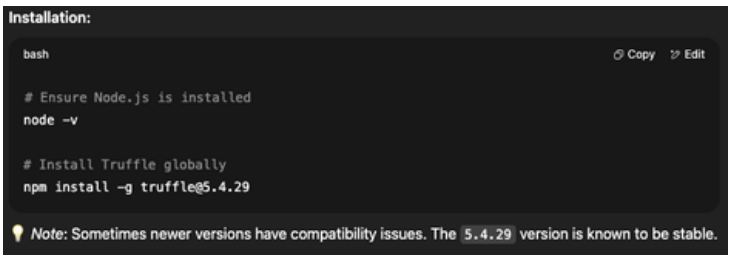
To make development easier, we use:

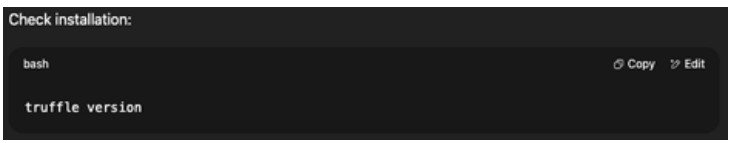
- Truffle – A development framework for compiling, testing, and deploying smart contracts.
- Ganache – A local blockchain for testing.
- Web3.js – A JavaScript library for interacting with the blockchain.

Installing Truffle

Truffle provides:

- Project scaffolding for smart contracts.
- Automated compilation and deployment scripts.
- Built-in integration with testing frameworks.





Installing Ganache

Ganache simulates a blockchain locally:

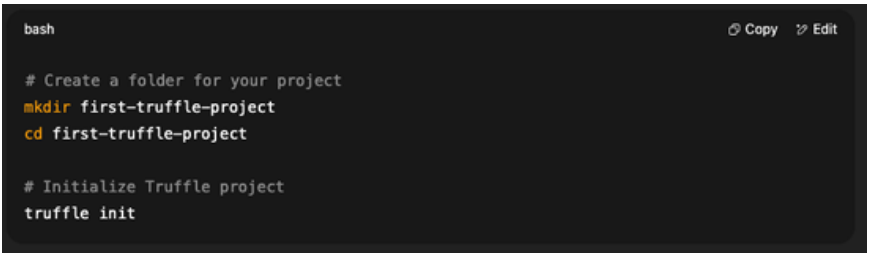
- Runs instantly without mining delays.
- Provides pre-funded accounts for testing.
- Lets you inspect blocks, transactions, and events.

Download:

- Go to https://trufflesuite.com/ganache/ and install for your OS.
- After opening Ganache:
  - You'll see 10 pre-funded accounts.
  - Each account has a private key.
  - You can track blocks, contracts, and transactions.

Creating Your First Truffle Project

With Truffle installed:



This creates:

- /contracts – For Solidity smart contracts.
- /migrations – For deployment scripts.
- /test – For testing contracts.
- truffle-config.js – Project configuration.

How It All Connects

Workflow overview:

1. Write contracts in Solidity inside /contracts.
2. Compile & deploy using Truffle.
3. Test contracts locally with Ganache.
4. Interact using JavaScript + Web3.js.
5. Move to testnet when ready (e.g., Sepolia or Mumbai).

CHAPTER 4

Understanding JSON-RPC

Why JSON-RPC Matters

When moving from Web 2.0 development to Web 3.0, one of the biggest shifts is how the frontend communicates with the backend.
In Web 2.0:

- Frontend sends HTTP requests to a centralized server.
- The server connects to a database and returns data.

In Web 3.0:

- The blockchain itself acts as the “database.”
- Your DApp communicates with nodes in the blockchain network.
- This communication happens via JSON-RPC — a remote procedure call protocol that uses JSON for encoding messages.

Blockchain Nodes and Communication

- A node is a computer running blockchain software.
- Every node stores the blockchain’s data and can process requests.
- Nodes communicate with each other using the same JSON-RPC protocol that we, as developers, use to talk to them.

Example:

- Bitcoin nodes store transactions like:

- Ethereum nodes store transactions and smart contract interactions.

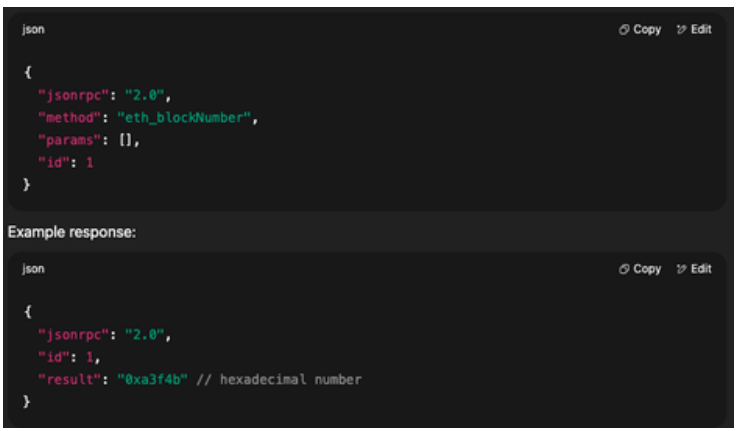
The JSON-RPC Protocol

- JSON: JavaScript Object Notation — a lightweight data format.
- RPC: Remote Procedure Call — a way to invoke functions or methods on another system.

JSON-RPC combines both:

- You send a JSON object specifying:
  - method: the blockchain function to call.
  - params: the arguments for that function.
  - id: a unique request identifier.
- The node responds with a JSON object containing the result.

Example request to get the latest block number:



Low-Level vs. High-Level Interaction

Direct JSON-RPC calls are low-level.
They are precise but require you to:

- Manually handle hex values.
- Sign transactions.
- Encode and decode smart contract data.

To simplify development, we use high-level libraries like:

- web3.js
- ethers.js
- web3.py

These libraries wrap JSON-RPC methods into easy-to-use JavaScript/Python functions.
Example with ethers.js:

```javascript
import { ethers } from "ethers";

const provider = new ethers.JsonRpcProvider("http://127.0.0.1:8545");

async function main() {
  const blockNumber = await provider.getBlockNumber();
  console.log("Latest Block:", blockNumber);
}

main();
```

Instead of manually sending JSON, we just call provider.getBlockNumber().

