 # Near Blockchain Development Guide

Welcome to the comprehensive guide for developing on the Near blockchain. This guide will walk you through the entire process of setting up your environment, writing smart contracts, deploying them, and interacting with them using various tools.

Consolidated with ♡ by the NeuraNFT team.

Table of Contents

Documentation

- [NEAR Documentation](https://docs.near.org)
- [Rust Book](https://doc.rust-lang.org/book/)
- [Anaconda Documentation](https://docs.anaconda.com)

- [WSL Documentation](https://docs.microsoft.com/en-us/windows/wsl/)

- [Windows installation](https://docs.near.org/blog/getting-started-on-windows)

Useful Links

- [cargo-near](https://github.com/near/cargo-near) - NEAR smart contract development toolkit for Rust

- [near CLI](https://near.cli.rs) - Interact with NEAR blockchain from command line

- [NEAR Rust SDK Documentation](https://docs.near.org/sdk/rust/introduction)

- [NEAR Documentation](https://docs.near.org)

- [NEAR StackOverflow](https://stackoverflow.com/questions/tagged/nearprotocol)

- [NEAR Discord](https://near.chat)

- [NEAR Telegram Developers Community Group](https://t.me/neardev)

- NEAR DevHub: [Telegram](https://t.me/neardevhub), [Twitter](https://twitter.com/neardevhub)

 # NEAR Development Environment Setup Guide

This guide provides step-by-step instructions for setting up a NEAR development environment using WSL, Anaconda, Rust, and Node.js.

1. WSL and Anaconda Base Setup

Install WSL

```bash
# Open PowerShell as Administrator and run:
wsl --install
# Restart your computer
```

Basic WSL Dependencies

```bash
sudo apt update && sudo apt upgrade
```

sudo apt install -y build-essential curl wget git pkg-config libssl-dev
```

### Install Anaconda

```bash
# Download Anaconda
wget https://repo.anaconda.com/archive/Anaconda3-2024.02-Linux-x86_64.sh

# Install Anaconda
bash Anaconda3-2024.02-Linux-x86_64.sh
# Follow the prompts and accept the license agreement
# Say 'yes' to initializing conda

# Activate Anaconda
source ~/.bashrc

# Verify installation
conda --version
```

2. Rust and Node.js Setup

Install Rust

```bash
# Install Rust toolchain
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
# Choose option 1 for default installation

# Add Rust to current shell
source $HOME/.cargo/env

# Verify installation

rustc --version
cargo --version

# Add WebAssembly target
rustup target add wasm32-unknown-unknown
```

Install Node.js in Anaconda Environment

```bash
# Create a new conda environment for NEAR development
conda create -n near-dev python=3.12
conda activate near-dev

# Install Node.js through conda
conda install nodejs

# Verify Node.js installation
node --version
npm --version
```

3. NEAR Development Setup

Install NEAR CLI

```bash
# Install NEAR CLI globally
npm install -g near-cli

# Verify NEAR CLI installation
near --version
```
```

Useful Commands for Development

```bash
# Create new account
near create-account your-account.testnet --masterAccount testnet

# Delete account
near delete your-account.testnet your-master-account.testnet

# View account details
near state your-account.testnet

# View logs
near view-state your-account.testnet --finality final
```

Troubleshooting

Common issues and solutions:

1. If Rust commands aren't found:
```bash
source $HOME/.cargo/env
```

2. If conda commands aren't found:
```bash
source ~/.bashrc
```

3. If Node.js packages aren't found:
```bash
```

npm config set prefix '~/.npm-global'
export PATH=~/.npm-global/bin:$PATH
```

4. WSL filesystem issues:
```bash
# Run from Windows PowerShell:
wsl --shutdown
wsl
```

 # Rust Smart Contract Development

Setup NEAR Development Tools
```bash
# Install cargo-near for creating NEAR projects
cargo install cargo-near

