Harnessing distributed ledgers to develop a trusted platform for the energy market

Introduction

Utilidex's mission is to revolutionise the way customers buy, sell and optimise their energy estate. The company's digital platform, known as the Utilidex | Hub is used by a range of participants in the market, including energy generators/prosumers, suppliers and corporates. This customer set of buyers and sellers, working on a common platform, creates a unique opportunity to leverage the power of Blockchains, or distributed ledger, technology to facilitate new types of energy transactions. In partnership, Utilidex and Microsoft helped develop the start of an enterprise solution that includes the security and trust features needed for practical commercial use.

Utilidex is publishing some of the work they had been doing with Microsoft on Blockchains, as part of their wider initiative to demystify the technology and help with its adoption in energy as explained on their blog. This document discusses incorporating a Blockchain into an enterprise solution for the energy market. It explores security, deployment, integrating with applications, and user authentication. Read this document to see how a Blockchain can become part of your solution and how Azure services can provide it with the enterprise capabilities needed. Specifically, this document looks at distributed ledgers based on Ethereum Blockchain technology.

Utilidex has been working with Microsoft for several years. This paper is the outcome of one of the workshops run with Utilidex at Microsoft offices in London between 21 and 24 February 2017. This paper shares our learning to that point in this rapidly developing area. It documents our journey and considerations at that time. We expect that relentless progress will offer alternatives to our approaches. It is the identifying of enterprise requirements that we believe will provide value for readers interested in using this transformational technology.

Contents

Customer Profile

Whilst still a young company, the team have some 30+ years' experience between them in energy, implementing enterprise-grade solutions across billing, trading and settlements in UK, EU and Australia.

The company's Utilidex | Hub product provides services to a number of major corporates, generators and suppliers. And with rapid changes in the energy market, around how energy is generated and consumed, the company is looking at a number of ways that Blockchain could help.

"*Our energy mix is changing, and so too is the opportunity for customers to buy/sell and time shift their energy, all with an aim to either reduce costs or purchase green energy*" commented Richard Brys from Utilidex, "*when you combine some interesting technologies like IoT, Analytics, Algorithms and now Blockchain, you have a real recipe for something new.* "

Utilidex are aiming to extend their solution with a new type of exchange (known as the "Utilidex | Flexchange") for generators, retailers and consumers.

The Team
- Richard Brys, CEO & Product Architect, Utilidex
- Mike McCloskey, Director & Co-Founder, Utilidex
- Samir Gupta, Engineering Director, Utilidex
- Sudhir Gupta, Technical Lead, Utilidex
- Abhinav Jain, Technical Lead, Utilidex
- Vaibhav Mishra, Technical Lead, Utilidex
- David Goon, Solutions Strategist, Microsoft UK (DX)
- Mike Ormond, Technical Evangelist, Microsoft UK (DX)
- Jonathan Collinge, Technical Evangelist, Microsoft UK (DX)
- Ben Roscorla, Technical Evangelist, Microsoft UK (DX)
- Thomas Conte, Senior Software Development Engineer, Microsoft (TED)

Reviewers
- Gina Dragulin, Director of Audience Evangelism, Microsoft UK (DX)
- John Donnelly, Technical Evangelist, Microsoft UK (DX)
- Michael Platt, Principal Software Development Engineer, Microsoft (TED)

Problem Statement

Utilidex needs a way to facilitate direct energy transactions via their solution minimizing escrow. One example where this would be beneficial is when one party is over-consuming and another has

capacity. They could balance their energy use directly and at the same time help balance the grid. A distributed ledger promises to be an enabler as it provides the trust to allow a direct exchange to take place. Building an enterprise-level solution for Utilidex has many security and operational requirements.

Requirements

While Bitcoin and similar Unspent Transaction Output (UTXO) distributed ledgers are well established, those that can execute code, such as Ethereum Smart Contracts, are relatively new (e.g. Ethereum released in 2015) and not fully developed for private enterprise use.

The following aspects were prioritised:
- Establishing a secure environment for the distributed ledger infrastructure.
- Automating deployment of distributed ledger participant components.
- Developing distributed ledger code that minimizes risks while maximizing value.
- Building a pattern for enterprise solution integration with the distributed ledger.
- Protecting access to the whole with user-level authentication and authorization.

Each requirement will be explored in its own Technical Discussion section in this document.
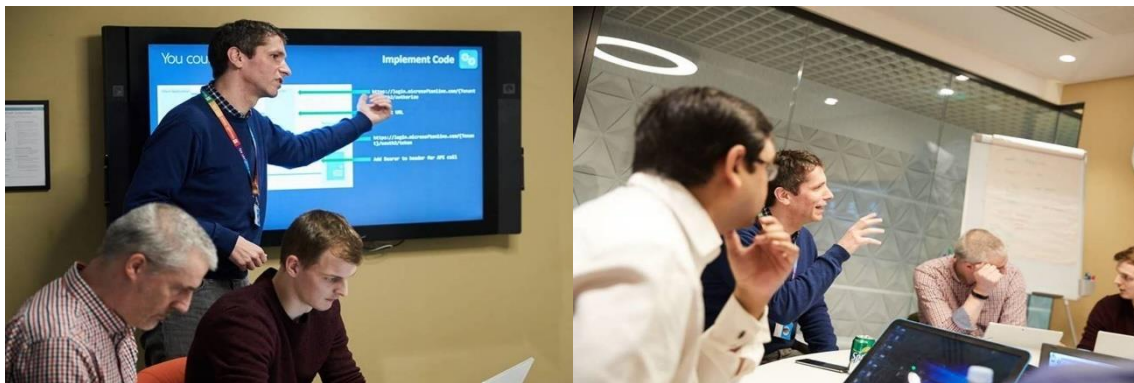
Partner Context

Utilidex wanted three key outcomes from the development activity.
- Demystify the technology so it could be adopted by the energy market
- To demonstrate a simple straightforward use-case before adding complexity.
- To provide a pragmatic working implementation given the hype and high-profile attacks distributed ledgers had suffered recently, e.g. Ethereum's "The DAO" attack.

Microsoft, working from Reading in the UK, adopted a collaborative learning approach with the Utilidex team. This was facilitated by Visual Studio Team Services and a shared Microsoft OneNote notebook. Regular Skype for Business conferences allowed us to align understanding.

- Starting late 2016, we ran preparation days which built-up knowledge and experience for the development week. This allowed the team time to identify and overcome blockers for the evolving technology.
- A comprehensive preparation guide was then created to maximize productivity during the development workshop in London.
- The days in London were then structured into two phases. First, experience-sharing; and then a development sprint where new code was built on the foundations established.

From February 21 to 24, we started from a fresh Azure subscription provided by Utilidex and finished with a very basic end-to-end solution leveraging a distributed ledger. This solution allowed the creation of entries into a distributed ledger deployed for multiple participants.

Solution Design

Our design used a distributed ledger for what we believe are its strengths (immutability) and implements business logic elsewhere. This simplifies the distributed ledger implementation and lowers the risks of errors with complex distributed ledger code.

By using Azure to then build out the enterprise solution, we leveraged well-understood and proven technologies to deliver business value to Utilidex. We also put the distributed ledger behind an API so that it can be swapped out with manageable impact to the solution.

There is a choice to use either a public distributed ledger or to set up a new one for a limited set of participants. BitCoin and Ethereum are examples that have single public distributed ledgers that are globally accessible. We chose to set a new one up for Utilidex as there is a known set of participating organizations and this allows us to control how it is created and run for development and test. Our network follows the consortium model. A later transition to a public distributed ledger is possible.

For identity management, we use Azure Active Directory and have a central instance controlled by Utilidex specifically for the Flexchange. New participants would register their users to this instance and thus gain access to the solution if Utilidex approves. This avoids federation complexity initially though it will need to be revisited if user single sign-on is desired.

Technology SWOT Analysis

To choose the distributed ledger technology to use, the team evaluated multiple available implementations via SWOT Analysis. At that time, we felt these could be divided into Commercial and open-sourced distributed ledgers. Commercial implementations are third-party SaaS solutions while Ethereum is an example of open-source.

COMMERCIAL DISTRIBUTED LEDGERS	ETHEREUM DISTRIBUTED LEDGER
Strengths: • Solution-oriented • Formal support provisions • Commercially viable/focused Weaknesses: • Lock-in to vendor • Dependency on vendor skills • Abstracted/Prescribed so less flexible • Offerings are all relatively new Opportunity • Build a relationship with the vendor • Scalable as vendor provides guarantees • Rapid development due to solution focus Threat • Dependency on vendor (roadmap, lifecycle, connection, pricing)	Strengths: • Availability of tools • Bletchley* uses it • Community support and participation • Familiar due to popularity • Tooling is functional and improving Weaknesses: • GPL licensing so must be careful with IP • Transaction rate is low • Volatility as it is still developing • Consensus is resource intensive Opportunity: • Proof of Stake consensus scalability • Open and extensible Threat: • Security of implementation • Dependency on Ethereum Foundation • Credibility due to number of changes • Roadmap is unclear

* Bletchley allows developers to automatically create a Blockchain deployment for test purposes.

We decided on Ethereum for its tools support and active community. Shortly after our development workshop concluded in February, Quorum was announced. While too late for us to assess, it should be considered for any future implementations.

Scenarios

Based on the Ethereum implementation, we then designed the architecture for the following user journeys.

