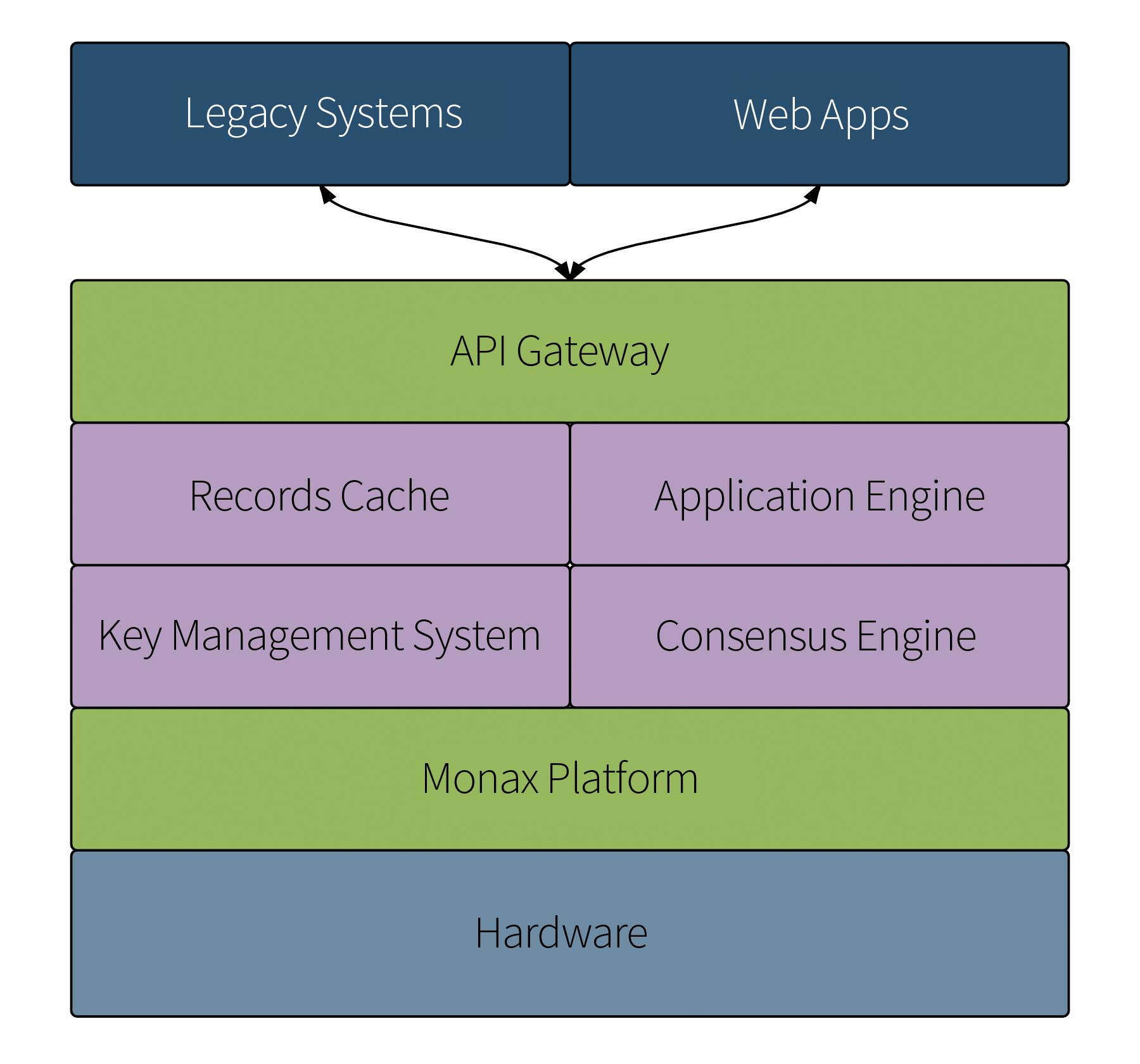
**Introduction**

Monax Application Platform is built from the ground up to provide a logical base for developers and DevOps to build, test, and operate ecosystem applications. It is designed to support multiple blockchain nodes connected to multiple blockchain networks with different smart contract interpreters, all seamlessly connected to other microservices necessary to build, test, and run the ecosystem application, such as key management systems and distributed data lakes. Ecosystem applications are the next generation of process management tools and the Monax platform is the gateway to building sophisticated enterprise-grade ecosystem applications.