# Create a new NEAR project
cargo near new your_project_name
cd your_project_name
```
```

```

## Project Structure

After creating a new project, you'll have the following structure:

```
your_project_name/
├── Cargo.toml
├── README.md
├── src/
│   └── lib.rs
└── target/
```

## Build and Test

```bash
# Build the contract
cargo build --target wasm32-unknown-unknown --release

# Run tests
cargo test
```

Environment Management

Remember to activate your conda environment before working:

```bash
conda activate near-dev
```

Useful Commands

```bash
# View NEAR account status
near state <account_id>
```
```

Deploy contract

near deploy --accountId <account_id> --wasmFile target/wasm32-unknown-unknown/release/<contract_name>.wasm

Call contract methods

near call <contract_id> <method_name> '{"param": "value"}' --accountId <account_id>

```



**# Setup NEAR Development Tools**

**## 1. Install Node.js and npm**

First, ensure you have Node.js (version 12 or higher) installed:
```

```bash
# Check Node.js version
node --version

# Check npm version
npm --version

# If you need to install Node.js, download from:
# https://nodejs.org/
```

2. Install NEAR CLI

```bash
# Install NEAR CLI globally
npm install -g near-cli

# Verify installation
near --version
```

Prerequisites

- Node.js 12 or higher
- NEAR CLI (`npm install -g near-cli`)
- A NEAR account (create one at wallet.near.org)

3. Create a new NEAR project

Option 1: Using create-near-app (Recommended for beginners)

```bash
# Install create-near-app globally
npm install -g create-near-app
```

```
# Create a new project
npx create-near-app

- make the selections as needed
```

```bash
# Navigate to project directory
cd your_project_name

# Install dependencies
npm install
```

Project Structure

After creating a new project, you'll have the following structure:

```
your_project_name/
├── package.json
├── README.md
├── src/
│   ├── index.js        # Main contract file
│   ├── metadata.js     # Metadata handling
│   ├── access.js       # Access control
│   └── utils/
│       ├── types.js    # Type definitions
│       └── helpers.js  # Helper functions
├── build/              # Compiled contract files
└── tests/              # Test files
```

Option 2: Manual Setup (For more control)

```bash
# Create project directory
mkdir your_project_name
cd your_project_name

# Initialize npm project
npm init -y

# Install necessary dependencies
npm install near-sdk-js near-api-js

# Install development dependencies
npm install --save-dev jest @babel/core @babel/preset-env
```

4. Project Configuration

Create a package.json with essential scripts:

```json
{
  "name": "your_project_name",
  "version": "1.0.0",
  "scripts": {
    "build": "near-sdk-js build src/index.js build/contract.wasm",
    "dev": "npm run build && npm run deploy",
    "deploy": "near deploy --accountId your-test-account.testnet --wasmFile
build/contract.wasm",
    "test": "jest"
  },
  "dependencies": {
    "near-sdk-js": "latest",
    "near-api-js": "latest"
  },
```

```
  "devDependencies": {
    "jest": "^27.0.0",
    "@babel/core": "^7.0.0",
    "@babel/preset-env": "^7.0.0"
  }
}
```

5. Create Basic Project Structure

```bash
# Create project directories
mkdir -p src/utils tests build

# Create initial files
touch src/index.js
touch src/config.js
touch src/utils/helpers.js
touch tests/main.test.js
```

6. Initialize NEAR Development Environment

```bash
# Login to NEAR testnet
near login

# Set environment to testnet
export NEAR_ENV=testnet

# Or set to mainnet when ready
export NEAR_ENV=mainnet

# Create a test account if needed
```

near create-account your-test-account.testnet --masterAccount testnet
```

## 7. Setup Basic Contract Structure

Create src/index.js:

```javascript
import { NearBindgen, near, call, view, initialize } from "near-sdk-js";

@NearBindgen({})
class Contract {
    constructor() {
        this.message = "Hello NEAR";
    }