1. A new user registers an organization to participate in the Utilidex distributed ledger. This results in a new deployment of services and components to allow the registered organization to interact with Ethereum.
2. The user creates one or more Ethereum accounts with which to buy/sell energy on behalf of the organization. This is done via a *Member Deployment* web-based portal

3. The user proposes a new energy buy/sell via a central web-based portal with another organization.

4. Users from each organization can agree to the proposal. This is done via each *Member Deployment* web-based portals.

By "deployment" we mean all the elements needed for an organization to participate in the distributed ledger. This includes virtual machines, networking, web applications and APIs, databases and other required Azure services. These are deployed to an Azure subscription separate from Utilidex.

An initial design was drafted to illustrate the interconnections. The orange Org Node box denotes the possibility of multiple instances which is refined in the following implementation diagram.
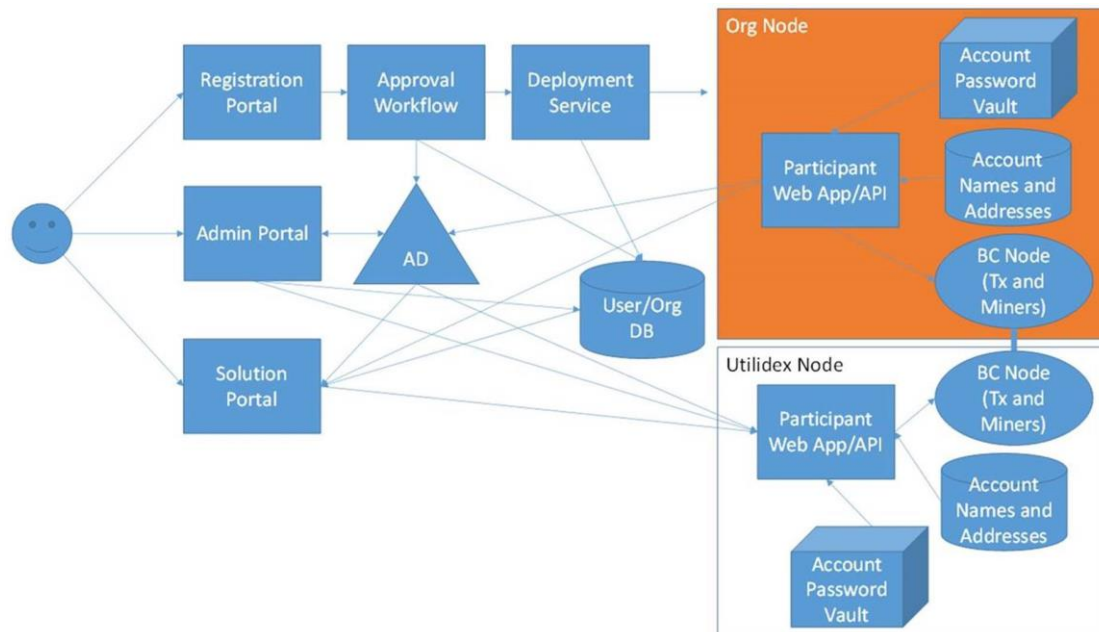
In the description below, words in *italics* refer to components in the above diagram.
- A new user enters organization details via a *Registration Portal*. The registration goes through an *Approval Workflow* within Utilidex.
- On approval:
  - The user as added to the Flexchange Azure *AD*.
  - User and organization details are tracked in the *User/Org DB*.
  - The *Deployment Service* then deploys an *Org Node* (member deployment). Each registered organization will run its own member deployment.
- The *Admin Portal* can be used to add or remove additional organization users.
- The *Solution Portal* is where new agreement proposals are initiated.
- The *Participant Web App/API* allows users to create new Ethereum accounts and to agree proposals.
  - Ethereum account details are tracked in the *Account Names and Addresses* DB.
  - Ethereum account secrets are backed up to the *Account Password Vault*.
- *BC Node (Tx and Miners)* are the actual Ethereum client instances.

The above is refined into the implementation design below.
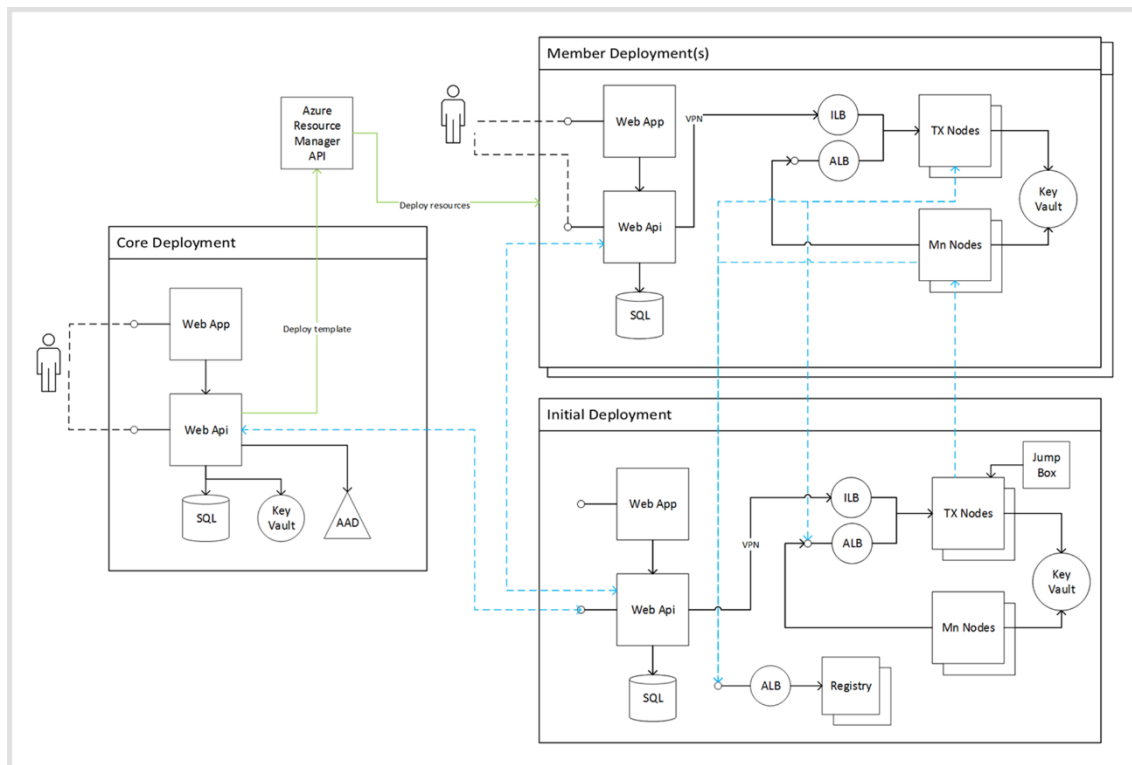
The major elements in the diagram are:

1. *Core Deployment* – the deployment that manages the solution and is run by Utilidex. It contains new user registration and new agreement proposal functions.
2. *Initial Deployment* – this is the deployment for Utilidex to participate in the Ethereum instance. It defines the Ethereum genesis block which identifies the new instance, and hosts the IP address registry of participating member deployments.
3. *Member Deployment(s)* – These are additional deployments for organizations participating in the Utilidex distributed ledger. They contain Ethereum account creation and proposal agreement functions. They sit in their own independent Azure subscriptions.

Components in the implementation design:

- *AAD* – Azure Active Directory for user and application access and authentication.
- *Web App* – ASP.NET MVC web application running in App Services.
- *Web Api* – REST-based API App running in App Services.
- *SQL* – Azure SQL Databases used for various tracking data.
- *VPN* – Virtual Private Network established via App Service Virtual Network integration.
- *TX Nodes* – Ethereum transaction nodes being gateways to the distributed ledger.
- *Mn Nodes* – Ethereum miner nodes that verify entries and builds the distributed ledger.
- *Registry* – Ethereum registry containing addresses of participating member deployments.
- *Jump Box* – Virtual machine containing developer tools for deployment of Ethereum code to the distributed ledger.
- *ILB* – Azure Internal Load Balancer for VPN traffic from App Services.
- *ALB* – Azure Load Balancer for public Ethereum traffic (synchronization and registry).

The design provides for the following:

1. Member deployments are automated via Azure Resource Manager. This could eventually be self-service via Azure Marketplace. Member deployments are made in their own Azure subscriptions and provides confidence that they are independent of Utilidex.

2. Azure App Services is used to host web applications and APIs. This maximizes developer productivity via established DevOps patterns. It minimizes impact to Utilidex development processes versus working in the Ethereum IaaS-based environment.

GitHub Repositories

All output from this effort will be made available from the following repositories. We will publish them as soon as necessary legal and technical reviews are complete.

While all effort was made to align to good practise, the implementations will not follow ideal patterns due to the nature and short duration of the workshop. In cases where a quick implementation was chosen over the best approach, a note is included for future development.