In Monax architecture, the application and the consensus engine run in separate processes on the operating system, and communicate with each other via a simple socket protocol. This architecture is similar in spirit to that which has been used to serve websites on the internet for decades, namely, by having an http server and the actual web-application in separate processes: requests are received by the web server, filtered, and forwarded to the application. Here blockchain design is no different, except that the server component is replicated across many nodes and achieves consensus on a request before it is forwarded to the application state. Hence, we can easily write blockchain applications in any programming language, while the networking, consensus, and blockchain storage is all handled under the hood.

|  |  |  |
| --- | --- | --- |
|  | Monax (burrow) | Ethereum (geth) |
| Consensus | Tendermint Consensus Engine Proof of Stake PBFT Algorithm | Ethereum’s Consensus Engine Proof of Work |
| Interfaces to Client | ECDSA ed22519 curve (same as PGP, SSH, SSL, etc.) | ECDSA secp2k1 curve (same as Bitcoin, etc.) |
| Key Types | Built to Specification EVM | Built to Specification EVM |
| Permission layer | Yes | No |

**Overview of an Ecosystem Application**

The above diagram provides a high level overview of the different pieces of an ecosystem application. The green boxes are part of the Monax technology stack while purple represents external services managed by the CLI.

### **Chains**

monax chains is the gateway to unlocking the power of permissionable, smart contract optimized blockchains – of which our [Burrow](https://monax.io/platform/db) client is the industry leader. It exposes a range of options for developers to create, administer, and operate blockchains of various varieties.

### **Packages**

monax pkgs is the gateway to unlocking the power of smart contract systems. It exposes a range of options for developers to create, build, test, and deploy complex systems of smart contracts to their monax chains.

### **Keys**

monax keys is the gateway to the common signing pipe that monax exposes. It exposes a range of developers to quickly get up to speed with prototyping their ecosystem applications. It also provides the reference API implementation for wallet-makers, and other, more advanced, signing solutions to satisfy in order to work seamlessly with the monax platform.

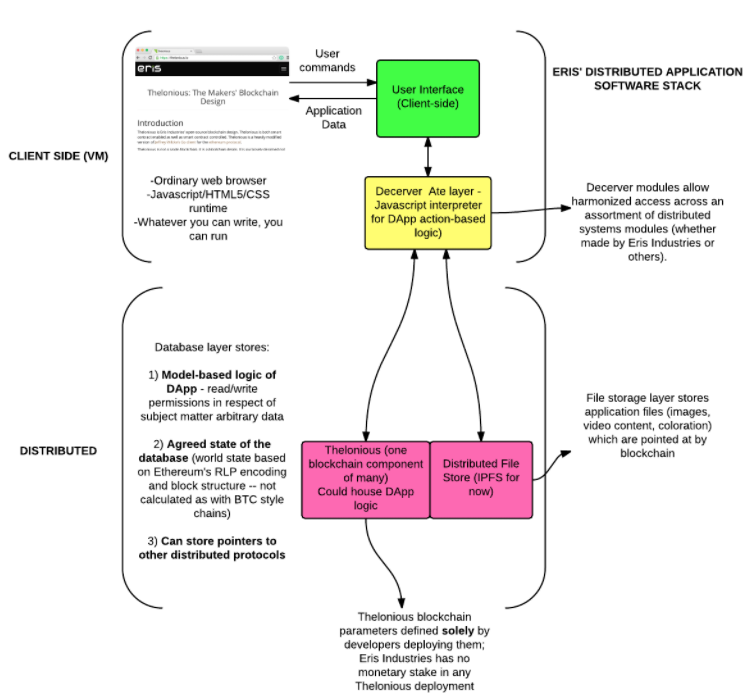
### **Files**

monax files is the gateway to unlocking the power of distributed data lakes. It exposes a range of functionality for working with content-addressable and distributed data management systems.