Example: Talking to Ganache via JSON-RPC

Ganache is a local blockchain node.
We can query it directly:
Request gas price:

```json
{
  "jsonrpc": "2.0",
  "method": "eth_gasPrice",
  "params": [],
  "id": 1
}
```

Sample Response:

```json
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": "0x09184e72a000" // gas price in wei
}
```

Example: Listing Accounts

```json
{
  "jsonrpc": "2.0",
  "method": "eth_accounts",
  "params": [],
  "id": 1
}
```

Response:

```json
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": [
    "0x627306090abaB3A6e1400e9345bC60c78a8BEf57",
    "0xf17f52151EbEF6C7334FAD080c5704D77216b732"
  ]
}
```

Example: Getting an Account Balance

Request:

```json
{
  "jsonrpc": "2.0",
  "method": "eth_getBalance",
  "params": [
    "0x627306090abaB3A6e1400e9345bC60c78a8BEf57", // account address
    "latest" // block number or 'latest'
  ],
  "id": 1
}
```

Response:

```json
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": "0x56bc75e2d63100000" // balance in wei
}
```

Convert from wei to ether using a library function or manual conversion:

- 1 ether = 10^{18} wei.

Real Network Example

We can do the same with a public node, such as:

- Ethereum mainnet via Infura
- Binance Smart Chain mainnet

Example BSC mainnet RPC endpoint:

Querying gas price or account balances here will give real network data.

Summary

- JSON-RPC is the core communication protocol for blockchain nodes.
- All high-level libraries simply wrap JSON-RPC calls.
- You can query local blockchains like Ganache or real networks like Ethereum Mainnet.
- For complex tasks (e.g., interacting with smart contracts), high-level libraries like web3.js or ethers.js are far more practical.

CHAPTER 5

Getting Started

The Development Stack Overview

In a real DApp workflow, your application communicates with the blockchain through nodes.
These nodes can be:

- Your own node (impractical for most developers due to size and sync requirements)
- Public RPC nodes (often with usage limits)
- Paid node services like Infura, Alchemy, Moralis, QuickNode.

For development, we'll use:

- Ganache – a local Ethereum blockchain for testing.
- Truffle – a framework for compiling, deploying, and testing smart contracts.
- Web3.js – a JavaScript library for interacting with Ethereum nodes.

Setting Up Ganache

- Open Ganache.
- Create a new workspace or quick-start blockchain.
- Ganache generates:
- 10 test accounts (each with 100 ETH by default in test environment).
- A local RPC endpoint (default: http://127.0.0.1:8545).
- Each account comes with a private key for testing.

Creating a Truffle Project

From your terminal:

```
mkdir truffle-ganache-demo
cd truffle-ganache-demo
truffle init
```

Truffle will create:

- contracts/ – store Solidity smart contracts.
- migrations/ – scripts to deploy contracts.
- test/ – smart contract tests.
- truffle-config.js – network and compiler settings.

Configuring Truffle to Connect to Ganache

Edit truffle-config.js:

```
module.exports = {
  networks: {
    development: {
      host: "127.0.0.1",
      port: 8545,
      network_id: "*" // Match any network id
    }
  },
  compilers: {
    solc: {
      version: "0.8.17" // Match Solidity version used in your contracts
    }
  }
};
```

Writing Your First Smart Contract

Create HelloWorld.sol in the contracts/ folder:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

contract HelloWorld {
    uint public myNumber = 10;

    function setMyNumber(uint newNumber) public {
        myNumber = newNumber;
    }
}
```

Creating a Migration Script

Create a new file in migrations/ named 2_deploy_hello_world.js:

```
const HelloWorld = artifacts.require("HelloWorld");

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

Compiling the Contract

```
truffle compile
```

This generates:

- ABI (Application Binary Interface) – describes how to interact with the contract.
- Bytecode – compiled code to be deployed to the blockchain.

Deploying to Ganache

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

You'll see:

- New blocks mined in Ganache.
- Gas costs deducted from the first account.
- Contract address where the contract is deployed.

Interacting with the Contract Using Truffle Console

Open the console:

```
truffle console --network development
```

Example interaction:

```
// Get deployed instance
let instance = await HelloWorld.deployed();

// Read the number
let num = await instance.myNumber();
num.toString(); // "10"

// Update the number
await instance.setMyNumber(42);

// Read again
let updated = await instance.myNumber();
updated.toString(); // "42"
```

All changes are recorded on the local blockchain inside Ganache.

Interacting Using Web3.js

Web3.js comes pre-bundled with Truffle, so you can use it directly in the console:

```
// Check if Ganache is mining
web3.eth.isMining()
```

Or in a separate JavaScript file:

```
const Web3 = require("web3");
const web3 = new Web3("http://127.0.0.1:8545");

(async () => {
  const accounts = await web3.eth.getAccounts();
  console.log("Accounts:", accounts);
})();
```

CHAPTER 6

Using Truffle, Ganache, and Web3.js Together

The Goal

In the previous chapters, we:

- Learned how to set up Truffle and Ganache.
- Wrote and deployed simple smart contracts.
- Interacted with contracts through the Truffle console.

Now we'll take the next step:

- Combine Truffle, Ganache, and Web3.js in a real workflow.
- Begin interacting with the blockchain directly from an HTML frontend.
- Understand the distinction between reading and writing blockchain data.

Ganache Refresher

- Ganache provides:
  - 10 pre-funded test accounts.
  - A mnemonic (seed phrase) that generates these accounts.
  - Full control over blocks, transactions, and events.

- Default RPC endpoint:

```
http://127.0.0.1:8545
```

Creating a New Project

In the terminal:

```
mkdir truffle-web3-demo
cd truffle-web3-demo
truffle init
```

This creates:

- contracts/ – Solidity contracts
- migrations/ – Deployment scripts
- test/ – Contract tests
- truffle-config.js – Network/compiler config

Update truffle-config.js:

```
development: {
  host: "127.0.0.1",
  port: 8545,
  network_id: "*"
}
```

Writing a Simple Contract

Example SimpleStorage.sol:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

contract SimpleStorage {
    string public name;

    function setName(string memory newName) public {
        name = newName;
    }
}
```

Deploying the Contract

In migrations/2_deploy_contracts.js:

```
const SimpleStorage = artifacts.require("SimpleStorage");

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

Compile and deploy:

```
truffle compile
truffle migrate --network development
```

Using the Truffle Console

From the console:

```
let instance = await SimpleStorage.deployed();
await instance.setName("Alice");
let currentName = await instance.name();
currentName; // "Alice"
```

Installing Web3.js

In your project directory:

```
npm init -y
npm install web3
```

Creating a Frontend with HTML + Web3.js

Create index.html in a src/ folder:

```
<!DOCTYPE html>
<html>
<head>
  <title>Web3.js + Ganache Demo</title>
  <script src="https://cdn.jsdelivr.net/npm/web3/dist/web3.min.js"></script
</head>
<body>
  <h1>Check Ganache Status</h1>
  <p id="status"></p>

  <script>
    // Connect to Ganache
    const web3 = new Web3("http://127.0.0.1:8545");

    async function checkMiningStatus() {
      const isMining = await web3.eth.isMining();
      document.getElementById("status").innerText = `Mining: ${isMining}`;
    }

    checkMiningStatus();
  </script>
</body>
</html>
```

Running the HTML

- Use the Live Server extension in VS Code (or similar) to open index.html.
- You should see Mining: true if Ganache is running.

