Contents

[Tutorial #2 – Registrar 2](#_Toc60683190)

[Pre-requisites 2](#_Toc60683191)

[Create a private blockchain 2](#_Toc60683192)

[Create a wallet 5](#_Toc60683193)

[Meet Alice and Bob 6](#_Toc60683194)

[Create a contract 8](#_Toc60683195)

[Domain registration interface 13](#_Toc60683196)

[Helper methods 13](#_Toc60683197)

[Domain name lookup 14](#_Toc60683198)

[Domain name registration 14](#_Toc60683199)

[Domain name transfer 15](#_Toc60683200)

[Domain name deletion 15](#_Toc60683201)

[Contract deployment 16](#_Toc60683202)

[Registering a domain 17](#_Toc60683203)

[Transferring a domain 23](#_Toc60683204)

[Deleting a domain 26](#_Toc60683205)

[Exercise for the reader 27](#_Toc60683206)

[Source code listing 27](#_Toc60683207)

# Tutorial #2 – Registrar

This tutorial will walk you through the process of creating real-world Neo Smart Contract using the Neo Blockchain toolkit. We will build a blockchain-based domain registration system.

This specific tutorial is designed for developers who prefer to work with their keyboards—most of the instructions will direct you to use the command-line tooling provided with the Neo Blockchain Toolkit. If you prefer to work with a graphical user interface, there is another version of this tutorial that is more UI-centric *[[insert link]]*.

This specific tutorial is designed for developers who prefer to work with a graphical user interface—most of the instructions will direct you to use functionality that is added to Visual Studio Code by the Neo Blockchain Toolkit. If you prefer to work with your keyboard, there is another version of this tutorial that is more command-line focused *[[insert link]]*.

## Pre-requisites

You will need the following software to follow along with this tutorial:

* .NET Core SDK *[[TODO: What version is required?]]*
  + We’ll write the smart contract code using the C# programming language. The .NET Core SDK is required to compile the C# code.
* Visual Studio Code
  + This is a very popular code editor; the Neo Blockchain Toolkit builds on top of VS Code.
* C# Visual Studio Code extension
  + This will add support to VS Code for editing and building C# source code.
* Neo Blockchain Toolkit Visual Studio Code extension)
  + This will add support to VS Code for visualizing Neo blockchains, running private blockchain instances and debugging Neo smart contracts.
* Neon smart contract compiler
  + This tool allows you to convert compiled C# code into bytecode suitable for execution by the Neo virtual machine.
* Neo Express
  + This tool allows you to run a private instance of the Neo blockchain.

All of the software listed above is freely available and cross-platform (you can follow along on Windows, Mac or Linux).