### **Services**

monax services is the gateway to unlocking services which your application needs to rely upon. It exposes a range of base operational functionality for integrating microservices into your ecosystem application.

**Application Structure**



**Prerequisite**

* Install Docker.
* Install Go.

go get github.com/monax/monax/cmd/monax

**Overview**

The monax tool is centered around a very few concepts:

* services — things that you turn on or off
* chains — develop permissioned chains
* pkgs — our smart contract tool chain
* keys — wrapping of our key management tooling

These concepts provide the core functionality of what we think a true smart contract application platform requires.

To get started using monax to see what the tooling can do and how it can help your development patterns for smart contract applications, please go through the documentation.

**Architecture of the Tool**

monax is mostly an opinionated wrapper around Docker's API. We have found that running applications locally which require sophisticated installation paths and/or complex configuration work best when used from Docker's container based system.

Each of the concepts listed above is described in a bit more detail below.

**Services**

Services are "things that you turn on or off". Examples of services include:

* a PGP daemon
* an IPFS node
* a Bitcoin node
* an Ethereum node
* a Tendermint test chain node
* BigchainDB service
* ZCash node

Services work from a base of **service definition files**. These files are held on the host in the following location: ~/.monax/services. Service definition files tell monax how a docker container should be started. The specification for service definition files is located [here](https://monax.io/docs/specs/services_specification).

To see the various ways in which monax can interact with services, please type:

monax services

**Chains**

Chains are an opinionated toolchain around permissioned chains. They can be most easily thought of as your "develop" branch for chains. In other words, if you need to work **on** a permissioned chain, then it is best to use monax chains. Chains hardcode most of the service starting criteria, but still allow for some flexibility as to how chains are worked with.

To see the various ways in which monax can help you develop chains, please type:

monax chains

**Pkgs**

Pkgs are an opinionated toolkit to help you deploy and test your smart contract packages on both permissioned and unpermissioned blockchain networks.

monax pkgs is a package manager to deal with contracts. The package manager is a yaml based automation framework which makes it trivial to deploy and test your smart contract systems. The specification for monax:jobs definition files is located [here](https://monax.io/docs/specs/jobs_specification).

Pkgs give you access to test your smart contracts both against "throwaway chains" which are one time use chains that are needed for the sole purpose of testing smart contract packages, as well as existing blockchain networks.

To see the various ways in which monax can help you develop smart contract applications, please type:

monax pkgs

**Keys**

Keys is an opinionated toolchain around [monax-keys](https://github.com/monax/keys). Please note that this concept of the monax platform is **for development only** and should not be used in production because it has not been fully security audited **and we do not plan for it to be**. In production the keys service should be replaced with your audited security system of choice.

To see the various ways in which monax can help you manage your various key pairs, please type:

monax keys

**Directory Structure**

Created by monax init in $HOME directory:

├── .monax/

│ ├── monax.toml

│ ├── apps/

│ ├── bundles/

│ ├── chains/

│ ├── account-types/

│ ├── chain-types/

│ ├── keys/

│ ├── data/

│ ├── names/

│ ├── scratch/

│ ├── data/

│ ├── languages/

│ ├── services/

│ ├── keys.toml

For installation and writing your first application

monax is the CLI ecosystem application platform built by Monax.

There are four steps need to get moving with Monax:

1. **Install** the platform.
2. **Roll** the blockchain base for your ecosystem application.
3. **Deploy** your ecosystem application using smart contract templates and a simple, web-based user interface.
4. **Integrate** your ecosystem application with a web server or other microservices.

## Step 1. Install the Monax Platform

**Dependencies**: monax has 2 dependencies: [Docker](https://www.docker.com/) and for macOS and Windows only [Docker Machine](https://docs.docker.com/machine/). Docker is a run anywhere container solution which makes development, deployment, testing, and running of ecosystem applications a breeze and Docker Machine allows you to run Docker on remote machines. We do not currently support Docker for Mac/Windows as they are still in beta.