    @initialize({})
    init() {
        this.message = "Contract initialized";
    }

    @view({})
    get_greeting() {
        return this.message;
    }

    @call({})
    set_greeting({ message }) {
        this.message = message;
    }
}

export default Contract;
```
```

```

## Additional Setup (Optional)

Create .gitignore:
```
node_modules/
build/
.env
.DS_Store
coverage/
```

Create .env for environment variables:
```
NEAR_ENV=testnet
NEAR_ACCOUNT_ID=your-test-account.testnet
NEAR_PRIVATE_KEY=your-private-key
```

### Contract Development Best Practices

#### File Organization
- Keep contract logic modular and separated into different files
- Use clear naming conventions for methods and variables
- Implement proper access control mechanisms
- Handle errors gracefully with try/catch

#### Testing
- Write unit tests for all contract methods
```

- Test edge cases and error conditions
- Use NEAR's simulation testing capabilities
- Test on testnet before mainnet deployment

Security Considerations
- Implement proper access control
- Validate all inputs
- Handle edge cases properly
- Use safe math operations
- Follow NEAR security best practices

Troubleshooting

Common issues and solutions:
1. **Gas errors**: Increase gas allocation for complex operations
2. **Account errors**: Ensure account exists and has sufficient balance
3. **Build errors**: Check Node.js version and dependencies
4. **Deployment failures**: Verify account permissions and contract size

Resources

- [NEAR Documentation](https://docs.near.org)
- [NEAR JavaScript SDK](https://github.com/near/near-sdk-js)
- [NEAR Examples](https://examples.near.org)
- [NEAR API Reference](https://docs.near.org/api/rpc/introduction)

Support

For additional support:
- Join NEAR Discord: https://near.chat
- Visit NEAR Forum: https://near.org/developers
- Stack Overflow: Tag questions with `nearprotocol`

 # NEAR Protocol JavaScript/TypeScript Smart Contract Development Guide

1. Project Structure Overview

A typical NEAR Protocol project structure follows this organization:

```
your_project_name/
├── package.json
├── README.md
├── src/
│   ├── contract1.js       # contract1
│   ├── contract2.js       # contract2
│   ├── contract3.ts       # contract3 in TS
│   └── utils/
│       ├── types.js       # Type definitions
│       └── helpers.js     # Helper functions
├── build/                 # Compiled contract files
├── tests/                 # Test files
|   ├── test.ava.js
|   └── test1.ava.js
└── package.json
```

Key Directories and Files:

- `src/`: Contains all source files for your smart contracts
- `build/`: Contains compiled WASM files
- `tests/`: Contains test files using AVA test framework

- `package.json`: Project configuration and scripts

2. Building and Testing Configuration

Basic package.json Configuration

package.json is the main configuration file for a Node.js project. It includes project metadata, dependencies, and scripts.

```json
{
  "scripts": {
    "build": "near-sdk-js build src/index.js build/Contract.wasm",
    "test": "ava tests/test.ava.js -- ./build/Contract.wasm"
  },
  "dependencies": {
    "near-sdk-js": "latest",
    "ava": "latest"
  }
}
```

```bash
# Terminal Code
# Build the contract
npm run build

# Deploy to testnet
npm run deploy

# Run tests
npm test
```

Standard Build Scripts

```json
{
  "scripts": {
    // Basic JavaScript build
    "build": "near-sdk-js build src/index.js build/Contract.wasm",

    // Basic test
    "test": "npm run build && ava tests/test.ava.js -- ./build/Contract.wasm",

    // TypeScript build
    "build:ts": "near-sdk-js build --generateABI src/contract.ts build/ContractTS.wasm",

    // TypeScript test
    "test:ts": "npm run build:ts && ava tests/test.ava.js -- build/ContractTS.wasm",

    // Optimized build
    "build:optimize": "near-sdk-js build --generateABI --optimizer src/optimized.js build/ContractOpt.wasm",

    // Test optimized build
    "test:optimize": "npm run build:optimize && ava tests/optimized.ava.js -- ./build/ContractOpt.wasm"
  }
}
```