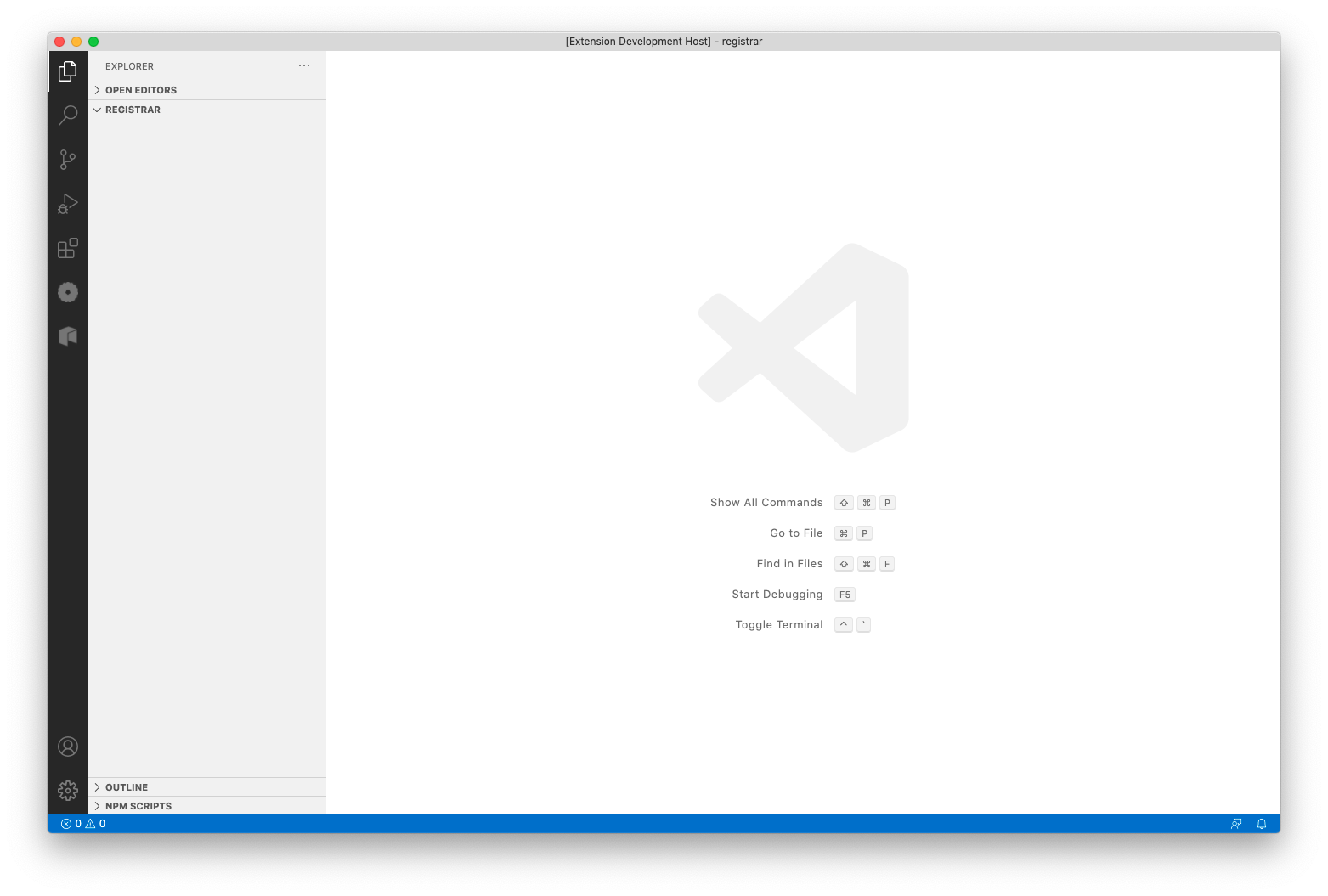
*[[TODO: Add or link to specific installation instructions for all of the above]]*

## Create a private blockchain

First, we’ll create a new empty folder—registrar—for our project. We’ll store all files related to our domain registration service in this folder.

$ mkdir registrar  
$ cd registrar

Load a new Visual Studio Code window, click on “Open Folder” and then use the folder selection dialog to create a new folder called registrar and then open that folder:



The first thing we will do is use Neo Express to create a private blockchain. This will allow us to deploy and invoke our contract while we are developing it without spending any real GAS.

$ nxp3 create  
Created 1 node privatenet at /Users/neo/xyz-token/default.neo-express  
 Note: The private keys for the accounts in this file are are \*not\* encrypted.  
 Do not use these accounts on MainNet or in any other system where security is a concern.

Take note of the security warning, a default.neo-express file will have been created that contains private keys, but those keys should only be used for local testing as they are not securely stored.

You can now run your private blockchain:

$ nxp3 run

In your terminal you’ll see the console output from Neo Express and will notice that about once every 15 seconds a new block is added to your blockchain. If you press Ctrl+C or close the terminal, you’re blockchain will stop. For now, leave Neo Express running and open a new terminal for the subsequent steps.

Click the Neo icon in the tool bar to open the Neo 3 Visual DevTracker:



Next, use the button in the Quick Start panel to create a new Neo Express Instance:

Graphical user interface, text, application

Description automatically generated

(Alternatively, you could select the “Create private blockchain” menu option from the context menu in the Blockchains panel.)

You’ll be asked how many consensus nodes that you want your private blockchain to have. For this example, one node is sufficient and will enable us to get the most out of Neo Express (some functionality—such as creating checkpoints—is disabled for multi-node blockchains).