Example: Getting the Latest Block Number

Add this to your <script>:

```
async function getBlockNumber() {
  const blockNumber = await web3.eth.getBlockNumber();
  console.log("Current Block:", blockNumber);
}
getBlockNumber();
```

CHAPTER 7

Creating and Managing Contract Instances

Smart Contracts as Classes

A good mental model for smart contracts:

- A contract = a class (in Object-Oriented Programming terms).
- A deployed contract = an instance of that class.
- Just like you can have multiple objects from the same class in a traditional program, you can have multiple deployed instances of the same contract on the blockchain.

What You Need to Interact with a Contract

To interact with an already deployed contract, you must have:

1. ABI (Application Binary Interface) – Describes the contract's functions, parameters, and return types.
2. It acts like a "public API" for the contract.
3. Contract Address – The unique blockchain address where that specific instance is deployed.

Without both, you can't fully interact with a deployed contract.

Example: Base Contract

Padrão.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

contract Padrão {
    string public nome;

    function mudaNome(string memory novoNome) public {
        nome = novoNome;
    }
}
```

- nome is a public state variable (readable without transactions).
- mudaNome() changes the value of nome (requires a transaction and gas).

Example: Contract That Creates Other Contracts

Criador.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

import "./Padrão.sol";

contract Criador {
    address public ultimoEnderecoCriado;

    function criaPadrão() public {
        Padrão novoContrato = new Padrão();
        ultimoEnderecoCriado = address(novoContrato);
    }
}
```

- *Criador* deploys new instances of the *Padrão* contract.
- Each call to *criaPadrão*() deploys a separate instance at a new address.

Deploying Both Contracts with Truffle

2_deploy_padrao.js

```
const Padrão = artifacts.require("Padrão");

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

3_deploy_criador.js

```
const Criador = artifacts.require("Criador");

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

Compile and migrate:

```
truffle compile
truffle migrate --network development
```

Interacting in the Truffle Console

```
// Get deployed instances
let padrao = await Padrão.deployed();
let criador = await Criador.deployed();

// Change the name in the first instance
await padrao.mudaNome("João");
(await padrao.nome()).toString(); // "João"

// Create a new instance via Criador
await criador.criaPadrão();
let novoEndereco = await criador.ultimoEnderecoCriado();
console.log("New instance address:", novoEndereco);

// Interact with the new instance
let padrao2 = await Padrão.at(novoEndereco);
(await padrao2.nome()).toString(); // Empty string

// Set a name for the second instance
await padrao2.mudaNome("Anna");
(await padrao2.nome()).toString(); // "Anna"
```

CHAPTER 8

Deploying Smart Contracts with Web3.js

Why a Low-Level Approach?

So far, we've been using Truffle to:

- Compile contracts.
- Deploy them to Ganache.
- Interact through the console or Web3.js with high-level abstractions.

However, Web3.js allows two different approaches:

1. High-Level: Using web3.eth.Contract to deploy and interact (recommended for most use cases).
2. Low-Level: Building and sending raw transactions manually, specifying bytecode, gas, and other parameters yourself.

In this chapter, we'll focus on the low-level way to deploy a contract, so you understand what's going on under the hood.

Requirements

- Ganache running on http://127.0.0.1:8545
- Compiled contract bytecode (from Truffle build folder)
- Node.js project with Web3.js installed:

```
npm init -y
npm install web3
```

Getting the Contract Bytecode

When Truffle compiles a contract, it generates a .json file in build/contracts/:

- ABI: Describes the contract interface.
- Bytecode: The compiled EVM code in hexadecimal format.

Example (inside Padrão.json):

```
"bytecode": "0x60806040..."
```

We'll use this bytecode to deploy manually.

HTML + Web3.js Setup

We'll use a simple HTML file with Web3.js included via CDN:

```
<!DOCTYPE html>
<html>
<head>
  <title>Deploy Contract with Web3.js</title>
  <script src="https://cdn.jsdelivr.net/npm/web3/dist/web3.min.js"></script>
</head>
<body>
  <button id="deployBtn">Deploy Contract</button>

  <script>
    const web3 = new Web3("http://127.0.0.1:8545");

    document.getElementById("deployBtn").addEventListener("click", async () =>
      const accounts = await web3.eth.getAccounts();

      // Load bytecode (in a real app, fetch from server or import JSON file)
      const bytecode = "0x6080604052348015600f57600080fd5b..."; // shortened

      // Build transaction object
      const txObject = {
        from: accounts[0],
        data: bytecode,
        gas: 500000 // temporary estimate
      };

      // Estimate gas accurately
      const gasEstimate = await web3.eth.estimateGas(txObject);
      txObject.gas = gasEstimate + 50000; // add buffer

      // Send transaction
      const receipt = await web3.eth.sendTransaction(txObject);
      console.log("Contract deployed at:", receipt.contractAddress);
    });
  </script>
</body>
</html>
```

Step-by-Step Breakdown

Connect to Ganache:

```
const web3 = new Web3("http://127.0.0.1:8545");
```

Get an account (must have funds in Ganache):

```
const accounts = await web3.eth.getAccounts();
```

Prepare transaction:

- from: account that pays for gas.
- data: compiled bytecode from the contract.
- gas: amount of gas to allow.

Estimate gas:

```
const gasEstimate = await web3.eth.estimateGas(txObject);
```

Send Transaction:

```
const receipt = await web3.eth.sendTransaction(txObject);
console.log(receipt.contractAddress);
```

Verifying Deployment

Once deployed, you can use Truffle or Web3.js to connect to the contract:

```
const Padrão = new web3.eth.Contract(abi, receipt.contractAddress);
```

Where:

- abi comes from the compiled JSON file.
- receipt.contractAddress is the new contract's address on Ganache.