Build Command Syntax Breakdown

Basic Build Command

```bash
near-sdk-js build <source-file> <output-file>
```

- `source-file`: Path to your contract's source file
- `output-file`: Where the compiled WASM will be saved

Example:

```bash
near-sdk-js build src/index.js build/Contract.wasm
```

Build Command Options

```bash
near-sdk-js build [options] <source-file> <output-file>
```

Common options:

- `--generateABI`: Generates an ABI file for the contract
- `--debug`: Includes debug information in the build
- `--optimizer`: Enables optimization for smaller, more efficient WASM
- `--noValidate`: Skips validation step (not recommended for production)

Example with options:

```bash
near-sdk-js build --generateABI --optimizer src/contract.js build/Contract.wasm
```

> Note as of 2024 Novmeber Near does not support ABI generation for Vanilla JS. We can only generate ABI for TypeScript and rust. For more information visit

https://github.com/near/abi

Running Build and Test Commands

Basic Commands

```bash
# Run basic build
npm run build

# Run basic test
npm run test

# Run TypeScript build
npm run build:ts

# Run TypeScript test
npm run test:ts
```

Using Arguments with npm run

Format:
```bash
npm run <script-name> -- [arguments]
```

Examples:
```bash
# Run test with specific WASM file
npm run test -- ./build/CustomContract.wasm
```

```
# Run build with custom output path
npm run build -- --out ./custom/path/contract.wasm
```

4. Advanced Script Configurations

Multiple Contract Builds

```json
{
  "scripts": {
    "build:all": "npm run build:contract1 && npm run build:contract2",
    "build:contract1": "near-sdk-js build src/contract1.js build/Contract1.wasm",
    "build:contract2": "near-sdk-js build src/contract2.js build/Contract2.wasm",
    "test:all": "npm run test:contract1 && npm run test:contract2",
    "test:contract1": "npm run build:contract1 && ava tests/contract1.ava.js -- ./build/Contract1.wasm",
    "test:contract2": "npm run build:contract2 && ava tests/contract2.ava.js -- ./build/Contract2.wasm"
  }
}
```

Environment-Specific Builds

```json
{
  "scripts": {
    "build:dev": "near-sdk-js build --debug src/index.js build/Contract-dev.wasm",
    "build:prod": "near-sdk-js build --optimizer src/index.js build/Contract-prod.wasm",
    "test:dev": "npm run build:dev && ava tests/test.ava.js -- ./build/Contract-dev.wasm",
    "test:prod": "npm run build:prod && ava tests/test.ava.js -- ./build/Contract-prod.wasm"
  }
}
```

Watch Mode for Development

```json
{
  "scripts": {
    "watch": "nodemon --watch src -e js,ts --exec 'npm run build'",
    "watch:test": "nodemon --watch src --watch tests -e js,ts --exec 'npm run test'"
  }
}
```

5. Common Script Patterns

Clean Build

```json
{
  "scripts": {
    "clean": "rm -rf build/*",
    "build:clean": "npm run clean && npm run build"
  }
}
```

Build with Type Checking

```json
{
  "scripts": {
    "type-check": "tsc --noEmit",
    "build:safe": "npm run type-check && npm run build:ts"
  }
}
```

Build with Multiple Environments

```json
{
  "scripts": {
    "build:testnet": "NEAR_ENV=testnet near-sdk-js build src/index.js build/Contract-testnet.wasm",
    "build:mainnet": "NEAR_ENV=mainnet near-sdk-js build src/index.js build/Contract-mainnet.wasm",
    "test:testnet": "NEAR_ENV=testnet npm run test",
    "test:mainnet": "NEAR_ENV=mainnet npm run test"
  }
}
```