When asked for a filename for the Neo Express configuration, we’ll use the name registrar.neo-express and save the file in the empty registrar folder.

Your screen should now look like this:

Graphical user interface, text, application, email

Description automatically generated

You can dismiss the message about the node being created (take note of the security warning, your registrar.neo-express file will contain private keys, but those keys should only be used for local testing as they are not securely stored). You can also close the Terminal panel showing Neo Express output if you wish—your blockchain will continue to run in the background and you’ll see new blocks appear in the Block Explorer panel about once every 15 seconds.

You can also check the “Hide empty blocks” checkbox so that only blocks containing transactions are shown. Initially you’ll only see the very first block but this will make it easier to identify our transactions later.

Graphical user interface, text, application

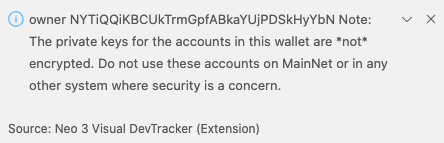
Description automatically generated

## Create a wallet

Next, we’ll create a wallet to use with our private blockchain. This wallet will be used to deploy our smart contract to the blockchain. Initially we’ll make domain registration free-of-charge so the owner won’t have any involvement after initial deployment (you could imagine us later improving the contract to charge fees—in NEO or GAS—for domain registration though and have the owner able to redeem those fees).

$ nxp3 wallet create owner  
owner  
 NYTiQQiKBCUkTrmGpfABkaYUjPDSkHyYbN  
 Note: The private keys for the accounts in this wallet are \*not\* encrypted.  
 Do not use these accounts on MainNet or in any other system where security is a concern.

Right click on registrar.neo-express in the Blockchains panel and click on the “Create wallet” menu option. When asked for a wallet name, type owner. You’ll see a message confirming that the wallet was created:



We now have a wallet for the smart contract owner, but that wallet doesn’t contain any assets. Deploying a smart contract to a Neo blockchain has a fee associated with it; the fee varies based on the size of the contract but is always paid in GAS.

Each Neo Express instance has a special wallet called “genesis” that is initially given the entire supply of NEO and GAS (the two assets native to the Neo blockchain). Let’s transfer some GAS from the genesis wallet to our owner wallet.