Currently we consider the most workable setup to be (what our tests consider authoritative) with these operating system and dependencies’ versions:

* Host OS = UBUNTU:16.04
* Docker = 1.11.2
* Docker Machine = 0.8.1

We are working steadily toward making monax available for a wide variety of host environments.

At the current time, monax requires docker version >= 1.9.1 and docker-machine version >= 0.4.1. We do not test against older versions of Docker and Docker Machine: monax may still work against earlier versions and we can make no guarantees of usability there.

### **Linux**

Please see the [Docker](https://docs.docker.com/installation/) documentation for how to install it for your Linux distribution.

**Essential**! After you install Docker, you must make sure that the user you are using to develop with monax has access to the Docker socket (which is accessible via the docker Linux usergroup). When you are logged in as the user you can do this:

sudo usermod -a -G docker $USER

That command will add the current user to the docker group which will mean that Docker will not need to be called from sudo. After you run that command, then please log out of the current shell and open a new shell. After that monax will then be able to connect to Docker.

Make sure that everything is set up with Docker by running (you shouldn’t see any errors in the command’s output):

docker version

**Note** you will need to make sure that you perform the above command for the user which will be running Monax.

If you’ve also chosen to install Docker Machine, please follow [these](https://docs.docker.com/machine/install-machine/#installing-machine-directly) instructions to install Docker Machine and [these](https://www.virtualbox.org/wiki/Linux_Downloads) to install VirtualBox; then create an Monax virtual machine and run the (eval) command:

docker-machine create -d virtualbox monax

eval $(docker-machine env monax)

**Note** Installation of VirtualBox is not a prerequisite, because you may choose to create your virtual machine on Amazon AWS cloud or DigitalOcean, but VirtualBox is what most people use alongside Docker Machine and what we recommend for the Docker Machine setup.

Proceed to one of the package or binary installations below to install the monax binary then finalize your setup by running.

monax init

monax init will be downloading a few Docker images which may take a few minutes.

#### **Debian Package Installation**

We have apt-get support for most current versions of Ubuntu and Debian Linux:

sudo add-apt-repository https://pkgs.monax.io/apt

curl -L https://pkgs.monax.io/apt/APT-GPG-KEY | sudo apt-key add -

sudo apt-get update && sudo apt-get install -y monax

#### **RPM Package Installation**

We have RPM support for most current versions of Fedora, CentOS, and RHEL:

sudo su -c "curl -L https://pkgs.monax.io/yum/monax.repo > \

/etc/yum.repos.d/monax.repo"

sudo yum update && sudo yum install -y monax

#### **Binary Installation**

Alternatively, you can download a release binary for the latest [Release](https://github.com/monax/monax/releases). Make sure you put the binary under one of the paths in the $PATH variable and that it has executable permissions:

curl -L https://github.com/monax/monax/releases/download/v0.16.0/monax\_0.16.0-linux-amd64 > monax

chmod +x monax

### **macOS**

We **highly recommend** that you utilize [Homebrew](https://brew.sh/) to install monax. Docker, Docker Machine, VirtualBox, and monax binary will be properly installed with:

brew cask install virtualbox

brew install monax docker docker-machine

If you are not a brew user then please install Docker, Docker machine, and VirtualBox by installing [Docker Toolbox](https://www.docker.com/products/docker-toolbox) and Monax binary from the [Release](https://github.com/monax/monax/releases) page. Make sure you put the binary under one of the paths in your $PATH variable and it has executable permissions:

curl -L https://github.com/monax/monax/releases/download/v0.16.0/monax\_0.16.0\_darwin\_amd64 > monax

chmod +x monax

If you don’t want to utilize Docker Toolbox, you can install those manually: follow [these](https://docs.docker.com/installation/) instructions to install Docker, [these](https://docs.docker.com/machine/install-machine/#installing-machine-directly) to install Docker Machine, and [these](https://www.virtualbox.org/wiki/Downloads) to install VirtualBox.