6. Build Script Best Practices

1. **Use Descriptive Names**

   ```json
   {
     "scripts": {
       "build:contract": "near-sdk-js build src/contract.js build/Contract.wasm",
       "build:utils": "near-sdk-js build src/utils.js build/Utils.wasm"
     }
   }
   ```

2. **Include Clean Steps**

   ```json
   {
     "scripts": {
       "clean": "rm -rf build/*",
       "prebuild": "npm run clean",
   ```

```
    "build": "near-sdk-js build src/index.js build/Contract.wasm"
  }
}
```

3. **Add Validation**

```json
{
  "scripts": {
    "validate": "near-sdk-js validate",
    "build": "npm run validate && near-sdk-js build src/index.js build/Contract.wasm"
  }
}
```

4. **Environment Variables**

```json
{
  "scripts": {
    "build:dev": "NODE_ENV=development near-sdk-js build src/index.js build/Contract.wasm",
    "build:prod": "NODE_ENV=production near-sdk-js build src/index.js build/Contract.wasm"
  }
}
```

3. Smart Contract Syntax

Basic Contract Structure

```javascript
import { NearBindgen, near, call, view, initialize } from "near-sdk-js";

@NearBindgen({})
export class MyContract {
  constructor() {
    // Initialize contract state
  }

  @initialize({})
  init() {
    // Initialization logic
  }

  @view({})
  getState() {
    // View method
  }

  @call({})
  setState(newState) {
    // Call method
  }
}
```

Storage Examples

```javascript
```

```
import { NearBindgen, near, UnorderedMap } from "near-sdk-js";

@NearBindgen({})
export class StorageExample {
  // Single value storage
  value = 0;

  // Map storage
  map = new UnorderedMap('m');

  @call({})
  setValue(value) {
    this.value = value;
    this.map.set('key', value);
  }
}
```

4. Testing Framework

Basic Test File Structure

```javascript
import anyTest from 'ava';
import { Worker } from 'near-workspaces';

const test = anyTest;

test.beforeEach(async t => {
  // Setup test environment
  const worker = t.context.worker = await Worker.init();
  const root = worker.rootAccount;
```

```
  const contract = await root.createSubAccount('test-account');

  // Deploy contract
  await contract.deploy(process.argv[2]);
  await contract.call(contract, 'init', {});

  t.context.accounts = { root, contract };
});

test.afterEach.always(async t => {
  await t.context.worker.tearDown().catch(console.error);
});
```

Writing Tests

```javascript
// View method test
test('test view method', async t => {
  const { contract } = t.context.accounts;
  const result = await contract.view('getState', {});
  t.is(result, expectedValue);
});

// Call method test
test('test call method', async t => {
  const { contract } = t.context.accounts;
  await contract.call('setState', { newState: 'value' });
  const result = await contract.view('getState', {});
  t.is(result, 'value');
});
```

```
// Access control test
test('test access control', async t => {
  const { root, contract } = t.context.accounts;
  const user = await root.createSubAccount('user');

  // Test unauthorized access
  await t.throwsAsync(async () => {
    await user.call(contract, 'adminOnly', {});
  }, { instanceOf: Error });
});
```

Testing Helper Functions

```javascript
async function createTestUser(root, name, initialBalance) {
  return await root.createSubAccount(name, {
    initialBalance: initialBalance || '10N'
  });
}

async function assertThrows(t, promise, errorMsg) {
  await t.throwsAsync(async () => {
    await promise;
  }, { message: errorMsg });
}
```

5. Common Testing Patterns

Testing State Changes

```javascript
test('state changes correctly', async t => {
  const { contract } = t.context.accounts;

  // Initial state
  const initial = await contract.view('getState', {});
  t.is(initial, 0);

  // Change state
  await contract.call('setState', { value: 42 });

  // Verify state change
  const final = await contract.view('getState', {});
  t.is(final, 42);
});
```