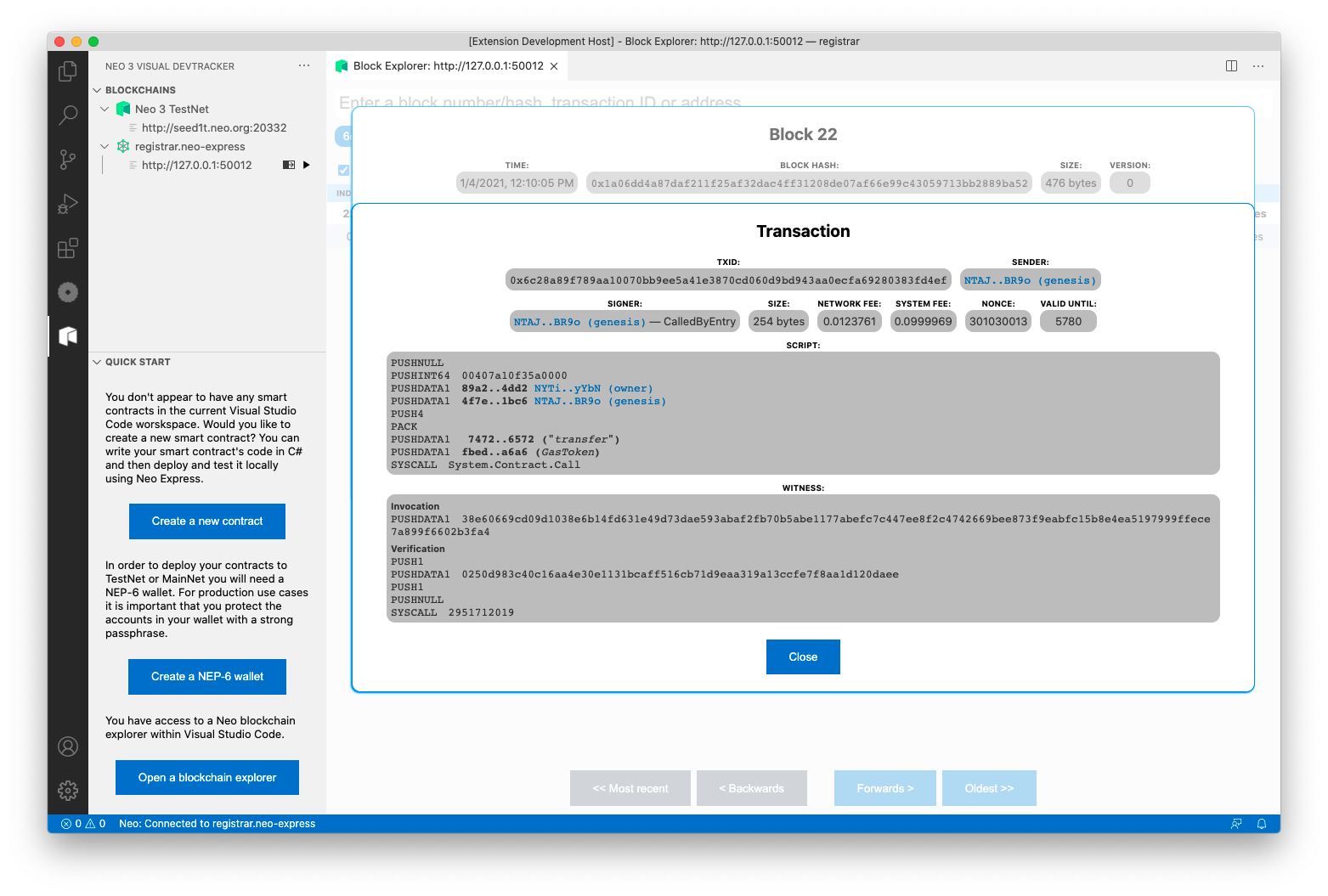
$ nxp3 transfer GAS 100000 genesis owner  
Transfer Transaction 0x0dac10eb08b1c92677f5dce20cccd0b235031a60fdba5781f6091272237510b7 submitted

Right click on registrar.neo-express in the Blockchains panel and click on the “Transfer assets” menu option. When prompted, select GAS as the asset. Enter 100,000 as the amount to transfer (this is more than enough to do multiple deployments of the contract we will later develop). Choose “genesis” as the source wallet and “owner” for the destination. You’ll see a message confirming that the transfer transaction was submitted:

Graphical user interface, text, application

Description automatically generated

Shortly after you’ll see a new non-empty block appear in your Block Explorer panel. You can click on that block to see a list of transactions in the block (there will only be one). You can click on the transaction to see the details.



## Meet Alice and Bob

Let’s create two more wallets so that we can later experiment with registering and transferring domains. We’ll call the wallets alice and bob (it is convention when describing protocols to name the first two participants Alice and Bob!)

The steps to create the wallets are exactly the same as above—when we created the owner wallet—just with different wallet names. Be sure to also transfer some GAS from the genesis wallet to Alice and Bob (as they will need some GAS to be able to invoke the registration contract that we will create).

The wallets that you have created are stored inside the .neo-express configuration file. If you open the file you should now see a wallets entry that looks something like this (your keys and addresses will be different, though):

A picture containing text

Description automatically generated

Your private blockchain should now have exactly three transactions—one for each of the transfers of GAS from genesis to owner, alice and bob:

Graphical user interface, text, application, email

Description automatically generated

## Create a contract

Now we’re ready to write the code for our smart contract.

We’ll use Visual Studio Code to write the code for our smart contract. Load a new VS Code window and open your registrar folder. The folder will contain a single file—default.neo-express—that was created by Neo Express to store your private blockchain configuration.

Click the Neo icon in the tool bar to open the Neo 3 Visual DevTracker:



Graphical user interface, application

Description automatically generated

Click the “Create a new contract” button in the Quick Start panel:

Graphical user interface, text, application, chat or text message

Description automatically generated

(Alternatively, you could select the “Create contract” menu option from the context menu in the Blockchains panel.)