Repository	Summary
https://github.com/utilidex/projectx-flexchange	Landing page for all repositories
https://github.com/utilidex/projectx-flexchange-orchestration	Overall orchestration scripts to stand up a new chain or add a member, etc.
https://github.com/utilidex/projectx-flexchange-smart-contract	Solidity EnergyExchange smart contract implementation
https://github.com/utilidex/projectx-flexchange-active-directory	Azure Active Directory configuration instructions.
https://github.com/utilidex/projectx-flexchange-core-apps	Core deployment Web and API Applications.
https://github.com/utilidex/projectx-flexchange-member-apps	Member deployment Web and API Applications.
https://github.com/utilidex/projectx-flexchange-vmss-automation	Automation scripts to start / stop Ethereum nodes.
https://github.com/utilidex/projectx-flexchange-dev-vm	DevVm/Jump Box in the *Initial Deployment*.
https://github.com/utilidex/projectx-flexchange-keystore-backup	Service to backup keystore.
https://github.com/utilidex/projectx-flexchange-client-watchdog	Service to monitor and restart Ethereum Geth clients on stalled nodes.
https://github.com/utilidex/projectx-flexchange-member-services	ARM template and PS scripts for App Services components in member deployments and integration to existing VNet.

Technical Discussion

Ethereum

Ethereum is an open-source distributed ledger, or Blockchain, implementation with Smart Contract functionality. It is available as a public network (https://ethstats.net/) but can also be used to build private or consortium networks which are limited to allowed participants.

Smart Contracts are code published onto the Ethereum Blockchain. They can be invoked and enforce a strict set of rules and adding verified outputs to an immutable audit trail. Smart Contracts can be written in a variety of languages. Solidity is the most popular Smart Contract language for Ethereum.

An Ethereum network consists of nodes, i.e. a set of compute resources, running an Ethereum client such as "Geth" (the Go-Ethereum client) which maintains a copy of the Blockchain database. The client is responsible for:

- Peer discovery (finding other nodes to connect to)
- Mining (the process of coming to consensus through proof-of-work)
- Synchronisation of the Blockchain and
- Acting as a transaction gateway

Discovery can be achieved either through a broadcast protocol or by constraining valid peers using a registry service. Proof-of-work is a consensus mechanism that requires completion of a predefined piece of work, typically by solving a computationally-expensive problem. Participants are incentivised to compete while correctly enforcing the consensus rules. They make a sacrifice (compute resource) with the promise of reward (Ether, the currency of Ethereum). It also acts to deter denial-of-service attacks. All nodes will sync the Blockchain while those designated as transaction nodes operate as a gateway into the network. Mining nodes also participate in forming consensus.

The Blockchain is an ever-growing chain of blocks containing verified transactions, each immutably linked together. The length of the chain is often referred to as the *block height*. Interaction with the Blockchain is through RPC calls into Ethereum nodes, typically transaction nodes, to initiate transactions or other operations. Such interactions must be funded in Ether. Cryptographic functions such as hashing and signing are used extensively in Blockchain implementations. Assets are often stored off-chain and referenced on the chain with a hash that validates their authenticity.

For more information about Ethereum visit https://www.ethereum.org/ and http://www.ethdocs.org/en/latest/.

Establishing a protected security perimeter for the Ethereum infrastructure.
Building on EthereumEx
The Bletchley (Ethereum Consortium Network) template was initially considered as a basis for the development effort.

> https://github.com/Azure/azure-quickstart-templates/tree/master/ethereum-consortium-blockchain-network

However, a real-world deployment would have the requirements below:
- The trust network must be distributed with each organization retaining control over their own deployments. This implies separate Azure subscriptions and resource groups.
- The deployment of a new Blockchain, and of new member deployments connecting to it needs to be automated. Member deployments have additional components such as applications, APIs and databases.
- New member deployments must be able to attach to an existing Blockchain.
- We must secure the network and prevent auto-discovery of member deployments.

The Bletchley template at the time only supported Ethereum and only for deployments within a single resource group which cannot be spread across subscriptions. It also did not support adding new members to the network. In March 2017, support was announced for multi-member consortium Blockchain networks, too late for our use in February.

> https://azure.microsoft.com/en-us/blog/multi-member-consortium-blockchain-networks-on-azure/

We tried to modify the original Bletchley template for our purposes. We used the template for the *Initial Deployment* and further instances of the template for additional member deployments. The *Initial Deployment* would have the registrar service. The network then needed to be secured between instances.
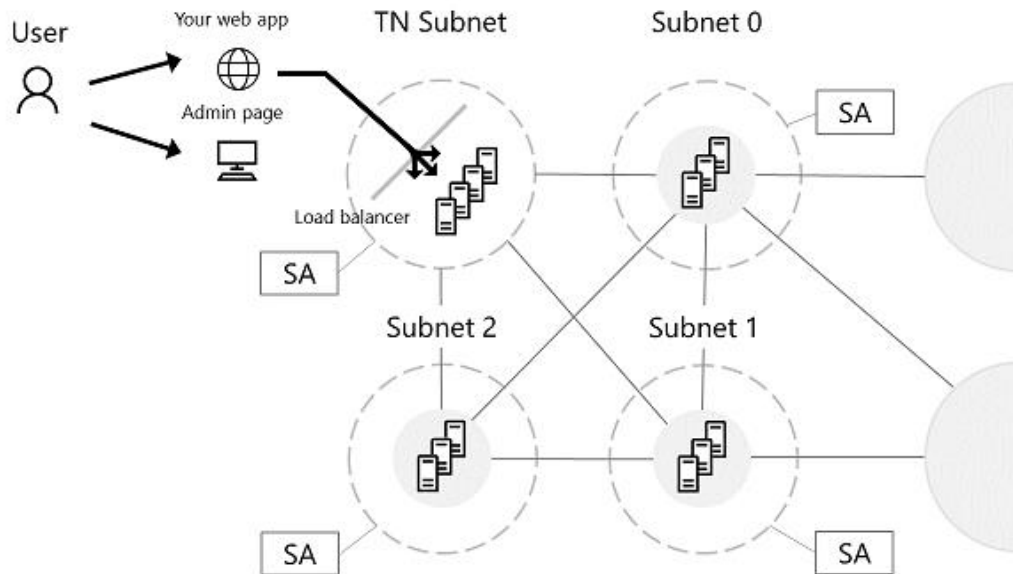
As each deployment was a full new Blockchain, it was complex with unneeded redundancy. The template also imposed limits (e.g. minimum number of participants) and had no facility to stop and restart the network which was necessary in development. Network bridging was expected to be complicated. Given the challenges, we turned to EthereumEx.

EthereumEx is by Eric Maino and Shawn Cicoria from the Microsoft Corp. DX team. They had similar problems and built a comprehensive solution based around a hierarchy of ARM templates.

https://github.com/EthereumEx/ethereum-arm-templates

Their templates have the notion of a "founder member" and "subsequent members". The founder member deployment was like the subsequent member deployments but with the addition of a boot-node and dashboard service. The relevant templates are listed below.

- template.consortium.json
- template.consortiumMember.json

The relationships between the templates are complex having dependencies on child templates to deploy sub-components such as Docker containers to host services and *bash* scripts to inject environment variables. Parameters are injected in a variety of ways and passed between scripts and templates. Trying to identify the runtime value of a parameter in a top-level template can be difficult due to the number of boundaries crossed. It was challenging to understand but provided us the opportunity to extend them to add additional components to support our needs.

EthereumEx provides most of what we needed but required a significant investment to understand and debug due to the complexity of its structure and dependencies.

Understanding Ethereum Network Requirements

When a new Ethereum client wants to join an existing network, it needs to connect to other clients already on the network. By default, Ethereum clients use a UDP-based discovery protocol to find and connect to them. This protocol will first query *bootstrap* nodes that return potential peers for the client to connect to.

To heighten security, we have disabled this discovery protocol by using the Geth flag *–no-discover*. Instead, we use the concept of *static nodes*. Static nodes are a list of existing clients you can create a persistent connection with. We do this by having the initial Utilidex deployment advertise itself as the static node provider for subsequent member deployments. This is done by exposing an API and the endpoint address of this API is injected into member deployments as an ARM template parameter.

Once member deployments have peered, they will synchronize their copy of the Blockchain over a TCP-based *sync protocol.* To encourage local consistency, a member's mining nodes will only be able to synchronise with their local transaction node.

We have ensured that only the necessary UDP and TCP traffic flow is possible on the network by using Network Security Groups (NSGs) to apply port filtering and ACLs. The following is the NSG of a miner node.

2 Inbound security rules

PRIORITY	NAME	SOURCE	DESTINATION	SERVICE	ACTION
100	allow-ssh	VirtualNetwork	Any	SSH (TCP/22)	Allow
102	allow-bootnodes	VirtualNetwork	Any	Custom (TCP/30303)	Allow

3 Outbound security rules

PRIORITY	NAME	SOURCE	DESTINATION	SERVICE	ACTION
101	block-bootnodes	Any	Internet	Custom (Any/30303)	Deny
102	allow-dashboard	Any	Internet	Custom (TCP/3000)	Allow
103	allow-registrar	Any	Internet	Custom (TCP/3001)	Allow

The following table summarizes the Blockchain-relevant ports that we are aware of.

Default Port	Purpose
8545	JSON-RPC
30303	Blockchain Synchronization
3000	Dashboard
3001	Registrar
22	SSH console into Linux OS

VPN (VNet Integration)

Each member deployment contains Ethereum components hosted in virtual machines (IaaS). The enterprise solution adds applications, APIs and databases. A way was needed to include the solution in the deployment.

A number of implementations were initially explored:
- Host the solution on one of the existing VMs

- Add an additional Docker container to an existing VM to host the solution
- Add an additional Linux VM to the VNet to host the solution

As there was a desire to minimise operational overhead and maximise Azure PaaS services it was decided to host the solution components in Azure App Services and Azure SQL Database. This raised an issue because PaaS and IaaS components sit in separate networks and the PaaS API layers need access to port 8545 on Ethereum transaction nodes to make JSON-RPC calls to the client. A way was needed to allow the connection without opening the port to the internet.

