AElf

Release 0.6.0

Dec 03, 2020

Contents

CHAPTER 1

Overview

1.1 Common dependencies

This section is divided into two sub sections: the first concerns the common dependencies that are needed for running a node. The second shows the extra dependencies needed for building the sources and/or smart contract development.

1.1.1 Pre-setup for Windows users

A convenient tool for Windows users is **Chocolatey** for installing dependencies. Follow the installation instructions below (see here for more details **Chocolatey installation**):

Open and administrative Powershell and enter the following commands:

```
Set-ExecutionPolicy AllSigned
or
Set-ExecutionPolicy Bypass -Scope Process

Set-ExecutionPolicy Bypass -Scope Process -Force; iex ((New-Object System.Net.WebClient).
↪DownloadString('https://chocolatey.org/install.ps1'))
```

Later, **Chocolatey** can be very useful for installing dependencies on Windows systems.

1.1.2 Pre-setup for macOS users

It is highly recommended that you install **Homebrew (or simply Brew)** to quickly and easily setup dependencies (see here for more details **Homebrew install page**). Open a terminal and execute the following command:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/
↪install)"
```

1.1.3 Node js

Next install nodejs by following the instructions here (see here for more details **Nodejs**):

On macOS:

```
brew install node
```

On Windows:

```
choco install nodejs
```

On Linux:

```
sudo apt-get install nodejs
```

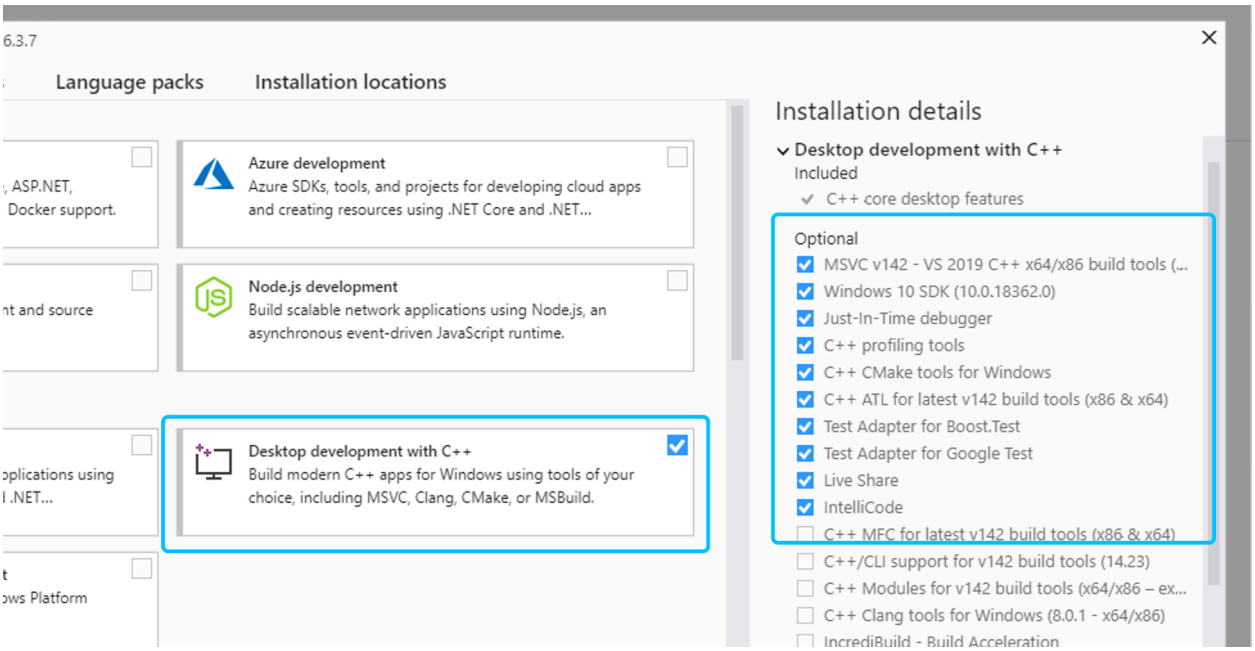
1.2 Building sources and development tools

You only need to follow this section if you intend to build **AElf** from the sources available on Github or if you plan on doing smart contract development.

1.2.1 Windows build tools

A dependency needed to build **AElf** from the command line under Windows is **Visual Studio Build Tools**. The easiest way is to use the **Visual Studio Installer**:

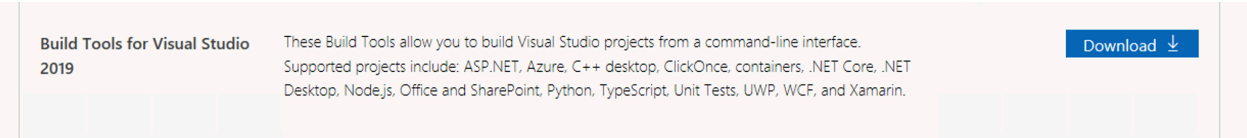
If you already have an edition of **Visual Studio** installed, open the **Visual Studio Installer** and add the **Desktop development with C++** workload:



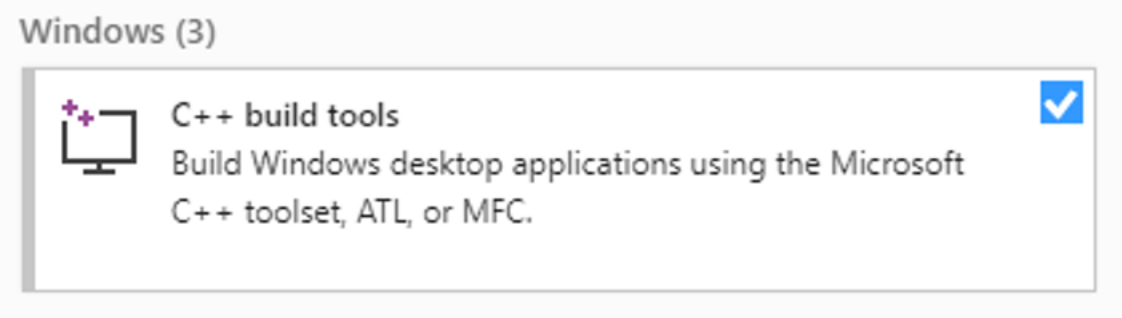
If you don't have any of the Visual Studio editions installed:

- you can download it here **Visual Studio Community Edition** for free and after the installation add the **Desktop development with C++** workload.

- or if you don't need or want a full blown installation of **Visual Studio**, you can download the build tools here: **Download Page**. Scroll down and under the section *Tools for Visual Studio 2019* download the build tools for Visual Studio:



After the installation open **Visual Studio Installer**, locate and install the *C++ build tools*.



1.2.2 Git

If you want to run a node or use our custom smart contract environment, at some point you will have to clone (download the source code) from **AElf** repository. For this you will have to use **Git** since we host our code on GitHub.

Click the following link to download Git for your platform (see here for more details **Getting Started - Installing Git**):

On macOS:

```
brew install git
```

On Windows:

```
choco install git
```

On Linux:

```
sudo apt install git-all
```

1.2.3 Development framework - dotnet core sdk

Most of **AElf** is developed with dotnet core, so you will need to download and install the .NET Core SDK before you start:

Download .NET Core 3.1

For now **AElf** depends on version 3.1 of the SDK, on the provided link find the download for your platform (for Windows and macOS the installer for x64 is the most convenient if your platform is compatible - most are these days), the page looks like this:

SDK 3.1.100

Visual Studio support
Visual Studio 2019 (v16.4)

Included in
Visual Studio 16.4.0

Included runtimes
.NET Core Runtime 3.1.0
ASP.NET Core Runtime 3.1.0
Desktop Runtime 3.1.0

Language support
C# 8.0
F# 4.7

OS	Installers	Binaries
Linux	Package manager instructions	ARM32 | ARM64 | x64 Alpine | x64 | RHEL 6 x64
macOS	x64	x64
Windows	x64 | x86	ARM32 | x64 | x86
All	dotnet-install scripts	

Wait for the download to finish, launch the installer and follow the instructions (for **AElf** all defaults provided in the installer should be correct).

To check the installation, you can open a terminal and run the `dotnet` command. If everything went fine it will show you `dotnet` options for the command line.

1.2.4 Protobuf

Depending on your platform, enter one of the following commands (see here for more details **Protobuf Github**):

On Windows, open a **Powershell** and enter the following commands:

```
choco install protoc --version=3.11.4 -y
choco upgrade unzip -y
```

On Linux:

```
# Make sure you grab the latest version
curl -OL https://github.com/google/protobuf/releases/download/v3.11.4/protoc-3.11.4-
↪linux-x86_64.zip

# Unzip
unzip protoc-3.11.4-linux-x86_64.zip -d protoc3

# Move protoc to /usr/local/bin/
sudo mv protoc3/bin/* /usr/local/bin/

# Move protoc3/include to /usr/local/include/
sudo mv protoc3/include/* /usr/local/include/

# Optional: change owner
sudo chown ${USER} /usr/local/bin/protoc
sudo chown -R ${USER} /usr/local/include/google
```

on macOS:

```
brew install protobuf@3.11
brew link --force --overwrite protobuf@3.11
```

1.3 Setup Boilerplate

1.3.1 Clone the repository

The following command will clone **Boilerplate** code into a folder, open a terminal and enter the following command:

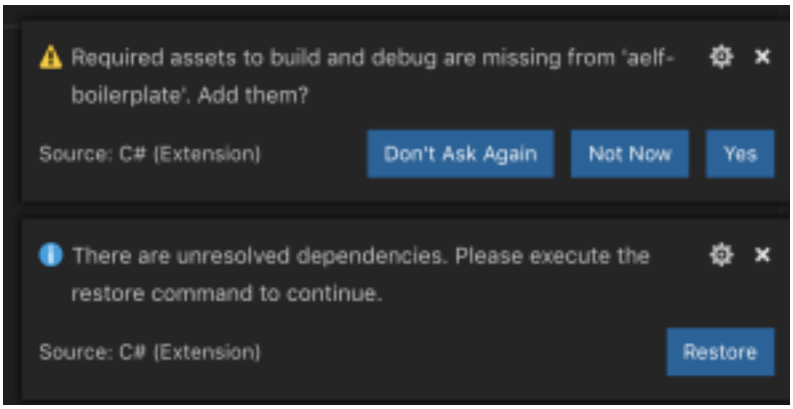
```
git clone https://github.com/AElfProject/aelf-boilerplate
```

The **boilerplate repo** contains a framework for easy smart contract development as well as examples (some explained in this series of articles).

1.3.2 Build and run

Open the project

If not already done, open vscode and open the **Boilerplate** folder. If asked to add some "required assets" say **yes**. There may also be some dependencies to restore: for all of them, choose **Restore**.



Open vscode's **Integrated Terminal** and build the project with the following command. Note: you can find out more about vscode's terminal **here**.

As stated earlier, Boilerplate takes care of the C# code generation and thus has a dependency on protobuf. If you don't already have it installed, you can refer to the guide for **manually install**.

Build and run

The next step is to build all the contracts in **Boilerplate** to ensure everything is working correctly. Once everything is built, we'll run as below

```
# enter the Launcher folder and build
cd chain/src/AElf.Boilerplate.Launcher/

# build
dotnet build

# run the node
dotnet run --no-build bin/Debug/netcoreapp3.1/AElf.Boilerplate.Launcher
```

When running **Boilerplate**, you might see some errors related to an incorrect password, to solve this, you need to backup your `data-dir/keys/` folder and start with an empty keys folder. Once you've cleaned the keys, stop and restart the node with the `dotnet run` command shown above.

At this point, the smart contracts have been deployed and are ready to be called (Boilerplate has a functioning API). You should see the node's logs in the terminal and see the node producing blocks. You can now stop the node by killing the process (usually **control-c** or **ctrl-c** in the terminal).

Run tests

Boilerplate makes it easy to write unit tests for your contracts. Here we'll take the tests of the Hello World contract included in Boilerplate as an example. To run the tests, navigate to the **AElf.Contracts.HelloWorldContract.Test** folder and run:

```
cd ../../test/AElf.Contracts.HelloWorldContract.Test/
dotnet test
```

The output should look somewhat like this, meaning that the tests have successfully executed:

```
Test Run Successful.
Total tests: 1
     Passed: 1
 Total time: 2.8865 Seconds
```

At this point, you have successfully downloaded, built, and run Boilerplate. You have also run the HelloWorld contract's tests that are included in Boilerplate. Later articles will show you how to add a contract and its tests and add it to the deployment process.

Try code generator

Code generation

Navigate to **AElf.Boilerplate.CodeGenerator** folder and open `appsettings.json`, modify `Content` node, tune `New` values as you wish.

For example, if you want to develop a `NovelWritingContract`.

```
    "Contents": [
      {
        "Origin": "AElf.Contracts.HelloWorldContract",
        "New": "Ean.Contracts.NovelWritingContract"
      },
      {
        "Origin": "HelloWorld",
        "New": "NovelWriting"
      },
      {
        "Origin": "hello_world",
        "New": "novel_writing"
      }
    ],
```

Run the code generator and then you will find a `AElf.Contracts.NovelWritingContract.sln` in `aelf-boilerplate\chain`, you can use this sln to develop your own smart contract.

```
# enter the Launcher folder and build
cd chain/src/AElf.Boilerplate.CodeGenerator/

# build
dotnet build

# run the node
dotnet run --no-build bin/Debug/netcoreapp3.1/AElf.Boilerplate.CodeGenerator
```

Single node contract deployment

With `AElf.Contracts.XXContract.sln`, you can run project `AElf.Boilerplate.XXContract.Launcher` which is newly generated via above step, the `XXContract` will be automatically deployed in the block of height 2.

Check following code in `AElf.Boilerplate.XXContract.Launcher/DeployContractsSystemTransactionGenerator.cs`:

```
public async Task<List<Transaction>> GenerateTransactionsAsync(Address @from, long␣
↪preBlockHeight,
    Hash preBlockHash)
{
    if (preBlockHeight == 1)
    {
        var code = ByteString.CopyFrom(GetContractCodes());
        return new List<Transaction>
        {
            await _transactionGeneratingService.GenerateTransactionAsync(
                ZeroSmartContractAddressNameProvider.Name, nameof(BasicContractZero.
↪DeploySmartContract),
                new ContractDeploymentInput
                {
                    Category = KernelConstants.DefaultRunnerCategory,
                    Code = code
                }.ToByteString())
        };
    }

    return new List<Transaction>();
}

private byte[] GetContractCodes()
{
    return ContractsDeployer.GetContractCodes<DeployContractsSystemTransactionGenerator>
↪(_contractOptions
        .GenesisContractDir)["AElf.Contracts.XXContract"];
}
```

You can customize code in `if` section to add more actions to deploy more contracts.

For example, you develop two smart contract using one generated sln: `XXContract` and `YYContract`, the deployment code should be like this:

```
public async Task<List<Transaction>> GenerateTransactionsAsync(Address @from, long␣
↪preBlockHeight,
    Hash preBlockHash)
{
    if (preBlockHeight == 1)
    {
        var xxCode = ByteString.CopyFrom(GetContractCodes("AElf.Contracts.XXContract"));
        var yyCode = ByteString.CopyFrom(GetContractCodes("AElf.Contracts.YYContract"));
        return new List<Transaction>
        {
            await _transactionGeneratingService.GenerateTransactionAsync(
                ZeroSmartContractAddressNameProvider.Name, nameof(BasicContractZero.
↪DeploySmartContract),
                new ContractDeploymentInput
                {
                    Category = KernelConstants.DefaultRunnerCategory,
```

(continues on next page)

(continued from previous page)

```
                    Code = xxCode
                }.ToByteString()),
            await _transactionGeneratingService.GenerateTransactionAsync(
                ZeroSmartContractAddressNameProvider.Name, nameof(BasicContractZero.
↪DeploySmartContract),
                new ContractDeploymentInput
                {
                    Category = KernelConstants.DefaultRunnerCategory,
                    Code = yyCode
                }.ToByteString())
        };
    }

    return new List<Transaction>();
}

private byte[] GetContractCodes(string contractName)
{
    return ContractsDeployer.GetContractCodes<DeployContractsSystemTransactionGenerator>
↪(_contractOptions
        .GenesisContractDir)[contractName];
}
```

Don't forget to make sure these contracts are referenced by this `AElf.Boilerplate.XXContract.Launcher` project.

1.3.3 More on Boilerplate

Boilerplate is an environment that is used to develop smart contracts and dApps. After writing and testing your contract on **Boilerplate**, you can deploy it to a running **AElf** chain. Internally **Boilerplate** will run an simplified node that will automatically have your contract deployed on it at genesis.

Boilerplate is composed of two root folders: **chain** and **web**. This series of tutorial articles focuses on contract development so we'll only go into the details of the **chain** part of **Boilerplate**. Here is a brief overview of the folders:

```
.
  chain
      src
      contract
          AElf.Contracts.HelloWorldContract
              AElf.Contracts.HelloWorldContract.csproj
              HelloWorldContract.cs
              HelloWorldContractState.cs
              ...
      protobuf
          hello_world_contract.proto
          ...
      test
          AElf.Contracts.HelloWorldContract.Test
              AElf.Contracts.HelloWorldContract.Test.csproj
```

(continues on next page)

(continued from previous page)

```
            HelloWorldContractTest.cs
    ...
```

The hello world contract and its tests are split between the following folders:

- **contract**: this folder contains the csharp projects (.csproj) along with the contract implementation (.cs files).
- **protobuf**: contains the .proto definition of the contract.
- **test**: contains the test project and files (basic xUnit test project).

You can use this layout as a template for your future smart contracts. Before you do, we recommend you follow through all the articles of this series.

You will also notice the **src** folder. This folder contains **Boilerplate**'s modules and the executable for the node.

All production contracts (contracts destined to be deployed to a live chain) must go through a complete review process by the contract author and undergo proper testing. It is the author's responsibility to check the validity and security of his contract. The author should not simply copy the contracts contained in **Boilerplate**. It's the author's responsibility to ensure the security and correctness of his contracts.

1.3.4 Next

You've just seen a short introduction on how to run a smart contract that is already included in **Boilerplate**. The next article will show you a complete smart contract and extra content on how to organize your code and test files.

The main usage of aelf-boilerplate is to develop contracts for AElf blockchains. Once you've downloaded or cloned this project, that process looks something like this:

1. Use **AElf.Boilerplate.sln**, run project **AElf.Boilerplate.Launcher**, and try **Greeter** project located in *web/greeter* to make sure the AElf blockchain can be run in local machine.
2. Use **AElf.Contracts.BingoContract.sln**, run project **AElf.Boilerplate.BingoContract.Launcher**, and try Bingo Game located in *web/ReactNativeBingo*, similar to the code of **Bingo Game**, which is a DApp of the AElf blockchain.
3. Use **AElf.Boilerplate.sln**, modify the *appsettings.json* in project **AElf.Boilerplate.CodeGenerator**, running this project will generate a contract development template as well as a new sln file.
4. With the new sln file you can develop your new contract, and build your new contract project will generate a patched contract dll which can be deployed to AElf TestNet/MainNet.

Besides, we provided demo contracts of most of our **AElf Contract Standards(ACS)**. As shown before, aelf-boilerplate project is enough for you to getting familiar with AElf contract development, but it has to say that aelf-boilerplate is a start point of developing AElf contract, not a destination.

But before you either start try **Greeter** and **Bingo Game**, or ready to develop a smart contract, you'll need to install the following tools and frameworks.

For most of these dependencies we provide ready-to-use command line instructions. In case of problems or if you have more complex needs, we provide the official link with full instructions.

CHAPTER 2

Write Contract

2.1 Smart contract implementation

This article will guide you through how to use **AElf Boilerplate** to implement a smart contract. It takes an example on the **Greeter** contract that's already included in Boilerplate. Based on the concepts this article presents, you'll be able to create your own basic contract.

2.1.1 Greeter contract

The following content will walk you through the basics of writing a smart contract; this process contains essentially four steps:

- **create the project**: generate the contract template using **AElf Boilerplate**'s code generator.
- **define the contract and its types**: the methods and types needed in your contract should be defined in a protobuf file, following typical protobuf syntax.
- **generate the code**: build the project to generate the base contract code from the proto definition.
- **extend the generated code**: implement the logic of the contract methods.

The `Greeter` contract is a very simple contract that exposes a `Greet` method that simply logs to the console and returns a "Hello World" message and a more sophisticated `GreetTo` method that records every greeting it receives and returns the greeting message as well as the time of the greeting.

This tutorial shows you how to develop a smart contract with the contract SDK; you can find more details here. **Boilerplate** will automatically add the reference to the SDK.

2.1.2 Create the project

With **AElf Boilerplate**'s code generator, you can easily and quickly set up a contract project. See *here* for details.

2.1.3 Defining the contract

After creating the contract project, you can define the methods and types of your contract. **AElf** defines smart contracts as services that are implemented using `gRPC` and `Protobuf`. The definition contains no logic; at build time the proto file is used to generate C# classes that will be used to implement the logic and state of the contract.

We recommend putting the contract's definition in **protobuf** folder so that it can easily be included in the *build/generation* process and also that you name the contract with the following syntax **contract_name_contract.proto**:

```
.
  Boilerplate
      chain
          protobuf
              aelf
                  options.proto // contract options
                  core.proto    // core blockchain types
              greeter_contract.proto
              another_contract.proto
              token_contract.proto // system contracts
              acs0.proto // AElf contract standard
              ...
```

The "protobuf" folder already contains a certain amount of contract definitions, including tutorial examples, system contracts. You'll also notice it contains **AElf** Contract Standard definitions that are also defined the same way as contracts. Lastly, it also contains **options.proto** and **core.proto** that contain fundamental types for developing smart contracts, more on this later.

Best practices:

- place your contract definition in **protobuf** folder.
- name your contract with **contractname_contract.proto**, all lower case.

Now let's take a look a the Greeter contract's definition:

```
// protobuf/greeter_contract.proto

syntax = "proto3";

import "aelf/options.proto";

import "google/protobuf/empty.proto";
import "google/protobuf/timestamp.proto";
import "google/protobuf/wrappers.proto";

option csharp_namespace = "AElf.Contracts.Greeter";

service GreeterContract {
    option (aelf.csharp_state) = "AElf.Contracts.Greeter.GreeterContractState";

    // Actions
    rpc Greet (google.protobuf.Empty) returns (google.protobuf.StringValue) { }
    rpc GreetTo (google.protobuf.StringValue) returns (GreetToOutput) { }
```

(continues on next page)

(continued from previous page)

```
    // Views
    rpc GetGreetedList (google.protobuf.Empty) returns (GreetedList) {
        option (aelf.is_view) = true;
    }
}

message GreetToOutput {
    string name = 1;
    google.protobuf.Timestamp greet_time = 2;
}

message GreetedList {
    repeated string value = 1;
}
```

Above is the full definition of the contract; it is mainly composed of three parts:

- **imports**: the dependencies of your contract.
- **the service definition**: the methods of your contract.
- **types**: some custom defined types used by the contract.

Let's have a deeper look at the three different parts.

Syntax, imports and namespace

```
syntax = "proto3";

import "aelf/options.proto";

import "google/protobuf/empty.proto";
import "google/protobuf/timestamp.proto";
import "google/protobuf/wrappers.proto";

option csharp_namespace = "AElf.Contracts.Greeter";
```

The first line specifies the syntax that this protobuf file uses, we recommend you always use **proto3** for your contracts. Next, you'll notice that this contract specifies some imports, let's analyze them briefly:

- **aelf/options.proto** : contracts can use **AElf** specific options; this file contains the definitions. One example is the **is_view** options that we will use later.
- **empty.proto, timestamp.proto and wrappers.proto** : these are proto files imported directly from protobuf's library. They are useful for defining things like an empty return value, time, and wrappers around some common types such as string.

The last line specifies an option that determines the target namespace of the generated code. Here the generated code will be in the `AElf.Contracts.Greeter` namespace.

The service definition

```
service GreeterContract {
    option (aelf.csharp_state) = "AElf.Contracts.Greeter.GreeterContractState";

    // Actions
    rpc Greet (google.protobuf.Empty) returns (google.protobuf.StringValue) { }
    rpc GreetTo (google.protobuf.StringValue) returns (GreetToOutput) { }