When asked for the contract name, enter Registration. A new file called RegistrationContract.cs will be created and opened—this is our smart contract code. It has been pre-populated with some example code, but we’ll shortly remove and replace most of that…

Graphical user interface, text, application

Description automatically generated

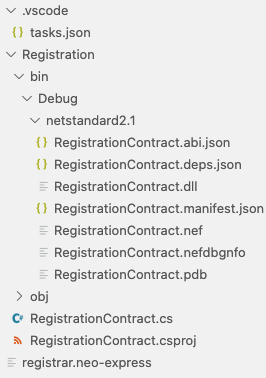
Graphical user interface, application

Description automatically generated

You can also see in the Explorer pane in VS Code that various other files have been created:

Text

Description automatically generated



The RegistrationContract.csproj file is an MS Build C# project configuration file; it tells the .NET Core SDK tooling how to build your project.

The tasks.json file is a Visual Studio Code configuration file that will allow you to build your code within Visual Studio Code. VS Code will have already built the sample code and the various files produced by the build are in the Registration/bin/debug/netstandard2.1 folder. You can rebuild your contract after making changes by choosing the “Run build task…” option in the “Terminal” menu in VS Code.

If you prefer to build using the command-line, you can delete the tasks.json file and instead run the dotnet build command within the Registration folder whenever you want to build:

$ cd Registration/  
$ dotnet build  
**Microsoft (R) Build Engine version 16.7.1+52cd83677 for .NET  
Copyright (C) Microsoft Corporation. All rights reserved.  
  
 Determining projects to restore...  
 All projects are up-to-date for restore.  
 RegistrationContract -> /Users/david/temp/cmd/registrar/Registration/bin/Debug/netstandard2.1/RegistrationContract.dll  
  
Build succeeded.  
 0 Warning(s)  
 0 Error(s)  
  
Time Elapsed 00:00:00.81**

Let’s remove the sample code and fill out some contract metadata, then we’ll be ready to write our own smart contract code…

The RegistrationContract.cs file contains a single class; it is called RegistrationContract and extends the SmartContract class (from the Neo.SmartContract.Framework package) to signify that it is a smart contract. The class has various attributes that are used to provide metadata that will be deployed to the Neo blockchain along with the contract:

[DisplayName("YourName.RegistrationContract")]  
[ManifestExtra("Author", "Your name")]  
[ManifestExtra("Email", "your@address.invalid")]  
[ManifestExtra("Description", "Describe your contract...")]

Let’s replace these with real values…

The DisplayName will be used to refer to your contract from within wallet software and other tools, it is common practice to provide a string consisting of an identifier for you (e.g. your GitHub ID, or company abbreviation) followed by a dot and then the contract name.

For the ManifestExtra attributes, replace the example values with real information. You can also remove the OnNumberChanged event, the MAP\_NAME constant and the ChangeNumber and GetNumber methods from the example contract:

using System;

using System.ComponentModel;

using System.Numerics;

using Neo.SmartContract.Framework;

using Neo.SmartContract.Framework.Services.Neo;

using Neo.SmartContract.Framework.Services.System;

namespace Registration

{

[DisplayName("djnicholson.RegistrationContract")]

[ManifestExtra("Author", "David Nicholson")]

[ManifestExtra("Email", "david@djntrading.com")]

[ManifestExtra("Description", "A domain registration service for Neo blockchains")]

public class RegistrationContract : SmartContract

{

}

}

You can rebuild your contract now to verify that it still builds. Our contract doesn’t do anything useful yet, though; next we’ll code various methods to make it a fully functional domain registration service!

## Domain registration interface

For this example, we’ll say that a valid domain name is any non-empty string consisting only of the characters ‘a’ through ‘z’.

We’ll support the following behaviors:

* Anyone can query whether a domain name is already registered (and if so, who owns it) using a Lookup method.
* Any available name can be registered by calling a Register method (signing the transaction as the intended owner of the domain).
* The existing owner of a domain can transfer it to someone else by invoking a Transfer method (providing the new owner as an argument and signing the transaction as the current owner).
* The existing owner of a domain can delete their registration by invoking a Delete method (signing the transaction as the current owner).