If you have chosen not to use Docker Toolbox at all, you need to create an Monax virtual machine and run the (eval) command:

docker-machine create -d virtualbox monax

eval $(docker-machine env monax)

**N.B.** At this time Docker for Mac (DFM) and Docker for Windows (DFW), which are still in beta, are not currently supported.

Finalize your setup by running:

monax init

monax init will be downloading a few Docker images which may take a few minutes.

### **Windows**

Install Docker, Docker Machine, and VirtualBox by downloading the [Docker Toolbox](https://www.docker.com/products/docker-toolbox) and Monax binary from the [Release](https://github.com/monax/monax/releases) page. Make sure you put the binary under one of the paths in your %PATH% variable.

If you don’t want to utilize Docker Toolbox, you can install those manually: follow [these](https://docs.docker.com/installation/) instructions to install Docker, [these](https://docs.docker.com/machine/install-machine/#installing-machine-directly) to install Docker Machine, and [these](https://www.virtualbox.org/wiki/Downloads) to install VirtualBox.

(You’ll want to run monax commands either from git bash or from the Docker Quickstart Terminal, a part of Docker Toolbox. If you prefer to use the cmd as your shell, you still can: every command should work as expected, though all the tutorials will assume that you are using the Docker Quickstart Terminal and are structured to support **only** that environment.)

If you have chosen not to use Docker Toolbox at all and use cmd as your shell, you need to create an Monax virtual machine:

docker-machine create -d virtualbox default

and create a script setenv.bat with these contents to be run before your every session with Monax:

@echo off

FOR /f "tokens=\*" %%i IN ('"docker-machine.exe" env default') DO %%i

**Note** – At this time Docker for Windows (DFW), which is still in beta, is not currently supported.

Finalize your setup by running:

monax init

monax init will be downloading a few Docker images which may take a few minutes.

### **ARM Installation (IoT devices)**

Although we once supported IoT installations, this has been temporarily disabled while the platform undergoes further consolidation. See [this issue](https://github.com/monax/monax/issues/1088) for more details on progress. See also the [deprecated ARM installation tutorial](https://monax.io/docs/deprecated/install-arm).

### **Building From Source**

If you would like to build from source [see our documentation](https://monax.io/docs/install-source).

### **Troubleshooting Your Install**

If you have any errors which arise during the installation process, please see our [trouble shooting page](https://monax.io/docs/install-troubleshooting) or join [The Marmot Den](https://slack.monax.io/)to ask for help.

## Step 2: Roll Your Own Blockchain in Seconds

If you want to create your blockchain it is two commands:

monax chains make test\_chain

monax chains start test\_chain --init-dir ~/.monax/chains/test\_chain/test\_chain\_full\_000

That test\_chain can be whatever name you would like it to be. These two commands will create a permissioned, smart contract enabled blockchain suitable for testing.

To check that your chain is running type (running chains have a \* symbol next to them rather than a -):

monax ls

You can peek at chain’s logs with these commands (-f for “follow”):

monax chains logs test\_chain

monax chains logs -f test\_chain

Note: although your chain may be “running” (i.e., has been started and has a docker container that is “ON”, it is possible that you chain is not making blocks and thus will be useless for deploying contracts. After running the above command, ensure you chain is indeed making blocks. Stay tuned for an monax chains info/status command.

Stop your chain:

monax chains stop test\_chain

Remove your chain (-f to force remove a running chain, -x to remove the chain’s separate data container which it writes to, and -d to remove the (local) chain directory entirely):

monax chains rm -xfd test\_chain

Obviously, you will want an ability to make chains which you properly parameterize. As such you can always type:

monax chains make --help

That’s it! Your chain is rolled!

Let’s remove all of the monax “stuff” before we move on to the next portion of the tutorials:

monax clean -yx

### **Step 2.a: Advanced Chain Making**

**Note:** If you’d like to get right into deploying contracts and building your ecosystem application, jump to Step 3 below.