CHAPTER 9

Interacting with Smart Contracts Using Web3.js

Once a smart contract is deployed to the blockchain, the next logical step is to interact with it. Interaction can take two main forms:

1. Reading data (no blockchain state change).
2. Writing data (changing the blockchain state).

In Web3.js, these correspond to:

- call → Used to read data from the blockchain without modifying it.
- sendTransaction → Used to send a transaction that changes the blockchain's state.

Call vs Transaction

Operation	Purpose	Costs Gas?	Requires `from` Address?
Call	Read-only interaction (no state change)	❌ No	❌ No
Transaction	State-changing interaction (write)	✅ Yes	✅ Yes

Reading Contract Data (call)

If your contract has a public variable or a view/pure function, you can read its value without spending gas.
For example, if you have:

```
string public name;
```

Solidity automatically generates a getter function named name().
How Web3.js Handles This at Low Level
Each function in a contract has a function signature, calculated as:

```
keccak256(functionName + parameterTypes)
```

Example:

```
"name()" → keccak256 → 0x2d… (first 4 bytes = function selector)
```

At low level, when calling name():

- We send a call request to the blockchain with:
  - The contract address (to)
  - The function selector as data
- The blockchain returns the encoded result, which we must decode from hexadecimal.

Example in Web3.js (Manual Low-Level Call):

```
const Web3 = require('web3');
const web3 = new Web3('http://127.0.0.1:7545'); // Ganache

const contractAddress = '0xYourContractAddressHere';
const functionSelector = '0x06fdde03'; // keccak256('name()').slice(0, 10)

const txData = {
  to: contractAddress,
  data: functionSelector
};

web3.eth.call(txData)
  .then(result => {
    console.log("Raw hex result:", result);
    console.log("Decoded string:", web3.utils.hexToUtf8(result));
  });
```

Writing to the Contract (sendTransaction)

When you change the state of a smart contract — for example, updating the name variable — you need to:

- Create a transaction
- Pay gas
- Specify a from account
- Provide enough gas limit

Low-Level Transaction
To call:

```
function changeName(string memory newName) public
```

We:

- Generate the function selector for changeName(string).
- Encode the string newName to hexadecimal (UTF-8 → hex).
- Pad data according to ABI encoding rules.
- Send a transaction with:
  - to = contract address
  - from = your account
  - gas = estimatedGas
  - data = encoded function selector + arguments

Example in Web3.js (Manual Low-Level Send):

```
const changeNameSelector = '0xA1B2C3D4'; // replace with actual keccak256('cha
const newName = web3.utils.utf8ToHex("Alice"); // convert to hex

const dataPayload = changeNameSelector + newName.slice(2).padEnd(64, '0');

const txObject = {
  from: '0xYourAccount',
  to: contractAddress,
  gas: 300000,
  data: dataPayload
};

web3.eth.sendTransaction(txObject)
  .then(receipt => {
    console.log("Transaction receipt:", receipt);
  });
```

Why Understand the Low Level?

Most developers use Web3.js contract abstractions:

```
const contract = new web3.eth.Contract(ABI, contractAddress);
await contract.methods.changeName("Alice").send({ from: account });
```

However, learning the low-level approach gives you:

- Deeper understanding of Ethereum transactions.
- Ability to debug when higher-level tools fail.
- Insight into ABI encoding/decoding.
- Skills useful for building custom blockchain tools.

CHAPTER 10

Simplifying Contract Interaction

In the previous chapter, we explored low-level smart contract interaction — manually encoding function selectors, arguments, and transaction data.
While this approach is valuable for understanding how Ethereum works under the hood, it is not practical for everyday development.
This is where web3.eth.Contract comes in.

What is web3.eth.Contract?

web3.eth.Contract is a high-level API provided by Web3.js that allows developers to:

- Load a smart contract using its ABI (Application Binary Interface) and address.
- Call contract methods without manually dealing with ABI encoding or decoding.
- Send transactions or perform read-only calls in a single line of code.

With web3.eth.Contract, the heavy lifting — such as function selector generation, argument encoding, and data decoding — is done automatically.

Loading a Contract Instance

To work with a contract, you need:

1. ABI – Defines the contract's methods, parameters, and data types.
2. Contract Address – Where the contract is deployed.

Example:

```
const Web3 = require('web3');
const web3 = new Web3('http://127.0.0.1:7545'); // Local Ganache instance

const contractABI = [ /* ABI JSON array here */ ];
const contractAddress = '0xYourContractAddressHere';

const myContract = new web3.eth.Contract(contractABI, contractAddress);
```

Reading Data with .call()

For read-only functions (marked view or pure in Solidity):

```
myContract.methods.name().call()
  .then(result => {
    console.log("Contract name:", result);
  });
```

What happens here:

- Web3.js encodes the request automatically.
- Sends it as a call to the Ethereum node.
- Decodes the response back into a readable format.

Writing Data with .send()

For state-changing functions (e.g., updating a name):

```
const account = '0xYourAccountAddress';

myContract.methods.changeName("Alice").send({ from: account, gas: 300000 })
  .then(receipt => {
    console.log("Transaction successful:", receipt);
  });
```

Key Points:

- .send() always requires:
  - from (the sender's address)
  - Sufficient gas
- It will create a real blockchain transaction.

Encoding Function Calls Without Sending

Sometimes you may only want to generate the encoded call data without sending a transaction — for example, for offline signing or debugging.

```
const encodedData = myContract.methods.changeName("Alice").encodeABI();
console.log("Encoded call data:", encodedData);
```

This returns the full ABI-encoded payload (function selector + arguments) exactly as Ethereum expects

Why Use web3.eth.Contract Instead of Manual Encoding?

- Faster development: No manual ABI encoding/decoding.
- Error reduction: Less chance of encoding mistakes.
- Easier to read: Your code focuses on business logic instead of low-level byte manipulation.
- Supports complex parameters: Arrays, structs, mappings — all handled automatically.

CHAPTER 11

Sending Native Currency to a Smart Contract

In Ethereum and other EVM-compatible blockchains, native currency refers to the built-in token used to pay for gas and transfer value:

- Ethereum Mainnet → Ether (ETH)
- BNB Chain → Binance Coin (BNB)
- Polygon → MATIC

Just like regular externally owned accounts (EOAs), smart contracts have their own addresses and can hold native currency.
The difference is:

- EOAs (your wallet) can initiate transactions by themselves.
- Smart contracts cannot initiate transactions; they can only execute logic when triggered by an external transaction.