A few options were considered:
- Using an App Service Environment.
- Adding application level authentication to create a secure channel over SSL.
- Using a VPN tunnel to create a secure channel.

API Management is also a potential approach that was discussed later.

VPN was chosen for its relative simplicity. It now needed integration into the automated deployment.
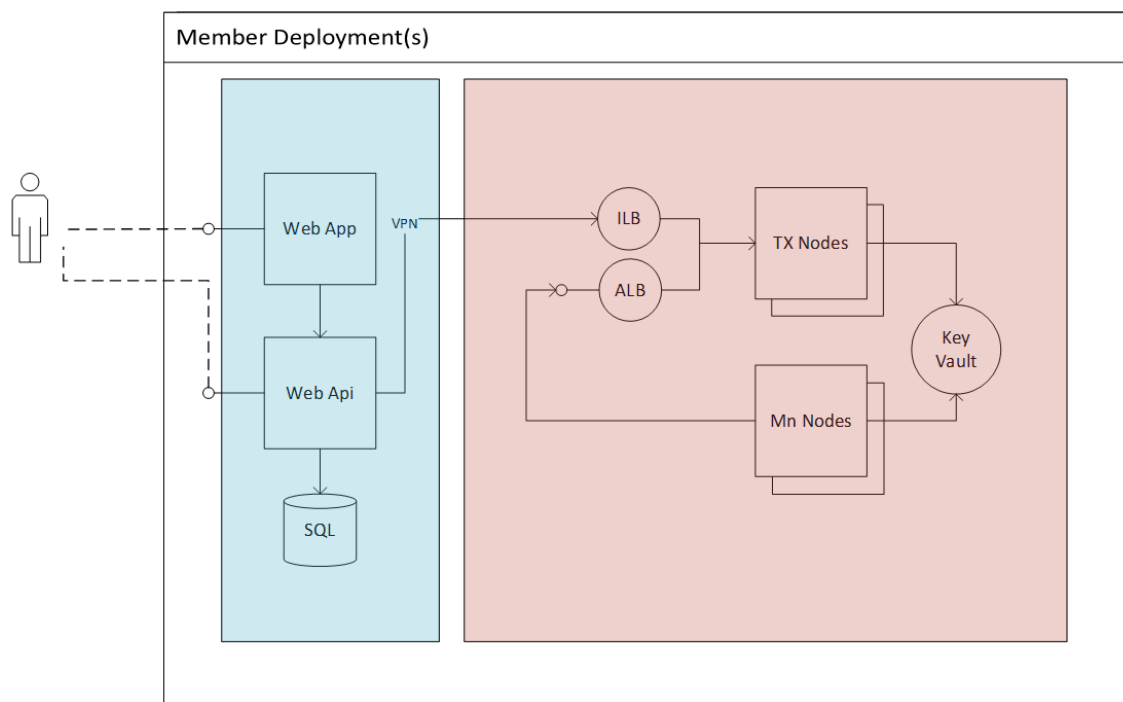
Our first approach was a Point-to-Site VPN from the App Services component to the IaaS VNet following:

> http://www.techdiction.com/2016/01/12/creating-a-point-to-site-vpn-connection-on-an-azure-resource-manager-virtual-network/
> http://www.techdiction.com/2016/02/04/connecting-azure-app-service-to-vnet-using-point-to-site-vpn-using-powershell/

The above required the generation and uploading of certificates to secure the connection. We then found that connecting an App Service to an Azure Virtual Network is now a native capability of Azure.

It is not possible to create a VNet Integration from an ARM template. PowerShell or some other mechanism that can invoke ARM APIs must be used to deploy it. A script was created but VNet integration can take up to an hour to deploy. When deploying VNet integration between a PaaS service and a VNet, other than via the Azure portal, it is also necessary to perform a manual reset via the Azure portal before connectivity is established. It is not possible to do this via the ARM APIs.

IP Whitelisting

We have secured as much as possible via the virtual network. However, Ethereum deployments need to synchronize with each other and perform other operations such as obtaining the list of *static nodes* from the *Initial Deployment*. Specifically, the *static nodes* API is exposed to the internet over port 3001. This is a security vulnerability as one can access this endpoint to retrieve a list of the static IP addresses. A hacker could intercept or interrogate this port to attack the network.

We have discussed possible ways this risk could be mitigated by whitelisting IP addresses though none were implemented within the time we had. Options considered include:

Network Security Groups

Network Security Groups allow you to apply rules to filter network traffic flowing between networked devices. A new rule using 'Source address prefix' to filter incoming traffic could apply an effective IP whitelist. Realistically, these public addresses will be dynamically generated by Azure so the IPs will need to be added programmatically at member deployment or a pool of static addresses is defined for allocation.

IPTables

IP traffic can also be filtered at the Virtual Machine networking layer. For Ethereum on Linux OS, adding IPTable rules to only accept traffic from whitelisted IPs could work but to update them, an agent will be needed to modify the files and restart the network stack on each machine.

Virtual Network Appliance

A Virtual Network Appliance (e.g. Barracuda) could be added to each member deployment. All incoming traffic would go through the appliance and apply the relevant IP whitelisting filters. 3rd party licensing could of course raise the cost of participation significantly.

Automating deployment of Ethereum participant nodes.

Azure Resource Manager

The EthereumEx templates are an excellent starting point for automating the deployment of Blockchain components. To this needs to be added App Services (Web Apps and API Apps), Azure SQL Databases, VNet Integration and a Windows Virtual Machine in the *Initial Deployment* for Smart Contract development. The EthereumEx templates are modularised with the root template orchestrating separate deployments of the right components and networks appropriate to the parameters provided.

We had to take a fork of the templates to incorporate additional applications in the Geth deployment for monitoring and private key management.

Several separate PowerShell scripts were developed to orchestrate the deployment of a new consortium Blockchain, a new member deployment to an existing Blockchain or a minimal test deployment. The scripts were combined into a single script capable of all three purposes.

The distinctions between the three are as follows:

Template	EthereumEx Components	Other Components	Comments
consortium	Registrar Dashboard Tx Node(s) Mn Nodes	Dev VM Web App API App Azure SQL DB VNet Integration	Tx = transaction Mn = miner
new.member	Tx Node(s) Mn Node(s)	Web App API App Azure SQL DB VNet Integration	Requires the IP address of an existing registrar service to connect to.
minimal	Registrar Dashboard Tx Node(s) Mn Nodes	None	

For example, the consortium template will:
- Check if the user is authenticated with Azure and initiate sign-in if they are not.
- Create a new Resource Group for the deployment.
- Initiate the deployment of the "template.consortium.json" ARM template.
  - Capture the results.
- Initiate the deployment of the Jump Box / DevVM.
  - Capture the results.
  - The Jump Box is attached to the VNet created by the template.consortium deployment.
  - The NSG for the VNet is updated to allow RDP traffic to the DevVM.
- Initiate the deployment of the App Services components (Web and API Apps) and SQL DB.
  - Capture the results.
- Initiate VNet Integration between the API App service VNet and the Ethereum VNet.

The templates use parameters files (in the ethereum-consortium-params folder) for values that seldom change and PowerShell arguments for deployment-specific parameters.

Ether Distribution

Ethereum is underpinned by its native currency Ether. Any transaction or operation on Ethereum must be paid for in this currency. In the public Ethereum Blockchain, Ether has real monetary value and so deters members from executing costly operations. In a consortium network, Ether's role is purely technical as it carries no real-world value. We manage the distribution and re-distribution of Ether to allow operations on the Blockchain.

Ether is created in two ways. First, an initial amount can be defined in the genesis.json file of the initial Blockchain account. New Ether is also earned by successfully mining new blocks. In our solution, Ether is first allocated to an initial Utilidex account. This Ether is then distributed for each new member deployment to allow them sufficient funds to operate.

There are 3 ways a member's Ether is affected:
1. Initial distribution from Utilidex.
2. Organic growth due to continuous mining.
3. Redistribution of Ether from Utilidex.

Initial distribution from Utilidex

The amount Utilidex initially distributes to any member could be static and equal to all or calculated for each member based on a heuristic such as a function of the environment (e.g. number of members).

Utilidex can distribute Ether by using a Send Transaction:

```
web3.eth.sendTransaction({to: memberAddress,
                          from: utilidexAddress,
                          value: web3.toWei(1000, "ether")})
```

Organic growth due to continuous mining

Member mining nodes will be organically generating Ether on behalf of their owners as blocks are successfully added to the Blockchain. This will offset some of the member's transaction costs. The following is a rough mathematical model of possible growth in Ether from mining:

Variables
Block mining rate in seconds: r
Number of miners: n
Number of transactions per block: x
Gas price: p
Gas per transaction: g
Mining reward: w
Number of uncles included in block: u

Calculations:
Reward for uncle inclusion: $c = u((7/8)w)$

Seconds to mine block: $t = nr$
assumes each node equal probability of mining any given block

Ether per block: $w + (xgp) + c$
Ether growth rate per second $= (w + (xgp) + c)/t$

Redistribution of Ether from Utilidex

Members will need more Ether distributed if their costs of transactions outpace their miners' ability to resupply their Ether balance. How long before this need arises needs to be considered. Ether redistribution beyond the initial allocation must be factored into the starting amount Utilidex gets to ensure they can supply additional Ether when required.