    // Views
    rpc GetGreetedList (google.protobuf.Empty) returns (GreetedList) {
        option (aelf.is_view) = true;
    }
}
```

The first line here uses the `aelf.csharp_state` option to specify the name (full name) of the state class. This means that the state of the contract should be defined in the `GreeterContractState` class under the `AElf.Contracts.Greeter` namespace.

Next, two **action** methods are defined: `Greet` and `GreetTo`. A contract method is defined by three things: the **method name**, the **input argument(s) type(s)** and the **output type**. For example, `Greet` requires that the input type is `google.protobuf.Empty` that is used to specify that this method takes no arguments and the output type will be a google.protobuf.StringValue is a traditional string. As you can see with the `GreetTo` method, you can use custom types as input and output of contract methods.

The service also defines a **view** method, that is, a method used only to query the contracts state, and that has no side effect on the state. For example, the definition of `GetGreetedList` uses the **aelf.is_view** option to make it a view method.

Best practice:

- use **google.protobuf.Empty** to specify that a method takes no arguments (import `google/protobuf/empty.proto`).
- use **google.protobuf.StringValue** to use a string (import `google/protobuf/wrappers.proto`).
- use the **aelf.is_view** option to create a view method (import `aelf/options.proto`).
- use the **aelf.csharp_state** to specify the namespace of your contracts state (import `aelf/options.proto`).

Custom types

```
message GreetToOutput {
    string name = 1;
    google.protobuf.Timestamp greet_time = 2;
}

message GreetedList {
    repeated string value = 1;
}
```

The protobuf file also includes the definition of two custom types. The **GreetToOutput** is the type returned by the `GreetTo` method and `GreetedList` is the return type of the `GetGreetedList` view method. You'll notice the **repeated** keyword the `GreetedList` message. This is protobuf syntax to represent a collection.

Best practice:

- use **google.protobuf.Timestamp** to represent a point in time (import `google/protobuf/timestamp.proto`).
- use **repeated** to represent a collection of items of the same type.

2.1.4 Extend the generated code

After defining and generating the code from the definition, the contract author extends the generated code to implement the logic of his contract. Two files are presented here:

- **GreeterContract**: the actual implementation of the logic, it inherits from the contract base generated by protobuf.
- **GreeterContractState**: the state class that contains properties for reading and writing the state. This class inherits the `ContractState` class from the C# SDK.

```
// contract/AElf.Contracts.GreeterContract/GreeterContract.cs

using Google.Protobuf.WellKnownTypes;

namespace AElf.Contracts.Greeter
{
    public class GreeterContract : GreeterContractContainer.GreeterContractBase
    {
        public override StringValue Greet(Empty input)
        {
            Context.LogDebug(() => "Hello World!");
            return new StringValue {Value = "Hello World!"};
        }

        public override GreetToOutput GreetTo(StringValue input)
        {
            // Should not greet to empty string or white space.
            Assert(!string.IsNullOrWhiteSpace(input.Value), "Invalid name.");

            // State.GreetedList.Value is null if not initialized.
            var greetList = State.GreetedList.Value ?? new GreetedList();

            // Add input.Value to State.GreetedList.Value if it's new to this list.
            if (!greetList.Value.Contains(input.Value))
            {
                greetList.Value.Add(input.Value);
            }

            // Update State.GreetedList.Value by setting it's value directly.
            State.GreetedList.Value = greetList;

            return new GreetToOutput
            {
                GreetTime = Context.CurrentBlockTime,
                Name = input.Value.Trim()
            };
        }
```

(continues on next page)

(continued from previous page)

```
        public override GreetedList GetGreetedList(Empty input)
        {
            return State.GreetedList.Value ?? new GreetedList();
        }
    }
}
```

```
// contract/AElf.Contracts.GreeterContract/GreeterContractState.cs

using AElf.Sdk.CSharp.State;

 namespace AElf.Contracts.Greeter
 {
    public class GreeterContractState : ContractState
    {
        public SingletonState<GreetedList> GreetedList { get; set; }
    }
 }
```

Let's briefly explain what is happening in the `GreetTo` method:

Asserting

```
Assert(!string.IsNullOrWhiteSpace(input.Value), "Invalid name.");
```

When writing a smart contract, it is often useful (and recommended) to validate the input. **AElf** smart contracts can use the `Assert` method defined in the base smart contract class to implement this pattern. For example, here, the method validates that the input string is null or composed only of white spaces. If the condition is false, this line will abort the execution of the transaction.

Accessing and saving state

```
var greetList = State.GreetedList.Value ?? new GreetedList();
...
State.GreetedList.Value = greetList;
```

From within the contract methods, you can easily access the contracts state through the `State` property of the contract. Here the state property refers to the `GreeterContractState` class in which is defined the `GreetedList` collection. The second effectively updates the state (this is needed; otherwise, the method would have no effect on the state).

Note that because the `GreetedList` type is wrapped in a `SingletonState` you have to use the `Value` property to access the data (more on this later).

Logging