We’ll also emit an event whenever ownership changes for a domain name.

## Helper methods

First it would be useful to implement a couple of helper methods.

The first helper method will implement our validation logic; it will determine if an arbitrary string is a valid domain name according to our rules and throw an exception if not:

static void Validate(string domain)

{

var domainBytes = domain.ToByteArray();

for (int i = 0; i < domain.Length; i++)

{

if (domainBytes[i] < 'a' || domainBytes[i] > 'z')

{

throw new Exception("Domains must only use lowercase a-z characters");

}

}

if (domain.Length == 0)

{

throw new Exception("Domains must be non-empty");

}

}

We’ll often need to know the current owner of a valid domain name, so let’s also add a helper method for that. We’ll use contract storage to keep track of which domain is owned by which address and will arrange for the method to return zero if a domain is unregistered.

static Neo.UInt160 GetOwner(string domain)

{

byte[] value = Storage.Get(Storage.CurrentContext, domain);

if (value == null)

{

return Neo.UInt160.Zero;

}

else

{

return (Neo.UInt160)value;

}

}

We also need to declare the event that we will be emitting whenever domain name ownership changes:

[DisplayName("ChangeOwner")]

public static event Action<string, Neo.UInt160> OnChangeOwner;

## Domain name lookup

Our first operation will allow people to lookup the current owner of a domain name (a return value of zero will represent that the domain is currently unregistered):

public static Neo.UInt160 Lookup(string domain)

{

Validate(domain);

return GetOwner(domain);

}

Note that we confirm the domain name is valid before doing any further processing; we will follow the same pattern for all of our contract operations.

## Domain name registration

Next, we need an operation to allow someone to register an available domain name:

public static void Register(string domain)

{

Validate(domain);

if (!GetOwner(domain).IsZero)

{

throw new Exception("Already registered");

}

var tx = (Transaction) ExecutionEngine.ScriptContainer;

Storage.Put(Storage.CurrentContext, domain, (byte[]) tx.Sender);

OnChangeOwner(domain, tx.Sender);

}

Note that we first check that the domain is valid and available. We then extract the address used to sign the transaction and update the contract storage so the mapping from this domain name to this address is persisted.

## Domain name transfer

Now we need an operation for transferring domain names:

public static void Transfer(string domain, Neo.UInt160 to)

{

Validate(domain);

var owner = GetOwner(domain);

if (owner.IsZero)

{

throw new Exception("Not registered");

}

if (!to.IsValid || to.IsZero)

{

throw new Exception("Invalid transferee");

}

if (!Runtime.CheckWitness(owner))

{

throw new Exception("Not authorized");

}

Storage.Put(Storage.CurrentContext, domain, (byte[])to);

OnChangeOwner(domain, to);

}

We confirm that the domain is already registered, then we make sure that the destination address is valid and the signer of the transaction is the current owner of the domain name. If all these checks pass we update our contract storage and emit our ownership change event.

## Domain name deletion

Finally, we need an operation for domain name owners to delete their registration:

public static void Delete(string domain)

{

Validate(domain);

var owner = GetOwner(domain);

if (owner.IsZero)

{

throw new Exception("Not registered");

}

if (!Runtime.CheckWitness(owner))

{

throw new Exception("Not authorized");

}

Storage.Delete(Storage.CurrentContext, domain);

OnChangeOwner(domain, Neo.UInt160.Zero);

}

We check that the domain is currently registered and the person it is registered to has signed the transaction; we then remove the relevant item from storage and emit our ownership change event (using an address of zero to signify that the domain has become available again).

Now we’re ready to deploy our contract to our private Neo blockchain!