Testing Events

```javascript
test('events are emitted correctly', async t => {
  const { contract } = t.context.accounts;

  const result = await contract.call('emitEvent', {
    message: 'test'
  });

  // Check logs
  t.deepEqual(result.logs, ['EVENT_EMITTED: test']);
});
```

Testing Access Control

```javascript
test('access control works', async t => {
  const { root, contract } = t.context.accounts;
  const user = await createTestUser(root, 'user');

  // Test authorized access
  await contract.call('grantAccess', { account: user.accountId });
  const hasAccess = await contract.view('checkAccess', {
    account: user.accountId
  });
  t.true(hasAccess);

  // Test unauthorized access
  const unauthorized = await createTestUser(root, 'unauthorized');
  const noAccess = await contract.view('checkAccess', {
    account: unauthorized.accountId
  });
  t.false(noAccess);
});
```

6. Best Practices

1. **Test Organization**
   - Group related tests together
   - Use descriptive test names
   - Keep tests focused and atomic

2. **Error Handling**
   - Test both success and failure cases

- Verify error messages
- Test edge cases

3. **State Management**
- Reset state between tests
- Use beforeEach and afterEach hooks
- Clean up resources properly

4. **Performance**
- Minimize contract calls in tests
- Use batch transactions when possible
- Test with realistic data sizes

7. Setup VS Code Configuration (Optional)

Create .vscode/settings.json:

```json
{
  "editor.formatOnSave": true,
  "javascript.suggestionActions.enabled": false,
  "javascript.validate.enable": false,
  "editor.codeActionsOnSave": {
    "source.fixAll.eslint": true
  }
}
```

This completes the setup for a NEAR JavaScript development environment. You now have:

- A fully configured NEAR development environment
- Basic contract structure

- Testing setup
- Deployment scripts
- Development tools integration

You can start developing your smart contract by modifying the src/index.js file and adding additional functionality as needed.

 # NEAR Protocol Account Creation and Deployment Guide

1. Account Creation

Creating TestNet Account

```bash
# Basic account creation using faucet
near create-account your-account.testnet --useFaucet

# Create account with initial balance
near create-account your-account.testnet --initialBalance 10 --useFaucet

# Create sub-account
near create-account sub-account.your-account.testnet --masterAccount your-account.testnet
```

Account Naming Rules

- Must be between 2-64 characters
- Can contain lowercase letters (a-z), digits (0-9), and hyphens (-)
- Cannot start or end with a hyphen
- TestNet accounts end with `.testnet`
- MainNet accounts end with `.near`

Key Management

After account creation, keys are stored at:

```bash
# TestNet keys
~/.near-credentials/testnet/your-account.testnet.json

# MainNet keys
~/.near-credentials/mainnet/your-account.near.json
```

Example key file content:

```json
{
  "account_id": "your-account.testnet",
  "public_key": "ed25519:XXXX...",
  "private_key": "ed25519:XXXX..."
}
```

2. Contract Deployment

Basic Deployment Commands

```bash
# Deploy to TestNet
near deploy --accountId your-account.testnet --wasmFile build/contract.wasm

# Deploy to MainNet
near deploy --accountId your-account.near --wasmFile build/contract.wasm

# Deploy with initialization
near deploy --accountId your-account.testnet \
  --wasmFile build/contract.wasm \
  --initFunction init \
  --initArgs '{"param1": "value1"}'
```

Environment-Specific Deployment

```bash
# Set environment
export NEAR_ENV=testnet  # or mainnet
```

```
# Deploy with specific network
near deploy --accountId your-account.testnet \
  --wasmFile build/contract.wasm \
  --networkId testnet
```

3. Contract Interaction

View Methods (Free, No Gas)

```bash
# Basic view call
near view your-account.testnet get_status

# View with parameters
near view your-account.testnet get_info '{"key": "value"}'

# View with multiple parameters
near view your-account.testnet get_data '{
  "user": "alice.testnet",
  "index": 1,
  "limit": 10
}'
```