The solution has an API to allow members to request more Ether. The API runs the Send Transaction operation on a Utilidex transaction node to transfer the currency to the member. Though not implemented, an agent could monitor member Ether balances and trigger a request when the balance falls below a threshold. The following code allows this.

```
web3.eth.getBalance('memberAccount')
```

Developer Jump Box

To manipulate Ethereum for our solution, it is necessary to:
- Compile Solidity Smart Contract code.
- Deploy Smart Contracts to the Blockchain.
- Interact with the Ethereum via a console.
- Have a suitable code editor to create the Smart Contracts and related assets.

A few possibilities were considered:
- A local development environment using Windows or Linux.

- Use an existing or new transaction node for the purpose. This will be Linux.
- An Azure virtual machine hosting a development environment in Windows or Linux.

A local development environment is most convenient but would need port 8545 on an Ethereum transaction node opened to the internet to allow interaction. This is unacceptable due to security concerns.

Using an instance of the transaction node was ruled out to maintain Separation of Concerns in the solution.

Creating a virtual machine in Azure on the same VNet as a transaction node allows remote connection via a secured SSH session. The virtual machine can then connect to an Ethereum client securely on port 8545 on the local network within the VNet. A virtual machine was thus incorporated into Utilidex's *Initial Deployment*. The machine can be started only when necessary to save costs.

The machine hosts the following components:
- Visual Studio Code (Integrated Development Environment)
- Git (Source Control)
- Truffle (Solidity compiler and Smart Contract deployment tool)
- Test RPC (Local Blockchain test environment)

To automate the deployment of the virtual machine and its numerous components and dependencies, the following asset was developed:

https://github.com/utilidex/projectx-flexchange-dev-vm

The workshop took a local Vagrant (https://www.vagrantup.com/) virtual machine and moved it into an ARM template with PowerShell scripts to install the components and dependencies. There is a version of the template that sets up networking to attach the virtual machine to an Ethereum transaction node by retrieving VNet information from the deployment. The scripts were also modified to allow RDP traffic to the machine through a Network Security Group.

The above is based on earlier work (https://github.com/mormond/TruffleDevBox) which itself was based on Thomas Conte's (https://github.com/tomconte/TruffleDevBox).

Ethereum Account Private Key Management

Our solution programmatically creates Ethereum accounts. When accounts are created, cryptographic keys are stored on the virtual machine running the Ethereum transaction node client. These keys need to be backed up against virtual machine failure as its storage isn't persisted.

Unfortunately, Azure Key Vault does not support Elliptical Curve Cryptography (ECC) keypairs, which is used by Ethereum. Developers will need to implement their own signing logic outside of Key Vault. It can however be used as a secure store of secrets. For expedience, we used a secured Azure Blob storage account for the same purpose.

A local service is put on each transaction node that periodically archives the entire local key store to storage. The key store can now be restored to the last backup. There will be a delta between the time of the last backup and the time of a failure. This delta is configurable to needed recovery thresholds. A view of the storage holding key store backups is as below.

Monitoring

Ethereum Blockchains with small numbers of participants, each with a small number of active nodes, were observed to either start diverging (have significantly different block heights) or fail to be reported in the EthStats dashboard. E.g. below we can see there are nodes at or around 10840 but also at 9054, 7734, 524, etc. Those nodes tend not to recover and need to be restarted.
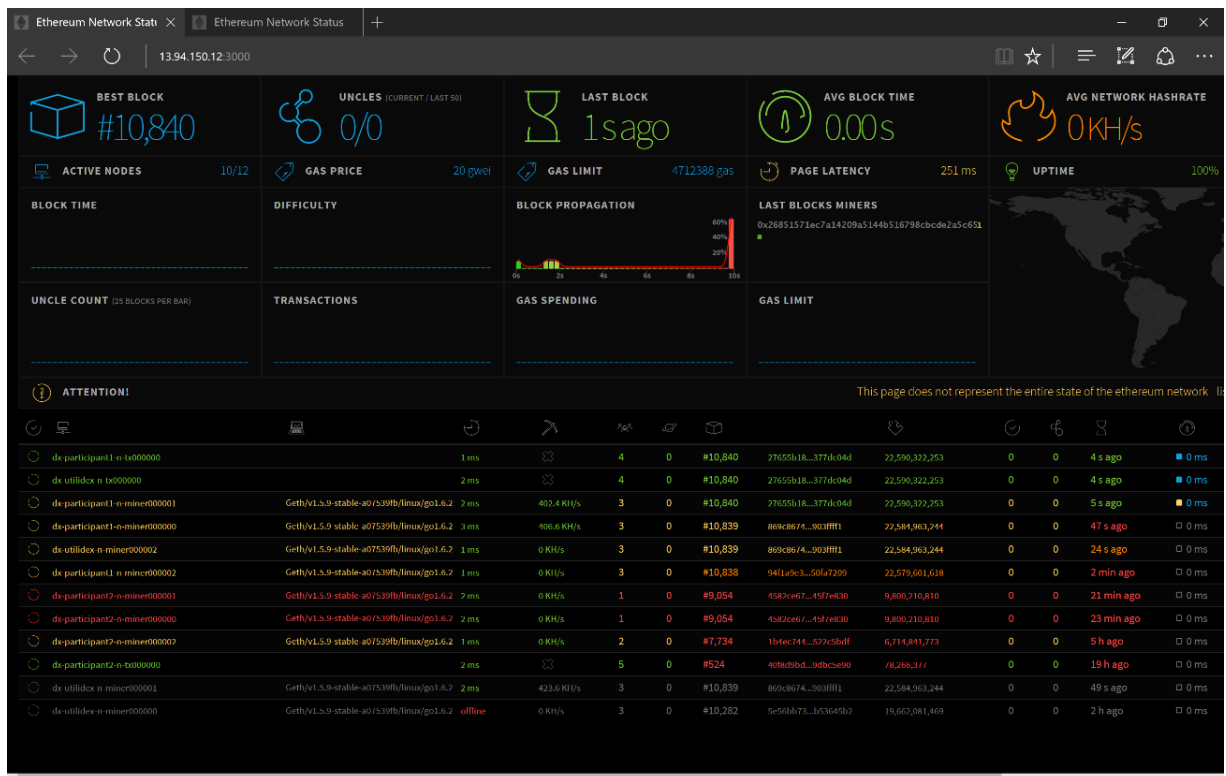
To address this issue, we monitor the block height of each individual node to detect stalls. A local service on each node periodically checks the block height for progress. If there is none, or an error is encountered, it restarts the node.

```javascript
try {
    var currentBlockNumber = web3.eth.blockNumber;
    console.log("Current block number " + currentBlockNumber);
    if (currentBlockNumber > previousBlockNumber) {
        console.log("Everything looks ok!");
        previousBlockNumber = currentBlockNumber;
    } else {
        console.log("Block number appear to have stalled, let m
        restartEth();
    }
} catch (error) {
    console.log("Exception querying JSON-RPC: " + error);
    restartEth();
}
```

This code made it necessary to fork the ARM templates from EthereumEx so it could be deployed to all nodes. The ARM template maintains the link to the EthereumEx Docker images for each of the following:

- Dashboard application
- Transaction node
- Miner node

We did not investigate the root cause of node divergence due to time constraints. In a public proof-of-work Blockchain this is an unlikely to be an issue because there are typically very large numbers of participants and those participants are motivated to ensure their nodes are in a healthy state to continue earning rewards. In a consortium Blockchain, it presents a real issue as participants are not as invested in the health of their nodes because Ether carries no real-world value. A mechanism for monitoring and maintaining the health of the nodes must be created.

Developing a Smart Contract that minimizes risks while maximizing value

Smart Contract Architecture

The aim of our Smart Contract was to allow participating members to create new agreement 'proposals'. A Smart Contract is the interface to the Ethereum Virtual Machine (EVM). We implemented our Smart Contract in the Solidity language, the most mature and popular for Ethereum. The Solidity language is very similar to JavaScript, however the semantics are considerably different. It is purposefully restricted to deter the developer from writing excessively costly logic such as array iterations.

The EVM is not appropriate to run complex business logic. We therefore designed each function to be as atomic and as self-contained as possible. Thus, we minimised the surface area of our Smart Contract. The design moves complex data shaping to other solution components, with the Smart Contract performing simple state changes that are implicitly tracked by the Blockchain.

Exceptions

Exceptions thrown in Solidity are not easily returned to the caller. The primary function is to unravel the current call as if it never happened. The Ethereum's eventing mechanism could be used to throw an ExceptionEvent to notify the caller but this was not implemented.

The Solidity language is a type-strict language. It has many common primitive value types such as *int, string* and *bool*, has support for *structs* and special types such as *address* and *mapping*. As available call stack memory is small, care must be taken around the numbers of function arguments used and local variables created. Lightweight types such as *uint8* and *byte* can save valuable memory.