```
Context.LogDebug(() => "Hello {0}!", input.Value);
```

It is also possible to log from smart contract methods. The above example will log "Hello" and the value of the input. It also prints useful information like the ID of the transaction. It will print in the console log if you launch the node with DEBUG mode. This is only for debug use and has no impacts on state at all.

More on state

As a reminder, here is the state definition in the contract (we specified the name of the class and a type) as well as the custom type `GreetedList`:

```
service GreeterContract {
    option (aelf.csharp_state) = "AElf.Contracts.Greeter.GreeterContractState";
    ...
}

// ...

message GreetedList {
    repeated string value = 1;
}
```

The `aelf.csharp_state` option allows the contract author to specify in which namespace and class name the state will be. To implement a state class, you need to inherit from the `ContractState` class that is contained in the C# SDK (notice the `using` statement here below).

Below is the state class that we saw previously:

```
using AElf.Sdk.CSharp.State;

 namespace AElf.Contracts.Greeter
 {
    public class GreeterContractState : ContractState
    {
        public SingletonState<GreetedList> GreetedList { get; set; }
    }
 }
```

The state uses the custom `GreetedList` type, which was generated from the Protobuf definition at build time and contained exactly one property: a singleton state of type `GreetedList`.

The `SingletonState` is part of the C# SDK and is used to represent exactly **one** value. The value can be of any type, including collection types. Here we only wanted our contract to store one list (here a list of strings).

Note that you have to wrap your state types in a type like `SingletonState` (others are also available like `MappedState`) because behind the scene, they implement the state read and write operations.

2.2 Bingo Game

2.2.1 Requirement Analysis

Basic Requirement

Only one rule Users can bet a certain amount of ELF on Bingo contract, and then users will gain more ELF or to lose all ELF bet before in the expected time.

For users, operation steps are as follows:

1. Send an Approve transaction by Token Contract to grant Bingo Contract amount of ELF.
2. Bet by Bingo Contract, and the outcome will be unveiled in the expected time.
3. After a certain time, or after the block height is reached, the user can use the Bingo contract to query the results, and at the same time, the Bingo contract will transfer a certain amount of ELF to the user (If the amount at this time is greater than the bet amount, it means that the user won; vice versa).

2.2.2 API List

In summary, two basic APIs are needed:

1. Play, corresponding to step 2;
2. Bingo, corresponding to step 3.

In order to make the Bingo contract a more complete DApp contract, two additional Action methods are added:

1. Register, which creates a file for users, can save the registration time and user's eigenvalues (these eigenvalues participate in the calculation of the random number used in the Bingo game);
2. Quit, which deletes users' file.

In addition, there are some View methods for querying information only:

1. GetAward, which allows users to query the award information of a bet;
2. GetPlayerInformation, used to query player's information.

Method	Parameters	Return	function
`Register`	Empty	Empty	register player information
`Quit`	Empty	Empty	delete player information
`Play`	Int64Value anount you debt	Int64Value the resulting block height	debt
`Bingo`	Hash the transaction id of Play	BoolValue True indicates win	query the game's result
`GetAward`	Hash the transaction id of Play	Int64Value award	query the amount of award
`GetPlayerInformation`	Address player's address	Player-Information	query player's information

2.2.3 Write Contract

Use the code generator to generate contracts and test projects

Open the `AElf.Boilerplate.CodeGenerator` project in the *AElf.Boilerplate<https://aelf-boilerplate-docs.readthedocs.io/en/latest/usage/setup.html#try-code-generator>*, and modify the Contents node in appsetting.json under this project:

```
{
  "Contents": [
  {
    "Origin": "AElf.Contracts.HelloWorldContract",
    "New": "AElf.Contracts.BingoContractDemo"
  },
  {
    "Origin": "HelloWorld",
    "New": "Bingo"
  },
  {
    "Origin": "hello_world",
    "New": "bingo"
  }
  ],
}
```

Then run the `AElf.Boilerplate.CodeGenerator` project. After running successfully, you will see a *AElf.Contracts.BingoContractDemo.sln* in the same directory as the *AElf.Boilerplate.sln* is in. After opening the sln, you will see that the contract project and test case project of the Bingo contract have been generated and are included in the new solution.

Define Proto

Based on the API list in the requirements analysis, the bingo_contract.proto file is as follows:

```
syntax = "proto3";
import "aelf/core.proto";
import "aelf/options.proto";
import "google/protobuf/empty.proto";
import "google/protobuf/wrappers.proto";
import "google/protobuf/timestamp.proto";
option csharp_namespace = "AElf.Contracts.BingoContractDemo";
service BingoContract {
    option (aelf.csharp_state) = "AElf.Contracts.BingoContractDemo.BingoContractState";

    // Actions
    rpc Register (google.protobuf.Empty) returns (google.protobuf.Empty) {
    }
    rpc Play (google.protobuf.Int64Value) returns (google.protobuf.Int64Value) {
    }
    rpc Bingo (aelf.Hash) returns (google.protobuf.BoolValue) {
    }
    rpc Quit (google.protobuf.Empty) returns (google.protobuf.Empty) {
    }

