Axoni's Enterprise Use of Truffle

Axoni Background¶

At Axoni, our mission is to make shared information more trusted and reliable. We do this through distributed technology that we call "Ethereum-inspired", which has both a broad and a specific meaning. In the broad sense, the core technology plays a role for its users analogous to the public blockchain, presenting a powerful and extensible capacity to side-step undesired intermediaries (yes, these exist for banks too...). In a more precise technical sense, our platform borrows certain concepts from Ethereum (e.g. accounts, contracts, gas...) and, importantly, we make efforts to comply with Ethereum-world interfaces (e.g. evm bytecode, web3 JSON-rpc).

Of course, an up-shot of the last is the ability to take part in all the innovation taking place in the broader Ethereum ecosystem, including client-side libraries that enable a "tastier" application development process. To this end, within the last year we have worked to leverage the Truffle suite and ecosystem to great effect. In fact, asreported , it is fair to say that Truffle serves by and large as the main toolset for on-chain application development at Axoni.

Using Truffle at Axoni¶

It was a delight for us to find that, given our compliance with the web3 JSON-RPC spec, truffle basically works "out-of-the-box" with no extensive customization required for many very useful stages in the application lifecycle: develop, build, test....

One notable exception: we found the nativetruffle migrate functionality not extensible to our use case. In part, this is due to the fact that our smart contract applications implement somethingsimilar to the proxy pattern , which runs counter to the "migration" paradigm. On the other hand, and for reasons beyond the scope of this post, tools such aszoslib designed to handle upgradeable smart contract application deployments, generate smart contract implementations incompatible with our use case. At a high level, our smart contract framework is optimized for high-throughput and highly complex Solidity applications which enables the ability to make automatic upgrades based on semantic versioning. [Note: I plan to speak more about our particular use cases and smart contract topologies at Trufflecon on August 3rd]

In any case, especially due to the often very high levels of complexity entailed by our use cases– combined with the notedapplication-level gotcha's of upgrade design patterns in solidity itself–we are biased towards a simple and opinionated declarative format that application and production engineers can easily understand and (when needed) manipulate.

This raises the question -- why use truffle for this? It seems there are at least three reasons:

1. truffle-config.js
2. /truffle.js
3. serves as an already existing central source of shared information about various network providers relevant for a particular application developer. It only makes sense to extend this format for use in production-like environments as well.
4. Truffle code artifacts, a battle-tested standard for smart contracts and solidity code, in particular, provide a common interface for a toolchain. By virtue of using truffle, there is no need to re-create this format. For a non-development deployment, acontracts_build_directory
5. parameter is supplied, which results in a pull from a remote artifacts repository.
6. Obviously@truffle/contract
7. itself provides a nice object oriented interface for deploying and performing operations on contracts within this script, thus avoiding any reinventing the wheel within the deployment library itself.

Thus, we ventured to build a package on top of truffle that manages configuration-driven smart contract deployments and upgrades.

Building a Truffle-Based Deployment Tool¶

The first question is what the configuration file should look like? We determined the "Straw Man" to be, simply, a two-dimensional list of job types and associated contracts specified for deployment. Starting with this, we found another concept we needed to add ("custom-partition") that is specific to Axcore, relating to the way the node software provisions data across network partitions; certain contracts are provisioned differently from others. In addition to this, we found that the only other concept we needed was a parameter specifying dependencies ("deps") that linearizes to a certain order to deployment (which may be necessary, for instance, if a given storage address stores reference to another). For concision, we added limited regex support. In the end, we arrived at a yaml-based file looking very much like the below:

version :

0.0.1 setup :

partitions :

```
-
custom-partition :
*
-
custom-partition :
^public-common
contracts :
-
name :
MagmaRegistryFooStorage
-
name :
MagmaRegistryBarStorage
deps :
-
MagmaRegistryFooStorage upgrade :
partitions :
-
custom-partition :
"*"
contracts :
-
name :
MagmaRegistryFoo
-
name :
MagmaRegistryBar
```

In terms of implementation, a truffle exec script reads this configuration file, and for each contract, a) deploys the contract b) performs some basic API configuration and c) installs it in the application through invoking a basicinitialization method. The basic top-level run (truffle exec) script looks very much like the below:

```
const
run
=
async
( done )
=>
{
let
jobParams
=
args ;
// Configure specific variables for a PROD-like environment:
if
( process . env . ENV
===
'PROD' )
{
jobParams
=
Object . assign ( args ,
handleProd ());
}
//hoisting injected vars to global scope:
global . web3
=
web3 ;
global . artifacts
=
artifacts ;
// Create instance of job based on type
const
jobInstance
=
new
Job ( jobParams . job ,
jobParams );
//Run job by deploying and invoking an initialization method on each contract
await
jobInstance . run (). catch ( e
=>
done ( e ));
done (); };
```

A command-line client can specify a given job to run, such as the "upgrade" job specified above. Given the above configuration, the below command would deploy (if necessary) and initialize "MagmaRegistryFooStorage", followed by "MagmaRegistryBarStorage" from a given artifacts repository:

```
etna --job upgrade --contracts_build_directory /path/to/artifacts
```

Future Plans and an Announcement¶

I am pleased to announce that we plan to open-source our core Solidity smart contract library, which provides an alternative version of the Proxy pattern, along with this aforementioned deployment tool. We believe both could be useful both for private and public applications and the Axoni team is excited to see what the community builds with these tools.

In the meantime, we hope the above at least provides a high-level example of an approach to building lightweight extensions within truffle, and as a simplifying approach to the smart contract deployment process, especially for Proxy Pattern topologies.

The Axoni team and I look forward to discussing this topic and much more (such as Axlang, our Scala-based Solidity-alternative) at Trufflecon.