Change Methods (Requires Gas)

```bash
# Basic call
near call your-account.testnet set_status '{"status": "active"}' \
  --accountId caller.testnet

# Call with attached deposit
near call your-account.testnet deposit '{"account": "bob.testnet"}' \
```

```
  --accountId caller.testnet \
  --deposit 10

# Call with specific gas
near call your-account.testnet complex_operation '{"data": "value"}' \
  --accountId caller.testnet \
  --gas 300000000000000
```

4. Account Management

Check Account State

```bash
# View account details
near state your-account.testnet

# View account balance
near view-account your-account.testnet

# View access keys
near keys your-account.testnet
```

Managing Keys

```bash
# Add access key
near add-key your-account.testnet ed25519:PUBLICKEY

# Delete access key
near delete-key your-account.testnet ed25519:PUBLICKEY

# Add function-call access key
```

```
near add-key your-account.testnet ed25519:PUBLICKEY \
  --contract-id contract.testnet \
  --method-names "method1,method2" \
  --allowance 10
```

5. Practical Examples

Complete Deployment Workflow

```bash
# 1. Create account
near create-account myapp.testnet --useFaucet

# 2. Build contract
npm run build

# 3. Deploy contract
near deploy --accountId myapp.testnet --wasmFile build/contract.wasm

# 4. Initialize contract
near call myapp.testnet init '{"owner": "myapp.testnet"}' --accountId myapp.testnet

# 5. Verify deployment
near view myapp.testnet get_status
```

NFT Contract Example

```bash
# Deploy NFT contract
near deploy --accountId nft.myapp.testnet \
  --wasmFile build/nft.wasm \
```

```
  --initFunction init \
  --initArgs '{
    "owner_id": "myapp.testnet",
    "metadata": {
      "spec": "nft-1.0.0",
      "name": "My NFT Collection",
      "symbol": "MNFT"
    }
  }'

# Mint NFT
near call nft.myapp.testnet nft_mint '{
  "token_id": "token-1",
  "metadata": {
    "title": "My First NFT",
    "description": "This is NFT #1",
    "media": "https://example.com/nft-1.png"
  },
  "receiver_id": "recipient.testnet"
}' --accountId myapp.testnet --deposit 0.1
```

Token Contract Example

```bash
# Deploy token contract
near deploy --accountId token.myapp.testnet \
  --wasmFile build/token.wasm \
  --initFunction init \
  --initArgs '{
    "owner_id": "myapp.testnet",
    "total_supply": "1000000000",
    "metadata": {
```

```
    "spec": "ft-1.0.0",
    "name": "My Token",
    "symbol": "MTK",
    "decimals": 18
  }
}'

# Transfer tokens
near call token.myapp.testnet ft_transfer '{
  "receiver_id": "user.testnet",
  "amount": "100000000"
}' --accountId myapp.testnet --deposit 0.000000000000000000000001
```

6. Troubleshooting

Common Issues and Solutions

1. ****Account Already Exists****
```bash
# Check if account exists
near state your-account.testnet

# Use different account name if exists
near create-account your-account-2.testnet --useFaucet
```

2. ****Insufficient Balance****
```bash
# Check account balance
near state your-account.testnet
```

Add funds using faucet (TestNet only)
near send faucet.testnet your-account.testnet 10
```

3. **Invalid Deployment**
```bash
# Verify WASM file
ls -l build/contract.wasm

# Rebuild and redeploy
npm run build
near deploy --accountId your-account.testnet --wasmFile build/contract.wasm --force
```

4. **Method Not Found**
```bash
# Check contract methods
near view your-account.testnet list_methods

# Verify method name and parameters
near view your-account.testnet get_method_names
```

Always remember to:
- Keep your key files secure
- Use TestNet for development and testing
- Verify contract code before MainNet deployment
- Monitor gas costs for change methods
- Back up your credentials
```