Blockchains are meant to be trustless, and that means everyone generates their own keys. Validators and any other accounts to be included at the inception of a chain must be included in the genesis.json file. This is done using the --known flag for monax chains make. See our [known chain making tutorial](https://monax.io/docs/known-chain-making) for more information. For the purposes of this tutorial, however, we’ll be using a simplechain with one account.

To learn about the account types paradigm, try the chain making wizard:

monax chains make toRemoveLater --wizard

This will drop you into an interactive, command line wizard. Follow the text and the prompts to chain making bliss. Since we’re going to throw this chain away later you can just press “Enter” at each of the prompts or you can change the variables and get a feel for the wizard.

Once the wizard exits let’s take a look at what was created:

ls ~/.monax/chains/toRemoveLater

You should see three \*.csv files and a bunch of directories. Let’s look in one of those directories:

ls ~/.monax/chains/toRemoveLater/toremovelater\_full\_000

In that directory you should see a genesis.json, a priv\_validator.json and a config.toml. The marmots call these a “bundle” as generally they are what is needed to get a chain going.

What about those csv files? There should be three of them. Let’s take a look:

cat ~/.monax/chains/toRemoveLater/accounts.csv

cat ~/.monax/chains/toRemoveLater/validators.csv

cat ~/.monax/chains/toRemoveLater/addresses.csv

The first two files can be used later to create a new genesis.json if the actual json gets lost. One of the things about this tooling is that it **creates** the keys for you. That is helpful in some circumstances. For production/consortium chains this is not appropriate. See the [known chain making tutorial](https://monax.io/docs/known-chain-making) for more info.

The monax chains make tool comes with advanced account type and chain type definition capabilities. More information on complex chain making is included in our [advanced chain making tutorial](https://monax.io/docs/chain-making).

The last file is the addresses.csv file which is another artifact of the chain making process. It simply has the addresses and the “names” of the nodes. We find it useful when scripting out complex interactions and it is simply a reference file along the lines of addr=$(cat $chain\_dir/addresses.csv | grep $name | cut -d ',' -f 1).

OK, enough playing around let’s get serious! Cleaning after our previous experiment:

monax clean -yx

Per the above and after our review of the account types, we know we want to have two Root account types and one Full account type for our new chain. So let’s get to business.

chain\_dir=$HOME/.monax/chains/firstchain

chain\_dir\_this=$chain\_dir/firstchain\_full\_000

That will just create a few variables we’ll be using in the future. Now, we’re ready.

monax chains make firstchain --account-types=Root:2,Full:1 --unsafe

That’s it! Let’s double check the files to make sure we are squared away.

ls $chain\_dir

ls $chain\_dir\_this

You’ll see a genesis.json, a priv\_validator.json and a config.toml in $chain\_dir\_this.

#### **Step 2.a.3: Instantiate the Blockchain**

With all the files prepared we’re ready to rock and roll.

monax chains start firstchain --init-dir $chain\_dir\_this

Check that the chain is running with:

monax ls

You’ll see something like:

CHAIN ON VERSION

firstchain \* 0.16.0

Note: You can see more information with monax ls --all.

To see the logs of the chain:

monax chains logs firstchain

To turn off the chain:

monax chains stop firstchain

Boom. You’re all set with your custom built, permissioned, smart contract-ified, chain.

You start your chain up again for the next step:

monax chains start firstchain

or remove everything with:

monax clean -yx

If anything went wrong, please see our trouble shooting guide -> [1](https://monax.io/docs/getting-started/#fn:1), [2](https://monax.io/docs/getting-started/#fn:2), [3](https://monax.io/docs/getting-started/#fn:3), [4](https://monax.io/docs/getting-started/#fn:4)

## Step 3: Deploy your ecosystem application using smart contract templates

In general we are going to take three steps in order to get our contracts deployed to the blockchain:

1. Write a simple contract
2. Make sure your application package has the proper information
3. Deploy the contracts

#### **Contracts Strategy**

We are going to use a very simple get / set contract which sets a variable and gets that same variable. It is about the easiest interactive contract one can imagine and as such we will use that for showing how to work with the Monax platform.