Why Send ETH to a Smart Contract?

Some contracts require you to send ETH (or native currency) along with a function call:

- Crowdfunding or donation contracts.
- Pay-to-play games.
- Escrow or deposit systems.
- Auctions.
- Reward-based dApps.

If the contract is not designed to withdraw this ETH, it will remain locked forever.
⚠ Warning: Always verify that a contract includes withdrawal logic if funds need to be recoverable.

Example Contract

A simple Solidity contract that accepts ETH and stores the sender's address:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract PayToRegister {
    address public lastPayer;

    function receivePayment() public payable {
        require(msg.value > 0, "No ETH sent");
        lastPayer = msg.sender;
    }
}
```

Key points:

- The function receivePayment is marked payable.
- msg.value contains the amount of ETH sent.
- msg.sender contains the address of the sender.

Sending ETH from Web3.js

Using web3.eth.Contract, you can attach ETH to any .send() call via the value property.

Example:

```
const Web3 = require('web3');
const web3 = new Web3('http://127.0.0.1:7545'); // Ganache local node

const contractABI = [ /* ABI JSON here */ ];
const contractAddress = '0xYourContractAddress';

const account = '0xYourAccountAddress';

const myContract = new web3.eth.Contract(contractABI, contractAddress);

// Send 0.01 ETH to the contract
myContract.methods.receivePayment().send({
    from: account,
    value: web3.utils.toWei('0.01', 'ether'), // convert ETH to Wei
    gas: 300000
})
.then(receipt => {
    console.log("Transaction successful:", receipt);
})
.catch(err => {
    console.error("Error sending ETH:", err);
});
```

Important Details

1. Function Must Be payable
   - If the function isn't payable, sending ETH will cause the transaction to revert.
2. Always Use Wei for Transfers
   - Ethereum's smallest unit is Wei.
   - Use web3.utils.toWei('amount', 'ether') to convert.
3. Gas Considerations
   - You can omit gas and let MetaMask or the node estimate it, but setting a limit manually gives more control.
4. Receiving Without a Function
   - Contracts can have a receive or fallback function to handle ETH sent without specifying a function.

Common Mistake: Locked Funds

If a contract can receive ETH but has no function to send ETH back out, those funds are permanently stuck.
Always verify:

```
function withdraw() public {
    payable(owner).transfer(address(this).balance);
}
```

Without such logic, the contract becomes a black hole for funds.

CHAPTER 12

Installing and Configuring MetaMask

When developing dApps, you rarely interact with the blockchain directly from your code.
Instead, you use a wallet application — a piece of software that securely stores your private keys and manages transaction signing.
One of the most popular wallets for Ethereum and EVM-compatible blockchains is MetaMask.

What is MetaMask?

MetaMask is:

- A browser extension (available for Chrome, Firefox, Brave, Edge).
- A mobile app for Android/iOS.
- A non-custodial wallet — meaning you control your private keys.
- A bridge between your dApp and the blockchain, via injected APIs (window.ethereum).

It manages:

- Your accounts (addresses + private keys).
- Network connections (Ethereum mainnet, testnets, custom RPCs).
- Transaction signing — your dApp sends the data, MetaMask asks for your approval, and signs it locally.

Installing MetaMask (Browser Extension)

1. Go to the official Chrome Web Store (or the extension store for your browser).
2. Search for MetaMask and click Add to browser.
3. Pin it to your browser's toolbar for quick access.

⚠ Security Tip: Only install from the official source — https://metamask.io/ → Install MetaMask for Chrome/Firefox.

Creating a New Wallet

1. When you first open MetaMask:
2. Create a new wallet or import an existing one.
3. MetaMask will generate a Secret Recovery Phrase (also called seed phrase):
4. A sequence of 12 words.
5. This is the master key to all your accounts.
6. Store it offline, in a secure place.
7. Anyone with your seed phrase can access all your funds.
8. Set a local password — this only unlocks MetaMask on your device; it is not a backup method.

Accounts and Private Keys

- Each wallet can hold multiple accounts.
- All accounts are derived from the same seed phrase.
- Each account has:
  - A public address (used to receive funds).
  - A private key (used to sign transactions — never share it).

Example:

```
Seed phrase → Account 1 (Address A, Private Key A)
            → Account 2 (Address B, Private Key B)
```

Deleting an account locally doesn't remove it from the blockchain; it only removes it from MetaMask's view.

Connecting to Networks

MetaMask supports:

- Ethereum Mainnet.
- Testnets like Sepolia or Goerli.
- EVM-compatible chains like BNB Chain, Polygon, Avalanche, etc.

To add a custom network:

1. Go to Settings → Networks → Add network.
2. Fill in:
    - Network Name: e.g., BNB Chain Mainnet
    - RPC URL: Node endpoint (from chain's docs).
    - Chain ID: e.g., 56 for BNB Mainnet, 97 for Testnet.
    - Currency Symbol: e.g., BNB.
3. Save and switch to the new network.

Getting Test Funds (Faucets)

For development:

- Use testnets to avoid spending real funds.
- Request free test tokens from a faucet (search: "Sepolia faucet", "BNB Testnet faucet").
- Send the faucet your public address and receive tokens instantly.

Example:

- On BNB Testnet (chain ID 97), you can request 1 BNB from a faucet to test your contract interactions.

Key Security Practices

- Never share your seed phrase or private key.
- Use testnets for development until ready for mainnet deployment.
- Bookmark faucet and network RPC links to avoid phishing sites.

CHAPTER 13

Accounts, Wallets, and Connecting with MetaMask

When building Web3 applications, understanding accounts and how wallets like MetaMask manage them is critical.
In Ethereum and other EVM-compatible chains, an account is essentially:

- A public address — used to receive funds.
- A private key — used to sign transactions.

These always come in pairs, known as a key pair.

Hot vs Cold Wallets

- Hot Wallet: Connected to the internet (MetaMask, Trust Wallet, Coinbase Wallet).
  - More convenient but potentially less secure if your device is compromised.
- Cold Wallet: Offline storage (hardware wallets like Ledger, Trezor).
  - More secure for long-term holding but less convenient for daily interactions.

What is a Wallet?

A wallet is software (or hardware) that:

- Stores your private keys securely.
- Generates public addresses.
- Signs transactions without exposing your private key to the internet.

MetaMask is a hot wallet that runs as a browser extension (and also has a mobile version). It also injects a special JavaScript object, window.ethereum, into the browser — allowing your web app to interact with the blockchain through the wallet.

How Addresses Are Derived

- Your seed phrase (12 or 24 words) generates all your accounts deterministically.
- Each account = address + private key.
- Addresses are derived from the public key using cryptographic hashing:

```
public key → Keccak-256 hash → last 20 bytes → Ethereum address
```

Accessing Accounts with Web3.js

If you are connected to a local blockchain (e.g., Ganache), you can list accounts directly:

```
const Web3 = require('web3');
const web3 = new Web3('http://127.0.0.1:7545'); // Ganache
web3.eth.getAccounts().then(accounts => {
  console.log(accounts);
});
```

This works because Ganache automatically unlocks accounts for development.

Accessing Accounts from MetaMask

With MetaMask, things work differently:

- Accounts are locked by default.
- A dApp cannot access them until the user grants permission.

MetaMask injects window.ethereum into the browser:

```
if (window.ethereum) {
    console.log("MetaMask detected:", window.ethereum);
}
```

Security Notes

- Never store private keys in your application.
- Always let MetaMask handle signing transactions.
- Always verify the network before sending transactions (to avoid sending real funds when testing).

To request access to accounts:

```
async function connectWallet() {
    if (window.ethereum) {
        try {
            const accounts = await window.ethereum.request({
                method: 'eth_requestAccounts'
            });
            console.log("Connected account:", accounts[0]);
        } catch (err) {
            console.error("User rejected connection:", err);
        }
    } else {
        console.log("MetaMask not installed");
    }
}
```

You would typically attach this to a "Connect Wallet" button in your UI.

What Happens Behind the Scenes

1. Your site calls window.ethereum.request({ method: 'eth_requestAccounts' }).
2. MetaMask shows a popup asking the user to approve connection.
3. Upon approval, MetaMask shares the selected account's public address with your dApp.
4. Any transaction requests go through MetaMask for user confirmation.

Network Context

When connected, MetaMask also decides which blockchain node to use. The user's selected network in MetaMask could be:

- Ethereum Mainnet
- Polygon
- BNB Chain
- A local development chain (e.g., Ganache via custom RPC)

Your dApp should always check:

```
const chainId = await window.ethereum.request({ method: 'eth_chainId' });
console.log("Connected to chain:", chainId);
```

CHAPTER 14

Connecting to the Blockchain via MetaMask

In the previous chapter, we learned how to detect MetaMask and request the user's account. Now, we'll go one step further:
using MetaMask as a bridge between our dApp and the blockchain.

How MetaMask Connects to the Blockchain

MetaMask is not a blockchain node itself.
Instead:

- It manages your private keys and signs transactions locally.
- It connects to the blockchain through a node provider (like Infura, Alchemy, or your own custom node).
- Each "Network" you see in MetaMask is simply a configuration pointing to an RPC endpoint of a blockchain node.

Example:

- Ethereum Mainnet uses Infura's endpoint by default.
- You can add custom networks like Polygon, Avalanche, or a local Ganache RPC.

When your dApp sends a transaction via MetaMask:

- The transaction is prepared in your app.
- MetaMask signs it with your private key.
- MetaMask sends it to the configured RPC node.
- The node broadcasts it to the blockchain network.

Private Key, Public Key, and Address Recap

- Private Key → Keep it secret. It allows you to move funds and sign messages.
- Public Key → Derived from the private key via cryptography.
- Address → Derived from the public key (via hashing) and is shorter for convenience.

You can think of it like:

- Address = Your public mailbox location (you can share it).
- Private Key = The only key that opens the mailbox (never share it).

Given a private key, you can always recover the public key and address. This is why protecting the private key is critical.

Importing an Account into MetaMask

In development, you may want to use an account from Ganache inside MetaMask.Steps:

- Copy the private key from Ganache.
- In MetaMask, click Import Account.
- Paste the private key.
- The account will appear, and any test ETH in Ganache will be accessible.

Interacting with the Blockchain via MetaMask

When using Web3.js (or ethers.js) with MetaMask:

- Read-only operations (calls) do not require a signature.
- Write operations (transactions) require a signature in MetaMask.

1. Reading Data from a Contract

Example: Retrieving the name variable from a contract.

```
const contract = new web3.eth.Contract(ABI, contractAddress);

async function getName() {
    const name = await contract.methods.name().call();
    console.log("Name in contract:", name);
}
```

Because this does not modify the blockchain, MetaMask will not prompt the user.

2. Writing Data to a Contract
Example: Changing the name variable in a contract.

```
async function setName(newName) {
    const accounts = await window.ethereum.request({ method: 'eth_requestAccounts' });
    const from = accounts[0];

    await contract.methods.setName(newName).send({ from });
}
```

Here:

- MetaMask will open a popup asking the user to confirm.
- Once confirmed, the transaction will be signed and sent to the blockchain.
- The state will update once the transaction is mined.

Why This Matters

This pattern — connect wallet → read from blockchain → write to blockchain —
is the core workflow of any Web3 application.
You must:

1. Detect the wallet.
2. Request account access.
3. Connect to the blockchain via MetaMask's injected provider.
4. Call smart contract methods (read or write).

CHAPTER 15

MetaMask Methods and Events

When integrating MetaMask into a decentralized application, it's important to understand two things:

1. The ethereum object injected into the browser by MetaMask.
2. The events and methods it provides to handle user actions.

The ethereum Object

When MetaMask is installed, it injects a global object into the browser:

```
window.ethereum
```

This object:

- Exists only if MetaMask (or another wallet) is installed and active.
- Is not the same as Web3.js — it's a low-level RPC provider.
- Provides methods for sending requests to the wallet and blockchain.
- Emits events when certain actions happen (like account or network changes).

If MetaMask is not installed, window.ethereum will be undefined.

Example – Checking if MetaMask is installed:

```
if (typeof window.ethereum !== 'undefined') {
    console.log('MetaMask is installed!');
} else {
    alert('Please install MetaMask to use this application.');
}
```

Core Methods

The ethereum object provides methods to interact with the wallet.

Method	Description	Example
`ethereum.request()`	Sends a request to MetaMask.	`ethereum.request({ method: 'eth_requestAccounts' })`
`eth_requestAccounts`	Prompts user to connect their account.	—
`eth_accounts`	Gets the currently connected accounts.	—
`eth_chainId`	Returns the current network chain ID.	—

Example – Requesting the User's Account:

```
async function connectWallet() {
    const accounts = await ethereum.request({ method: 'eth_requestAccounts' })
    console.log("Connected account:", accounts[0]);
}
```

Core Events

MetaMask emits events when the user changes accounts or networks. Listening to these events allows your dApp to stay in sync.

Event	Trigger	Typical Usage
`accountsChanged`	User switches account in MetaMask.	Update UI, reload data.
`chainChanged`	User switches network.	Reload dApp or prompt to switch back.
`connect`	Connection to a network is established.	Initialize app state.
`disconnect`	Connection is lost.	Notify user.

Example – Listening to Account Changes:

```
ethereum.on('accountsChanged', (accounts) => {
    console.log('Account changed to:', accounts[0]);
    // Optional: reload data for new account
});
```

Example – Listening to Network Changes:

```
ethereum.on('chainChanged', (chainId) => {
    console.log('Network changed to:', chainId);
    window.location.reload(); // Ensure app reloads with correct network
});
```

Getting the Current Network

Every blockchain network has a Chain ID.

For example:

- Ethereum Mainnet → 0x1
- Ropsten Testnet → 0x3
- Local Ganache → 0x539 (1337 decimal)

Example – Detecting Network:

You can also force reload when the network changes, so the dApp reconnects properly.

When to Use Web3.js or Ethers.js

While window.ethereum lets you talk directly to MetaMask,
libraries like Web3.js and Ethers.js make interacting with contracts easier:

- Automatically handle ABI encoding/decoding.
- Provide helpers for unit conversion, event listening, and contract calls.
- Work with any EIP-1193 provider (including MetaMask).

In practice:

- Use ethereum for connection & event handling.
- Use Ethers.js or Web3.js for contract interaction.

CHAPTER 16

Integrating the Client, MetaMask, and the Blockchain

In the previous chapters, you learned how MetaMask works as a bridge between your decentralized application (dApp) and the blockchain. Now, we will connect everything together:

- Detect if MetaMask is installed
- Request wallet connection
- Read state variables from a smart contract
- Send transactions to change state variables

Core Concept

Every blockchain dApp must perform two main tasks with a contract:

1. Read state variables (no transaction required)
2. Write/update state variables (transaction required)

These two actions are the foundation for any interaction — from a simple name update to a complex DeFi protocol.

Step-by-Step Integration Flow

The integration between a frontend app, MetaMask, and the blockchain follows this algorithm:

Step 1 – Check if MetaMask is Installed

MetaMask injects the window.ethereum object.
If it's missing, you should alert the user.

Step 2 – Check if a Wallet is Already Connected

If MetaMask is installed, you can check for connected accounts:

Step 3 – Connect to MetaMask

If there are no connected accounts, prompt the user to connect:

Step 4 – Prepare Contract Access

To interact with a smart contract, you need:

- Contract ABI (defines functions and events)
- Contract address (where it's deployed)
- Web3.js or Ethers.js instance using MetaMask as provider

Example with Web3.js:

```
const web3 = new Web3(window.ethereum);

const contract = new web3.eth.Contract(CONTRACT_ABI,
CONTRACT_ADDRESS);
```

Step 5 – Reading State Variables

Example: Reading a name variable from the contract.

```
async function getName() {
    const name = await
contract.methods.getName().call();
    console.log("Name in contract:", name);
}
```

This does not require a transaction, so the user won't be asked to confirm anything.

Step 6 – Writing to the Contract

Example: Changing the name variable with a transaction.

```
async function setName(newName) {
    const accounts = await ethereum.request({ method: 'eth_accounts'
}); await contract.methods.setName(newName).send({ from: accounts[0]
}); console.log("Name updated successfully!");
}
```

Key points:

- A transaction requires gas fees (except on testnets with faucet tokens).
- MetaMask will show a confirmation popup to the user.

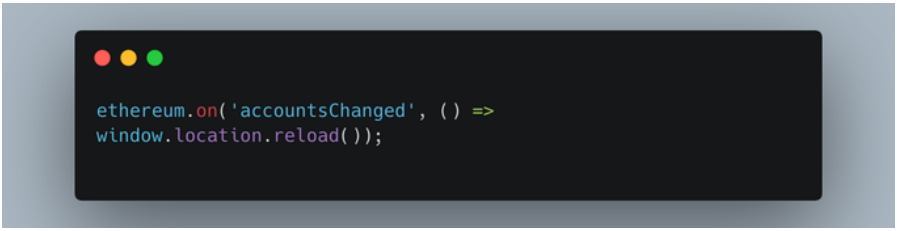
Putting It All Together

A minimal HTML + JavaScript example:

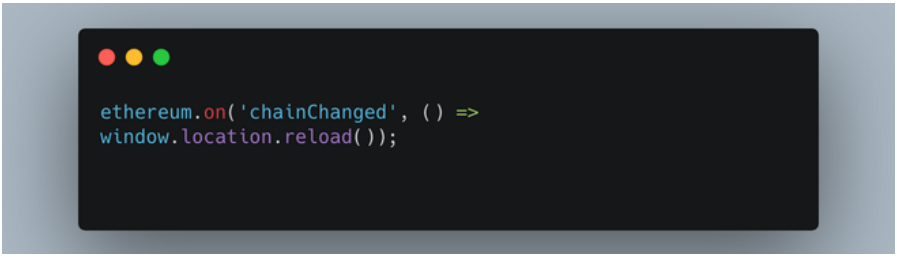
```
<input type="text" id="nameInput" placeholder="Enter new name" />
<button onclick="connectWallet()">Connect Wallet</button>
<button onclick="getName()">Get Name</button>
<button onclick="setName(document.getElementById('nameInput').value)">Set
Name</button>
<script src="https://cdn.jsdelivr.net/npm/web3@1.10.0/dist/web3.min.js"></script>
<script>
    let web3, contract;
    const CONTRACT_ABI = [...]; // Your ABI here
    const CONTRACT_ADDRESS = "0x123..."; // Your contract address

    async function connectWallet() {
        if (typeof window.ethereum === 'undefined') {
            alert("Install MetaMask first.");
            return;
        }
        web3 = new Web3(window.ethereum);
        await ethereum.request({ method: 'eth_requestAccounts' });
        contract = new web3.eth.Contract(CONTRACT_ABI, CONTRACT_ADDRESS);
        console.log("Connected to MetaMask");
    }

    async function getName() {
        const name = await contract.methods.name().call();
        console.log("Contract name:", name);
    }