The Smart Contract

The following is part of the implemented Smart Contract.

```
/**
 * Allows the contract owner to create a new proposal which must be signed
 * by all affiliated parties in order to be considered complete.
 **/
function CreateProposal(string id, address pA, address pB) public
{
    // Only allow contract owner to create proposals
    if(msg.sender != owner)
        return;

    // Only allow unique
    if(IdExists(id))
        return;

    // Create Proposal
    var proposal = Proposal({
        _id: id,
        _partyA: pA,
        _partyB: pB,
        _partyASigned: false,
        _partyBSigned: false,
        _initialised: true
    });

    // Store Proposal
    proposals[id] = proposal;
}

/**
 * Allows parties affiliated with a particular proposal to agree
 * to the terms by signing it.
 **/
function SignProposal(string id) public
{
    if(IdExists(id))
    {
        // Get the relevant proposal
        var proposal = proposals[id];

        // Check caller is a affiliated party and if so apply their signature
        if(proposal._partyA == msg.sender) {
            proposals[id]._partyASigned = true;
        } else if(proposal._partyB == msg.sender) {
            proposals[id]._partyBSigned = true;
        }
    }
}
```

Off-chain Hashing of Assets

The solution minimizes data in a Blockchain transaction so assets are stored outside of it. Assets could be a document or other digital data associated with an agreement. What is needed is a way to verify the authenticity of these assets that will live outside of the Blockchain. To do this, we calculate a hash value for the asset and this is what the Blockchain transaction will store.

The workshop designed the following process:

1. Whenever off-chain assets are generated, Utilidex will create a SHA256 hash of that asset and provide this as an argument to the Smart Contract call. Also provided will be the hash algorithm identifier (SHA256) and the location of the asset off-chain.
2. Each agreement participant can retrieve an asset from the location identified and recalculate the hash. If their hash matches the one stored on the Blockchain, then the asset has not changed. The participant can then rely on the asset to inform their assessment of the agreement.
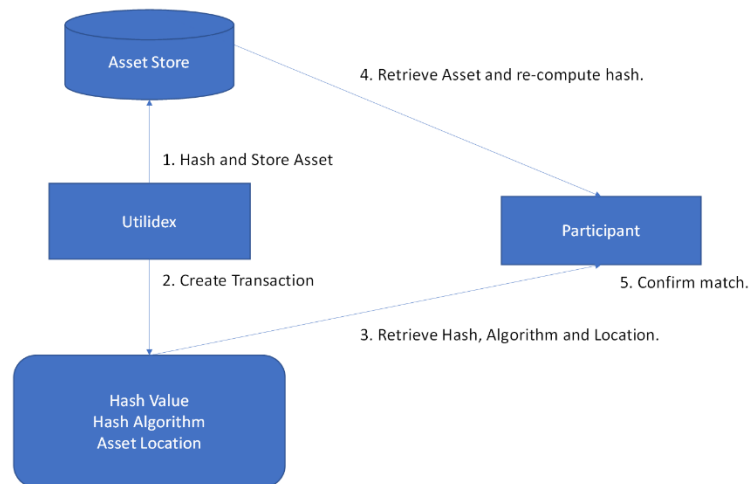
The assets themselves could be moved literally anywhere. Upon receipt of an asset, it can still be validated for authenticity by referring to the Blockchain transaction and comparing a recalculated hash using the algorithm specified for that asset. Authenticity is established if the hash matches that of the original regardless of how the asset came to be with the current owners.

Our Smart Contract was updated to implement:
1. Functions that take Hash Value, Algorithm and Asset Location arguments.

```
/**
* Allows the contract owner to create a new proposal which must be signed
* by all affiliated parties in order to be considered complete.
**/
function CreateProposal(string id, string hash, string algo) public
{
    // Only allow contract owner to create proposals
    if(msg.sender != owner)
        return;

    // Only allow unique
    if(IdExists(id))
        return;

    // Create Proposal
    var proposal = Proposal({
        _id: id,
        _initialised: true
        _hash:hash;
        _algo:algo;
        _memberCount:0;
    });

    // Store Proposal
    proposals[id] = proposal;
}
```

2. Functions to return these same values for validation use.

```
/**
* Allows parties affiliated with a particular proposal to get hash algo
**/
function GetAlgo(string id) returns (string algo)
{
    if(IdExists(id))
     {
          // Get the relevant proposal
          var proposal = proposals[id];

          // Check caller is a affiliated party and if so apply their signature
          if(proposal._parties[msg.sender]==address(0x0)) {
               algo= null;
          }
          else{
               algo = proposal._algo;
          }
     }
     else{
          algo = null;
     }
}

/**
* Allows parties affiliated with a particular proposal to get hash
**/
function GetHash(string id) returns (string hash)
{
    if(IdExists(id))
     {
          // Get the relevant proposal
          var proposal = proposals[id];

          // Check caller is a affiliated party and if so apply their signature
          if(proposal._parties[msg.sender]==address(0x0)) {
               hash= null;
          }
          else{
               hash = proposal._hash;
          }
     }
     else{
          hash = null;
     }
}
```

Actual calculation of the hash has to be done off-chain which simplifies the Smart Contract. The Ethereum Geth client automatically signs the transactions that include these hashes with the private key of the calling account as they are incorporated into the Blockchain. The hashes thus become part of the immutable distributed ledger.

Smart Contract DevOps

There is a need for automation of Smart Contract Build and Release using Truffle. The steps are:
1. Get Smart Contract code from a Git repository.
2. Compile it with Truffle.
3. Release and deploy it using Truffle migrate.
4. Manage the account used for Smart Contract deployments.

When a user commits changes to Git for the Smart Contract, Visual Studio Team Services (VSTS) *continuous integration* triggers the build process. PowerShell scripts were created and run on the Jump Box for each of the above steps. The VSTS build process is used to orchestrate invocation. PowerShell scripts were chosen because VSTS hosted build agents cannot compile Smart Contracts due to lack of Truffle tooling and dependencies at the time of the workshop. Customized Build Agents were expected to be too complex to set up within the time available.

We implemented the process up to Truffle migration of the Smart Contract. Account management for actual Smart Contract deployment is next. The PowerShell scripts are located at:

https://github.com/Utilidex/projectx-flexchange-smart-contract/tree/master/build

Building a pattern for an enterprise solution to integrate with Ethereum.

There are commercial considerations when building a solution on Ethereum. The following were worked on during the development workshop.

- There must be freedom to develop proprietary IP that can then be monetized.
- It should leverage existing development processes to maintain productivity.
- We must translate technical Ethereum concepts to solution capabilities for the business.
- Ethereum-specific application integration and management will need to be addressed.
  - Smart Contracts must be managed and versioned for multiple workflows.
  - There must be a way to call into the blockchain JSON-RPC APIs from the application.
- Billing/monetization models needed to be considered.

Protecting Proprietary IP

In choosing an open-sourced Ethereum Blockchain, the impact to proprietary IP needed to be assessed. We reviewed the components and their related libraries for licencing. Definitions can be found at https://opensource.org/licenses.

GETH Binaries: GPL3
GETH Libraries: LGPL3
WEB3.JS: LGPL3
NETHEREUM: MIT
TRUFFLE: MIT

Our non-legal understanding was that using Ethereum components should not require Utilidex to publish any of their own source code if we avoid inclusion of GPL source. However, to be certain, we introduced a 'firebreak' via an API layer. Utilidex proprietary applications thus do not use any Ethereum-related components but will interface via REST-based APIs to those that do. The APIs then contain the integration calls into Ethereum. This aligns well with our approach to keep complex processing logic outside of the Blockchain.
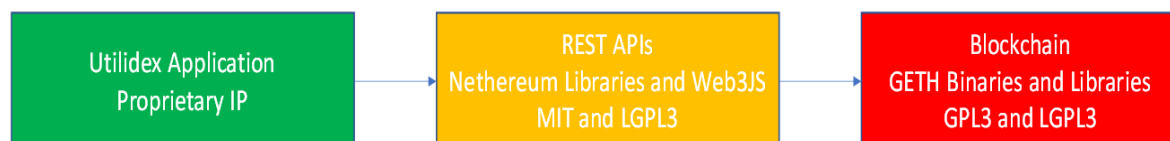
We also assume that any Smart Contract deployed on the Blockchain would be publicly readable and so would avoid embedding any proprietary IP in the Solidity code.

Maintaining Developer Efficiency

Utilidex has migrated to and enjoys Azure Platform-as-a-Service offerings. We thus maintained and maximized the use of these. Virtual Machines (IaaS) were limited to Ethereum needs (transaction and miner nodes, Registry and Jump Box instances). We incorporated the following services from the Azure PaaS catalogue. Additionally, all team collaboration and source repositories were stored on Visual Studio Team Services (VSTS).