### **Step 3.1: Make A Contract for Idi**

The first thing we’re going to do is to add a very simple contract.

cd ~/.monax/apps

mkdir idi

cd idi

Now you’ll make a file in this directory. Let’s assume that is called idi.sol and has the following contents

pragma solidity ^0.4.0;

contract IdisContractsFTW {

uint storedData;

function set(uint x) {

storedData = x;

}

function get() constant returns (uint retVal) {

return storedData;

}

}

[COPY SOURCE](https://monax.io/docs/getting-started/#copy)[/DOCS/CONTRACTS\_SIMPLE\_IDI/IDI.SOL](https://monax.io/docs/contracts_simple_idi/idi.sol)

What does this contract do? Well, it isn’t very interesting, we know. It merely gets and sets a value which is an unsigned integer type.

### **Step 3.2: Fixup your epm.yaml**

Next we need to make an epm.yaml and make it look something like this:

jobs:

- name: setStorageBase

job:

set:

val: 5

- name: deployStorageK

job:

deploy:

contract: idi.sol

- name: setStorage

job:

call:

destination: $deployStorageK

data: set $setStorageBase

- name: queryStorage

job:

query-contract:

destination: $deployStorageK

data: get

- name: assertStorage

job:

assert:

key: $queryStorage

relation: eq

val: $setStorageBase

[COPY SOURCE](https://monax.io/docs/getting-started/#copy)[/DOCS/CONTRACTS\_SIMPLE\_IDI/EPM.YAML](https://monax.io/docs/contracts_simple_idi/epm.yaml)

Now, what does this file mean? Well, this file is the manager file for how to deploy and test your smart contracts. The package manager invoked by monax pkgs do will read this file and perform a sequence of jobs with the various parameters supplied for the job type. It will perform these in the order they are built into the yaml file. So let’s go through them one by one and explain what each of these jobs are doing. For more on using various jobs [please see the jobs specification](https://monax.io/docs/specs/jobs_specification).

#### **Job 1: Set Job**

The set job simply sets a variable. The package manager includes a naive key value store which can be used for pretty much anything.

#### **Job 2: Deploy Job**

This job will compile and deploy the idi.sol contract using the local compiler service.

#### **Job 3: Call Job**

This job will send a call to the contract. The package manager will automagically utilize the abi’s produced during the compilation process and allow users to formulate contract calls using the very simple notation of functionName params. The package manager also allows for variable expansion.

So what this job is doing is this. The job is pulling the value of the $setStorageBase job (the package manager knows this because it resolved $ + jobName to the result of the setStorageBase job) and replacing that with the value, which is 5. Then it will send that 5 value to the set function of the contract which is at the destination that is the result of the deployStorageK job; in other words the result of Job 3. For more on variables in the package manager, please see the [variables specification](https://monax.io/docs/specs/variable_specification).

#### **Job 4: Query Contract Job**

This job is going to send what are alternatively called simulated calls or just queries to an accessor function of a contract. In other words, these are read transactions. Generally the query-contract is married to an accessor function (such as get in the idi.sol contract). Usually accessor, or read only functions, in a solidity contracts are denoted as a constant function which means that any call sent to the contract will not update the state of the contract.

The value returned from a query-contract job then is usually paired with an assert.

#### **Job 5: Assert Job**

In order to know that things have deployed or gone through correctly, you need to be able to assert relations. The package manager provides you with:

* equality
* non-equality
* greater than or equals (for integers & unsigned integers values only)
* greater than (for integers & unsigned integers values only)
* less than or equals (for integers & unsigned integers values only)
* less than (for integers & unsigned integers values only)

Relations can use either eq ne ge gt le lt syntax, or, in the alternative they can use == != >= > <= < syntax in the relation field. This is similar to Bash. To make this more explicit we have chosen in the above epm.yaml to use the eqsyntax, but feel free to replace with == syntax if you want.