    async function setName(newName) {
        const accounts = await ethereum.request({ method: 'eth_accounts' });
        await contract.methods.setName(newName).send({ from: accounts[0] });
        console.log("Name updated to:", newName);
    }
</script>
```

Best Practices

- Check the network before sending transactions to avoid wrong-chain errors.
- Handle account changes with:



- Handle network changes with:



Always validate inputs before sending transactions.

CHAPTER 17

Deploying Smart Contracts with Truffle Dashboard

In the previous chapters, we focused on integrating MetaMask and interacting with a smart contract locally.
Now, we will take the next step: deploying your smart contract to a real blockchain network using Truffle Dashboard.
This method avoids the need to manually manage private keys and provides a secure, user-friendly way to deploy directly from your MetaMask wallet.

What is Truffle Dashboard?

Truffle Dashboard is a feature introduced in recent versions of Truffle that:

- Allows deploying contracts directly through your browser.
- Uses MetaMask for transaction signing.
- Removes the need to store or expose private keys in your project.
- Works with any EVM-compatible network (Ethereum, BNB Smart Chain, Polygon, etc.) — both testnets and mainnets.

Before proceeding, make sure you have:

- MetaMask installed and configured.
- Truffle updated to the latest version.

Check if your Truffle version supports Dashboard:

If you see dashboard in the list, you are good to go.
If not:

```bash
npm uninstall -g truffle
npm install -g truffle
```

Starting the Dashboard

To launch the Truffle Dashboard:

- This will open a browser window at http://localhost:24012/.
- Click "Connect Wallet" and approve the MetaMask connection.
- Make sure MetaMask is connected to the correct network (e.g., BSC Testnet, Polygon Testnet, or Ethereum Mainnet).

Preparing the Deployment

In your Truffle project:

1. Go to the migrations/ folder.
2. Keep only the migration scripts you want to run (remove or comment out the rest to avoid deploying unwanted contracts).

Example: If you only want to deploy MyContract.sol, your 2_deploy_contracts.js should look like:

```
const MyContract =
artifacts.require("MyContract");
module.exports = function (deployer) {
  deployer.deploy(MyContract);
};
```

Deploying via Dashboard

Run:

```
truffle migrate --network
dashboard
```

- Truffle will compile your contracts.
- In your browser, MetaMask will prompt you to approve the deployment transaction.
- Once approved, Truffle will output the contract address.

Example output:

```
Deploying 'MyContract'
   > contract address:
0x121...talabodt:          0.0021 BNB
```

Verifying the Deployment

Once deployed, you can verify the contract on:

- Etherscan (Ethereum)
- BscScan (BNB Smart Chain)
- Polygonscan (Polygon)

Simply paste the contract address into the block explorer's search bar.

Updating Your Frontend

To make your dApp work with the newly deployed contract:

- Update the contract address in your frontend configuration.
- Keep the same ABI file if the contract code didn't change.
- Make sure your frontend checks if the user is on the correct network before allowing interactions.

Example – Network Check:

Best Practices

- Always deploy to a testnet first (like BSC Testnet or Goerli) before going live.
- Handle network mismatches gracefully — guide the user to switch networks.
- Monitor transaction status in your frontend to provide feedback while waiting for confirmation.
- Keep a record of deployed contract addresses for each environment.

CHAPTER 18

Testing Smart Contracts with Truffle

Once your smart contracts are deployed, they cannot be modified.
This immutability makes testing an essential part of blockchain development.
In traditional web applications, bugs can be fixed with a quick patch, and the existing database remains intact.
On the blockchain, a bug often means deploying an entirely new contract, migrating data (if possible), and asking users to interact with a different address.
Because of this, thorough testing before deployment is critical.

Why Testing Matters in Blockchain

- Immutability – Once deployed, the code cannot be changed.
- Security – Smart contracts handle valuable assets and need to prevent exploits.
- Cost – Deploying a new contract and migrating data can be expensive.
- Trust – Users trust contracts that have been well tested.

Testing Philosophy

- Two common testing approaches:
- Write the code, then write the tests – Traditional approach.
- Test-Driven Development (TDD) – Write the tests first, then implement code until the tests pass.
- In blockchain, a hybrid approach is common:
- Write tests for critical behaviors as you code.
- Focus on covering all important contract functions before deployment.

Truffle's Testing Framework

Truffle includes a built-in test runner using the Mocha framework with Chai assertions.
You can write tests in:

- Solidity – Good for certain on-chain logic verification.
- JavaScript – Most common, allows simulating interactions with contracts.

Project Setup for Testing

When you create a new Truffle project, it includes a /test directory.
That's where your test scripts go.

Example contract:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.11;

contract SimpleCoin {
    uint public value = 2000;
    mapping(address => uint) public balanceOf;

    constructor() {
        balanceOf[msg.sender] = value;
    }
}
```

Writing Your First Test

- Test file: /test/simplecoin.test.js

```
const SimpleCoin = artifacts.require("SimpleCoin");

contract("SimpleCoin", (accounts) => {
    it("should be deployed successfully", async () => {
        const instance = await SimpleCoin.deployed();
        assert(instance.address !== "", "Contract was not deployed");
    });
});
lanceOf[msg.sender] = value;
    }
}
```

How it works:

- artifacts.require loads the compiled contract.
- contract() defines a test suite for SimpleCoin.
- it() defines an individual test.
- assert() checks if the result matches expectations.

Running the Test

- Truffle compiles the contract.
- Deploys it to a local test blockchain (Ganache).
- Runs all tests in /test.

Testing Contract State

Example: Check the default value.

```
it("should have initial value of 2000", async () => {
    const instance = await SimpleCoin.deployed();
    const value = await instance.value();
    assert.equal(value.toNumber(), 2000, "Initial value is not 2000");
});
```

Testing Contract Behavior

Example: The deployer should receive an initial balance.

```
it("deployer should have balance of 2000", async () => {
    const instance = await SimpleCoin.deployed();
    const deployerBalance = await instance.balanceOf(accounts[0]);
    assert.equal(deployerBalance.toNumber(), 2000, "Deployer balance incorrect");
});
```

Test Execution Flow

When you run tests:

1. Ganache starts fresh with empty state.
2. Contracts are compiled and deployed.
3. Tests execute in isolation.
4. State resets between test runs.

This ensures consistent and repeatable results.

Best Practices for Smart Contract Testing

- Test all functions, including edge cases.
- Simulate errors and verify that the contract fails as expected.
- Test permission logic (e.g., only the owner can call certain functions).
- Use events to validate that certain actions occurred.
- Run tests frequently during development, not only at the end.

- MARLON FRADE