    // Views
```

(continues on next page)

(continued from previous page)

```
    rpc GetAward (aelf.Hash) returns (google.protobuf.Int64Value) {
        option (aelf.is_view) = true;
    }
    rpc GetPlayerInformation (aelf.Address) returns (PlayerInformation) {
        option (aelf.is_view) = true;
    }
}
message PlayerInformation {
    aelf.Hash seed = 1;
    repeated BoutInformation bouts = 2;
    google.protobuf.Timestamp register_time = 3;
}
message BoutInformation {
    int64 play_block_height = 1;
    int64 amount = 2;
    int64 award = 3;
    bool is_complete = 4;
    aelf.Hash play_id = 5;
    int64 bingo_block_height = 6;
}
```

Contract Implementation

Here only talk about the general idea of the Action method, specifically need to turn the code:

https://github.com/AElfProject/aelf-boilerplate/blob/dev/chain/contract/AElf.Contracts.BingoGameContract/BingoGameContract.cs

Register & Quit

Register

- Determine the Seed of the user, Seed is a hash value, participating in the calculation of the random number, each user is different, so as to ensure that different users get different results on the same height;
- Record the user's registration time.

Quit Just delete the user's information.

Play & Bingo

Play

- Use TransferFrom to deduct the user's bet amount;
- At the same time add a round (Bount) for the user, when the Bount is initialized, record three messages 1.PlayId, the transaction Id of this transaction, is used to uniquely identify the Bout (see BoutInformation for its data structure in the Proto definition);
- Amount Record the amount of the bet 3.Record the height of the block in which the Play transaction is packaged.

Bingo

- **Find the corresponding Bout according to PlayId, if the current block** height is greater than PlayBlockHeight + number of nodes * 8, you can get the result that you win or lose;
- **Use the current height and the user's Seed to calculate a random** number, and then treat the hash value as a bit Array, each of which is added to get a number ranging from 0 to 256.
- **Whether the number is divisible by 2 determines the user wins or** loses;
- **The range of this number determines the amount of win/loss for the** user, see the note of GetKind method for details.

2.2.4 Write Test

Because the token transfer is involved in this test, in addition to constructing the stub of the bingo contract, the stub of the token contract is also required, so the code referenced in csproj for the proto file is:

```
<ItemGroup>
  <ContractStub Include="..\..\protobuf\bingo_contract.proto">
    <Link>Protobuf\Proto\bingo_contract.proto</Link>
  </ContractStub>
  <ContractStub Include="..\..\protobuf\token_contract.proto">
    <Link>Protobuf\Proto\token_contract.proto</Link>
  </ContractStub>
</ItemGroup>
```

Then you can write test code directly in the Test method of BingoContractTest. Prepare the two stubs mentioned above:

```
// Get a stub for testing.
var keyPair = SampleECKeyPairs.KeyPairs[0];
var stub = GetBingoContractStub(keyPair);
var tokenStub =
    GetTester<TokenContractContainer.TokenContractStub>(
        GetAddress(TokenSmartContractAddressNameProvider.StringName), keyPair);
```

The stub is the stub of the bingo contract, and the tokenStub is the stub of the token contract.

In the unit test, the keyPair account is given a large amount of ELF by default, and the bingo contract needs a certain bonus pool to run, so first let the account transfer ELF to the bingo contract:

```
// Prepare awards.
await tokenStub.Transfer.SendAsync(new TransferInput
{
    To = BingoContractAddress,
    Symbol = "ELF",
    Amount = 100_00000000
});
```

Then you can start using the Bingo contract. Register

```
await stub.Register.SendAsync(new Empty());
```

After registration, take a look at PlayInformation:

```
// Now I have player information.
var address = Address.FromPublicKey(keyPair.PublicKey);
{
    var playerInformation = await stub.GetPlayerInformation.CallAsync(address);
    playerInformation.Seed.Value.ShouldNotBeEmpty();
    playerInformation.RegisterTime.ShouldNotBeNull();
}
```

Bet, but before you can bet, you need to Approve the bingo contract:

```
// Play.
await tokenStub.Approve.SendAsync(new ApproveInput
{
    Spender = BingoContractAddress,
    Symbol = "ELF",
    Amount = 10000
});
await stub.Play.SendAsync(new Int64Value {Value = 10000});
```

See if Bout is generated after betting.

```
Hash playId;
{
    var playerInformation = await stub.GetPlayerInformation.CallAsync(address);
    playerInformation.Bouts.ShouldNotBeEmpty();
    playId = playerInformation.Bouts.First().PlayId;
}
```

Since the outcome requires eight blocks, you need send seven invalid transactions (these transactions will fail, but the block height will increase) :

```
// Mine 7 more blocks.
for (var i = 0; i < 7; i++)
{
    await stub.Bingo.SendWithExceptionAsync(playId);
}
```

Last check the award, and that the award amount is greater than 0 indicates you win.

```
await stub.Bingo.SendAsync(playId);
var award = await stub.GetAward.CallAsync(playId);
award.Value.ShouldNotBe(0);
```