## Contract deployment

When you build your smart contract code, one of the files emitted is RegistrationContract.nef; this contains the Neo Virtual Machine bytecode for your contract and is required for deployment. You can deploy your contract to your private blockchain as follows:

$ nxp3 contract deploy Registration/bin/Debug/netstandard2.1/RegistrationContract.nef owner  
Deployment Transaction 0xe7443d7235775ca250d9271b676319be285679ffc9fab44b2a98825e8f492de7 submitted

Right click on registrar.neo-express in the Blockchains panel and click on the “Deploy contract” menu option. When asked which account to use, select the “owner” wallet that you created earlier. When asked which contract to deploy, select RegistrationContract.nef (this file contains the Neo Virtual Machine bytecode for your contract). You’ll see a message confirming that the deployment transaction was submitted:

Graphical user interface, text, application

Description automatically generated

Shortly after you’ll see a new non-empty block appear in your Block Explorer panel. You can click on that block to see a list of transactions in the block (there will only be one). You can click on the transaction to see the details.

Graphical user interface, text, application

Description automatically generated

You’ll notice that this transaction is somewhat larger than the transactions that we created earlier (when transferring GAS between accounts), that’s because this transaction contains the entire bytecode for your contract and all of its associated metadata! You can actually see the metadata in text format within the Block Explorer panel.

Your contract has now been deployed to your own private Neo blockchain. Next, we’ll experiment with registering some domains…

## Registering a domain

Neo Express allows you to invoke any contract deployed to your private blockchain. To do so, you must provide an “invoke file”; an invoke file is a JSON file that specifies one or more contract methods that should be invoked.

For more information and documentation on invoke files go to *[[insert link]]*.

Create a new text file and paste the following JSON into it:

[

{

"contract": "djnicholson.RegistrationContract",

"operation": "register",

"args": [ "widgets.neo", "@alice" ]

}

]

Save the file as alice-registration.neo-invoke.json and then run the following command:

$ nxp3 contract invoke alice-registration.neo-invoke.json alice  
Invocation Transaction 0xd47b9a9e729f10a75fa0466144728a85537074efdb62f2bc31a7fae205a28294 submitted

This command submits a transaction to your private blockchain that invokes all of the steps in the invoke file (only one step in our case) using Alice’s wallet.

Right click on registrar.neo-express in the Blockchains panel and click on the “Invoke contract” menu option. A new invoke file will be created for you and saved as invoke-files/Untitled.neo-invoke.json. By convention, invoke files use the .neo-invoke.json file extension, but you can rename the file to something more meaningful (e.g. alice-registration.neo-invoke.json) if you wish.

Graphical user interface, application, Teams

Description automatically generated

Invoke files can consist of multiple “steps”. The file created for you currently has one step, but all of the fields are currently empty. Let’s fill them out… Click into the first field and you will see a dropdown that lists all known contracts on your private blockchain:

Graphical user interface

Description automatically generated

Select your RegistrationContract, and then click into the “Operation” text box; you’ll see a list of all operations on your contract:

Graphical user interface, application

Description automatically generated

Select the “register” operation. You’ll notice that new text boxes appear for each of the arguments to the register method. Once filled out, your invoke file will look something like this:

Graphical user interface, application, Teams

Description automatically generated

Click the “Run this step” button to invoke your contract and when prompted choose Alice’s account. You’ll see a “Transactions” pane appear within the invoke file editor, this shows you the most recent transactions that you have submitted using this editor window and there will only be one transaction right now. The transaction will initially show as “pending” and then change to “ok” when your transaction is included in a block (within 15 seconds).

Graphical user interface, application, Teams

Description automatically generated

You can click on the transaction to see the same details that you would see if you found your transaction in the Block Explorer:

Graphical user interface, application

Description automatically generated

Congratulations, you just registered your first domain! widgets is now owned by Alice!

## Transferring a domain

Next let’s have Alice transfer the widgets domain to Bob.

Create a new invoke file called alice-to-bob-transfer.neo-invoke.json and populate it as follows:

[

{

"contract": "djnicholson.RegistrationContract",

"operation": "transfer",

"args": [ "widgets", "@bob" ]

}

]

Graphical user interface, application, Teams

Description automatically generated

Note that you can refer to wallet address in invoke files by prefixing the wallet name with an ‘@’ character.

Now run this invoke file the same was as before (again using Alice’s account to submit the transaction). Now Bob own’s the domain widgets!