Web Apps	API Apps	SQL Database	Active Directory	Key Vault (In Design)	Load Balancer	Visual Studio Team Services

Use of PaaS components allowed us rapid solution development with on-demand redeployments of applications, services and databases. The following shows a mapping of Azure PaaS components used to our solution architecture.
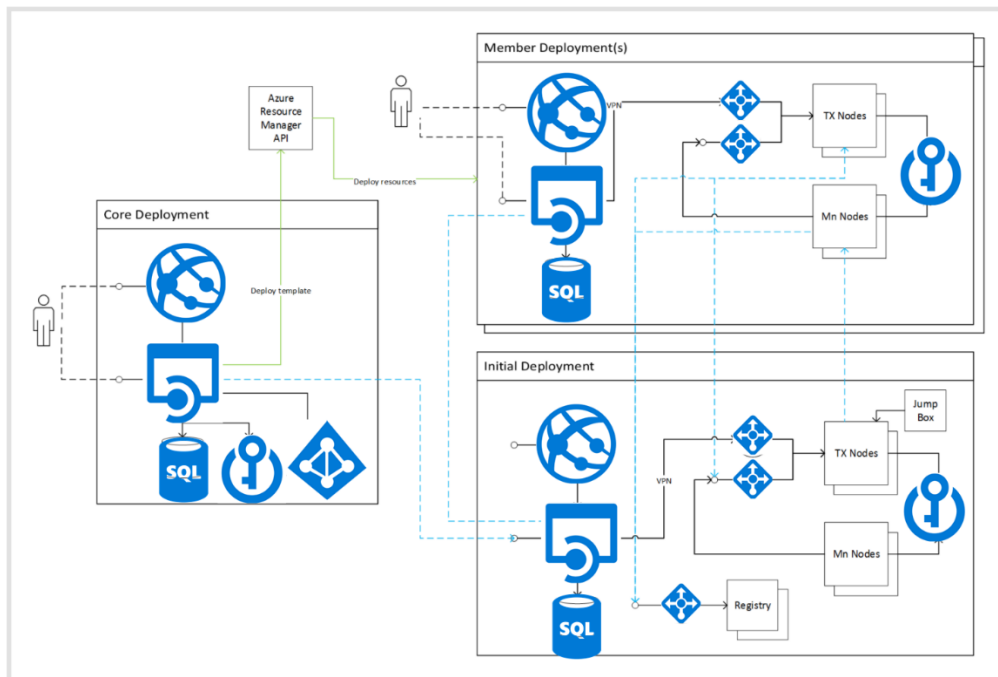
Bridging between Technology and the Business

Design decisions included how users would ultimately transact on the Blockchain. There is no authenticated security principal on the Geth client. An Ethereum account is simply an address with a stored private key protected by a passphrase. The passphrase is used to unlock the account to create transactions on the Blockchain.
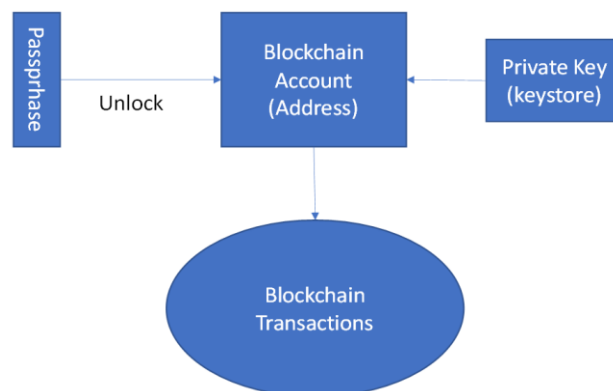
We used Azure Active Directory (AAD) to provide enterprise-grade security principals. These protect access to the application and API layers and provide us with an identity token. Authenticated users

then create Ethereum accounts for their organization. We envision role-based restrictions to enforce user access to accounts by leveraging AD groups as a future extension.
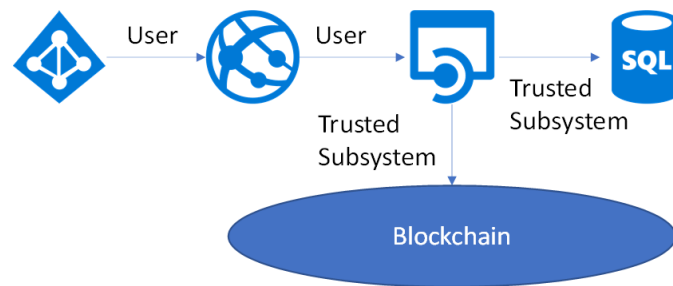
By separating individual users from organisational Ethereum accounts, AAD provides user authentication and authorization up to the Ethereum boundary. Mapping the users to Ethereum accounts then completes the flow allowing a user to initiate a Blockchain transaction on behalf of their organization. Access control to Ethereum nodes can be enforced via application logic to manage privileges e.g. to create transactions.

Application Integration

Smart Contract Versioning

A deployed Smart Contract cannot be changed. Smart Contract updates needs deployment of a new Smart Contract. Each Smart Contract has a unique Ethereum address (e.g. '0x79966a9a613cca85e7149a193b82f97b69e4c609'). Thus, in long running workflows, more than one version of a Smart Contract could be in use. In our basic solution, the Smart Contract is called in 3 instances.

- When a new agreement is proposed. The contract records the participants.
- When each participant agrees to the proposal. The contract records each agreement.
- When a check is made for completion of the agreement. The contract reports its completion.

It is possible that a Smart Contract is updated after a proposal is made. Participants of the proposal must use the version that originally created the proposal. All new proposals run on the updated contract. So, we need to track Smart Contract versions with the following rules.

- For any new proposal, the latest Smart Contract version will be used.
- Once used, all further activity in the workflow will use the same Smart Contract.

To track a Smart Contract, we store its Ethereum address and its Application Binary Interface (ABI). The ABI is a JSON string that describes the Smart Contract's methods, arguments and outputs. For Utilidex, when a Smart Contract is deployed, its Address and ABI will be updated in the Core Deployment database. This was a manual step during the workshop as the DevOps pipeline was still being developed.

Nethereum

To interact with the Ethereum Geth client, we make JSON-RPC calls into default-port 8545 exposed by the client on a transaction node. We call from the *Member Deployment*'s API layer, tunnel through VNet integrated VPN, to reach the endpoint. This Geth endpoint is unprotected and must not be exposed to the internet, hence the use of VNet Integration between the API App (PaaS) and the transaction node virtual machine instance (IaaS).

As we were developing in C# using ASP.NET Web (MVC) and API Apps, the integration library chosen was Nethereum.

> "Nethereum is the .Net integration library for Ethereum, it allows you to interact with Ethereum clients like geth, eth or parity using RPC."
> https://github.com/Nethereum/Nethereum

We first use the NuGet Package Manager in Visual Studio to install the library.

> PM > Install-Package Nethereum.Portable -Pre

Steps to make a Smart Contract call:
- Obtain the Smart Contract address and ABI.
- Obtain the function signature on the Smart Contract to be called.
- Unlock the Ethereum account making the call. This needs the account's passphrase.
- Make the call.

The following extract shows the code leading to a Smart Contract call.

```csharp
// STEP 1: Obtain the contract ABI

abi = db.GetContract(agreement.ContractID);

if (string.IsNullOrEmpty(abi))
{
    Contracts coreAPIClient = new Contracts(new BChainCoreAPIs());
    ClientCredential clientCredential = new ClientCredential(
        Startup.ClientId,
        Startup.AppSecret);
    AuthenticationContext authContext = new AuthenticationContext(Startup.Authority);
    AuthenticationResult result = await authContext.AcquireTokenAsync(
        Startup.CoreAPIRecourceId,
        clientCredential);

    try
    {
        // Get the contract ABI from the core instance and save a copy locally
        abi = coreAPIClient.Get(agreement.ContractID, result.AccessToken);
    }
    catch (Exception ex)
    {
        Debug.WriteLine(ex);
    }

    if (string.IsNullOrEmpty(abi))
    {
        success = false;
    }
    else
    {
```

```csharp
            success = (db.AddContract(agreement.ContractID, abi) > 0);
        }
    }

    // STEP 2: Get the function address to call on the smart contract

    if (success)
    {
        try
        {
            func = web3.Eth.GetContract(
                abi,
                agreement.ContractID).GetFunction("CreateProposal");
        }
        catch (Exception ex)
        {
            Debug.WriteLine(ex);
        }

        success = (func != null);
    }

    // STEP 3: Unlock the account so we can call the smart contract

    if (success)
    {
        string passphrase = db.GetAccountPassphrase(agreement.OriginatorAccount);

        try
        {
            success = await web3.Personal.UnlockAccount.SendRequestAsync(
                agreement.OriginatorAccount,
                passphrase,
                120);
        }
        catch (Exception ex)
        {
            success = false;
            Debug.WriteLine(ex);
        }
    }

    // STEP 4: Make the smart contract call

    if (success)
    {
        object[] args = new object[] {
            id,
```

```
            agreement.OriginatorAccount,
            agreement.CounterSigAccount };

        try
        {
            // Call the CreateProposal function on the smart contract
            await func.SendTransactionAsync(agreement.OriginatorAccount, args);
        }
        catch (Exception ex)
        {
            success = false;
            Debug.WriteLine(ex);
        }
    }
```

We encountered issues where Nethereum was updated but had dependencies that were not of the latest version resulting in reference errors. To resolve, we needed to add the following to the web.config file.

```
<assemblyIdentity name="crypto" publicKeyToken="0e99375e54769942" culture="neutral" />
<bindingRedirect oldVersion="0.0.0.0-1.8.1.2" newVersion="1.8.1.0" />
```

Billing Considerations

Utilidex Monetization Options

We discussed possible billing models for the solution.
- Per-API call via the use of API Management tracking.
- Business-led where a rate is decided based on business value generated.
- Flat-rate membership charges with additional revenue from value-added services.

Cost of Membership

There is a charge to each participating member for the deployment of the Ethereum member components into their Azure subscriptions. These are Azure running costs for the *Member Deployment* components and their related IT overhead. Members must include these in calculating the cost of participation.

Further billing discussion has been deferred to Utilidex.

# Protecting access to the solution with user-level authentication and authorization

This section describes the architecture of user-level authentication to access services to the underlying Blockchain. The diagrams represent the logical *authentication flow* through the application and are not physical representations of components as deployed.