Graphical user interface, application

Description automatically generated

You can verify this by trying to run the same invoke file again and verifying that the transaction results in an error (Alice is no longer the owner so our smart contract throws an exception).

Graphical user interface, application, Teams

Description automatically generated

## Deleting a domain

Finally, let’s delete the widgets domain.

Create a new invoke file called alice-to-bob-transfer.neo-invoke.json and populate it as follows:

[

{

"contract": "djnicholson.RegistrationContract",

"operation": "transfer",

"args": [ "widgets", "@bob" ]

}

]

Graphical user interface, application, Teams

Description automatically generated

Now run this invoke file the same was as before, but this time use Bob’s account to submit the transaction. Now nobody own’s the domain widgets and it is available for registration again!

## Exercise for the reader

Our contract allows anyone to register any domain free-of-charge (as long as they have enough GAS to pay to submit the invocation transactions). In a real-world you may want to charge fees when a user registers a domain; people could pay these fees in NEO, GAS or indeed any other NEP-17 asset.

As an exercise, you can modify the RegistrationContract to support this functionality: You can add an OnPayment method to your contract that will be called whenever someone pays assets to the contract. The OnPayment method provides the sender and amount of funds as arguments, you can determine what asset was paid by inspecting the ExecutionEngine.CallingScriptHash property provided by the runtime and you could make use of the optional data argument to allow the user to specify which name they would like to register. Within your OnPayment method you can reject the transaction—e.g., if the domain is unavailable or not enough funds were paid—by throwing an exception.

## Source code listing

Here is the complete smart contract source code:

using System;

using System.ComponentModel;

using System.Numerics;

using Neo.SmartContract.Framework;

using Neo.SmartContract.Framework.Services.Neo;

using Neo.SmartContract.Framework.Services.System;

namespace Registration

{

[DisplayName("djnicholson.RegistrationContract")]

[ManifestExtra("Author", "David Nicholson")]

[ManifestExtra("Email", "david@djntrading.com")]

[ManifestExtra("Description", "A domain registration service for Neo blockchains")]

public class RegistrationContract : SmartContract

{

static void Validate(string domain)

{

var domainBytes = domain.ToByteArray();

for (int i = 0; i < domain.Length; i++)

{

if (domainBytes[i] < 'a' || domainBytes[i] > 'z')

{

throw new Exception("Domains must only use lowercase a-z characters");

}

}

if (domain.Length == 0)

{

throw new Exception("Domains must be non-empty");

}

}

static Neo.UInt160 GetOwner(string domain)

{

byte[] value = Storage.Get(Storage.CurrentContext, domain);

if (value == null)

{

return Neo.UInt160.Zero;

}

else

{

return (Neo.UInt160) value;

}

}

[DisplayName("ChangeOwner")]

public static event Action<string, Neo.UInt160> OnChangeOwner;

public static Neo.UInt160 Lookup(string domain)

{

Validate(domain);

return GetOwner(domain);

}

public static void Register(string domain)

{

Validate(domain);

if (!GetOwner(domain).IsZero)

{

throw new Exception("Already registered");

}

var tx = (Transaction) ExecutionEngine.ScriptContainer;

Storage.Put(Storage.CurrentContext, domain, (byte[]) tx.Sender);

OnChangeOwner(domain, tx.Sender);

}

public static void Transfer(string domain, Neo.UInt160 to)

{

Validate(domain);

var owner = GetOwner(domain);

if (owner.IsZero)

{

throw new Exception("Not registered");

}

if (!to.IsValid || to.IsZero)

{

throw new Exception("Invalid transferee");

}

if (!Runtime.CheckWitness(owner))

{

throw new Exception("Not authorized");

}

Storage.Put(Storage.CurrentContext, domain, (byte[]) to);

OnChangeOwner(domain, to);

}

public static void Delete(string domain)

{

Validate (domain);

var owner = GetOwner(domain);

if (owner.IsZero)

{

throw new Exception("Not registered");

}

if (!Runtime.CheckWitness(owner))

{

throw new Exception("Not authorized");

}

Storage.Delete(Storage.CurrentContext, domain);

OnChangeOwner(domain, Neo.UInt160.Zero);

}

}

}