Azure Active Directory Protection

As Implemented

Internet-reachable endpoints in the solution are exposed by the ASP.NET MVC Web Apps and API Apps hosted on Azure App Services. A core deployment will have a Core Web/API App combination and each member deployment will have its own Member Web/API App pair.

Access to Web Apps will be secured by user login. Only the Web Apps can call the APIs. The API connects to the Ethereum endpoints via App Services VNet Integration from the App Service. This is how authentication works:
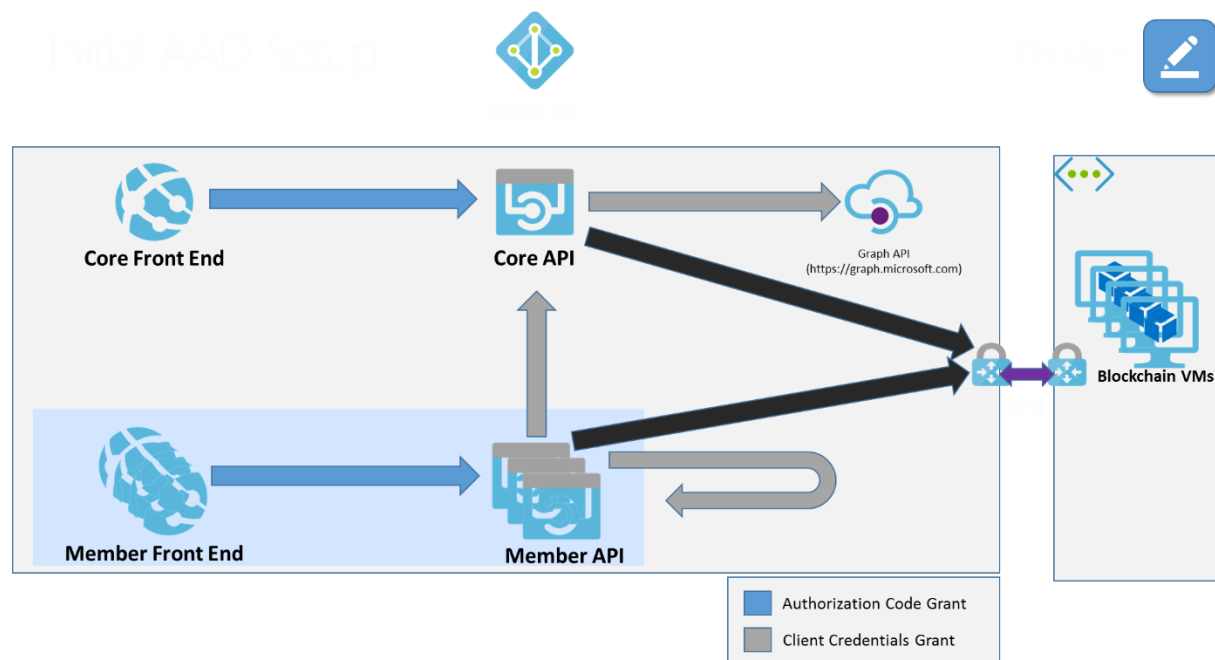

- Web Apps use an OAuth flow called *Authorization Code Grant* (Blue Arrows) requiring authentication with Azure Active Directory (AAD). Calls to the APIs are then done with a token from user authentication.
- The calls between APIs use an OAuth flow called *Client Credential Grant* (Grey Arrows) – which is not based on a user identity but the identity of the application.

All calls from APIs to Ethereum are unauthenticated since the Geth client does not use user security tokens. The RPC endpoint is protected by a VPN set up via App Services VNet Integration and is not exposed to the internet.

Future Elaboration

The authentication flow as implemented is a simplified form of what could be used in production. The same Application Registrations were used for all the Member Web/API App pairs despite them being different organisations. The following would be a possible extension.
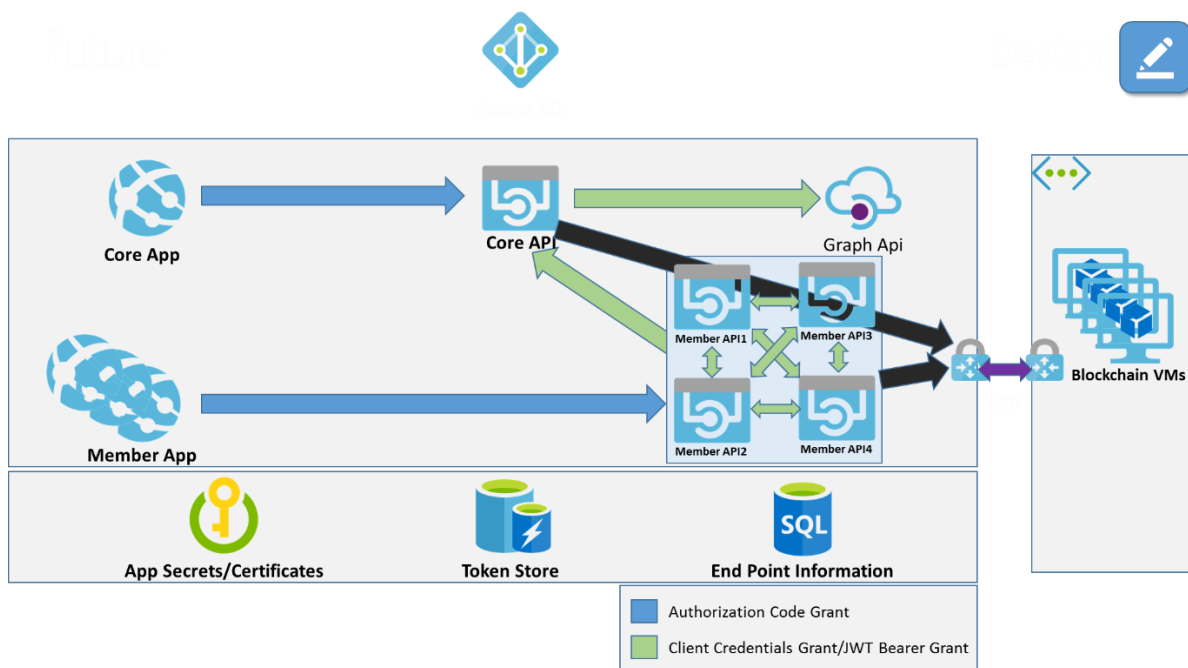
Each Member Web and API App pair has its own App Registration in Azure AD. This allows an administrator to deny specific members calling APIs to update the chain. This provides more granular access control.

There are other changes that should be considered:
- The App Secrets for all the Web and API Apps are stored in *web.config*. This will not scale with each service having its own registration. As configuration files can leak secrets, it is best to move these to more secure storage.
- The Token Cache for the authentication libraries uses session state. This will not scale and will also cause inconsistencies. E.g. The ASP.NET cookie could track a logged-in user, but the token may not be available. Use Redis Cache Service for the Token Cache.
- The calls between APIs could be changed from *Client Credential Grant* to *JWT Bearer Grant* (on-behalf of). This allows impersonation of the user through the APIs, enabling better access control.

Observations
Our thoughts on some challenges encountered during the 4 days in February:

ARM scripts must also be versioned
It is important to consider the versioning of the ARM templates and their dependencies. Taking a specific version of the parent template could result in breaking changes caused by its dependencies being updated on GitHub. In production, it would be necessary to control all the versions being used, either by forking the repositories and/or using contentRootOverride to fix dependencies to specific versions.

Linux knowledge requirement
Blockchain projects run primarily on Linux OS. There are development tools that allow you to write and deploy Blockchain code from a Windows OS but the live Blockchain clients will most likely operate on Linux OS. A basic understanding of the following Linux topics is helpful:
- Command line navigation, file manipulation and user management.

- Process model and networking.
- *Bash* scripting.
- File system structure and operations.

Azure App Services and other PaaS components were very reliable

By having Web and API Apps on App Services, we could focus on business logic and Nethereum integration without worrying about infrastructure. This allowed us to stand up 3 different sets of deployments easily. One each for isolated application/API development, an integration environment internal to Microsoft and the final deployment within the Utilidex subscription. App Services worked flawlessly throughout.

Bletchley template is useful for testing

In the early stages of the workshop, getting a stable Ethereum instance was a challenge since deployments needed to be modified constantly and this sometimes resulted in non-functioning Blockchains. Given that the web and API application development relied on having a deployed Smart Contract on a working Ethereum Blockchain, it could have been a blocker. We found the Bletchley template very useful to get an instance up and running so that the Smart Contract could be deployed to unblock the necessary integration development.

Conclusion

Blockchain is constantly improving as a technology. Through this partnership, Utilidex and Microsoft have gained valuable experience with Ethereum. With the work that was done, we found that while Blockchains provide a transformational ability for untrusted parties to work together, there are still many other capabilities that are needed for acceptable enterprise use. These features were lacking within the Ethereum implementation at the time, but the Microsoft Azure platform had services that credibly improved the Ethereum environment to meet those requirements.

Utilidex found the collaboration very valuable in providing a practical grounding in the risks and complexities involved in incorporating this technology in a commercial offering. It is now a matter of finding the right business model.

" It's really important for us to look at how flexible energy is transacted between parties in the future. As our world continues its exciting journey to renewable sources, and where customers can now produce as well as consume, we find ourselves needing new operating models in the energy market and Blockchain may very well have a big part to play" Richard Brys, CEO Utilidex.

For a summary of the experience and impact, please watch this video.