Both the key and the val (which in other testing frameworks are the given and expected in an assert function) use variable expansion to compare the result of what was supposed to be sent to the setStorageBase job (which should have been sent to and stored in the contracts’ storage) with what was received from the queryStorage job (which in turn called the get function of the contract).

### **Step 3.3: Deploy (and Test) The Contract**

See the Step 2 above if you need to review the chain making process. This series of commands assumed you followed that tutorial and continued here after monax chains stop firstchain.

First, let’s get our chain turned back on.

monax ls

If it’s on, you’ll see:

CHAIN ON VERSION

firstchain \* 0.16.0

Whereas if it has been stopped, the ON field will have - rather than \*. The same logic applies to services.

If firstchain is not running, then turn it on with:

monax chains start firstchain

or make a new chain if firstchain no longer exists.

Now, we are ready to deploy this world changing contract. Make sure you are in the ~/.monax/apps/idi folder, or wherever you saved your epm.yaml. Note that this is a very common pattern in simple contract testing and development; namely to (1) deploy a contract; (2) send it some transactions (or calls); (3) query some results from the contract (or query-contracts); and (4) assert a result. As you get moving with contract development you will likely find yourself doing this a lot.

addr=$(cat $chain\_dir/addresses.csv | grep firstchain\_full\_000 | cut -d ',' -f 1)

That will make sure we have available the address we would like to use to deploy the contracts. Now we’re ready. If the above does not output an address then check your $chain\_dir variable and also check that the firstchain\_full\_000 variable exists in the addresses.csv.

monax pkgs do --chain firstchain --address $addr

You should be able to use any of the addresses you generated during the chainmaking tutorial since they all have the same permission levels on the chain (which, if you followed the simple tutorial are basically all public). If you are using this tutorial outside of the tutorial sequence then you can just give it the address that you’d like to use to sign transactions instead of the grep firstchain\_full\_000 bash expansion.

(For those that do not know what is happening in that bash line: cat is used to “print the file” and “pipe” it into the second command; grep is a finder tool which will find the line which has the right name we want to use; the cut says split the line at the , and give me the first field).

Note that the package manager can override the account which is used in any single job and/or can set a default account job which will establish a default account within the yaml. We find setting the default account within the yaml to usually be counter-productive because others will not be able to easily use your yaml unless they have the same keys in their monax-keys (which we **never** recommend). For more on using accounts [please see the jobs specification](https://monax.io/docs/specs/jobs_specification).

Since we have a deployed contract on a running chain, please do take a look at the available options for deployment using the package manager with:

monax pkgs do --help

That’s it! Your contract is all ready to go. You should see the output in jobs\_output.json which will have the transaction hash of the transactions as well as the address of the deployed idi.sol contract. The job runner (monax pkgs do) can be leveraged for building and interacting with your custom application.

For making consortium chains with several stakeholders

There are three steps to making a permissioned chain with known keys:

1. Make for (or get from) the public keys for all parties
2. Make the genesis.json file and share it
3. Sort out a config.toml for each party
4. Instantiate the chain

We shall go through these in their logical order.

## Users Design

To do this we need to, first, consider, who will get what permissions and why. It is outside the scope of this tutorial to outline all of the considerations which would come into play when thinking about creating a permissioning system, but for the purposes of this tutorial, we will craft the genesis block to use the following paradigm:

* 1 Administrator (the developer who has **full** control over the chain) => this is a “Full Account” type
* 2 Validators (who participate in the consensus of the chain but do nothing else) => this is a “Validator Account” type

If you would like to understand all of the permissions which an monax chains smart contract network is capable of providing, [please see here for more information](https://monax.io/platform/db).

We use an abstraction to simplify the chain making process called Account Types. This abstraction is just that, an abstraction to help users quickly get up to speed. In order to reduce the complexity of dealing with different types of accounts typically built on a chain, we use the idea of “account types”. Account types are not restrictive in the sense that they are not the “only” types of accounts you can make with monax chains.

Account types are simply bundles of permissions no more no less. Using the monax tooling you can also create your own account types with your own bundles of permissions which will be helpful.

To learn about advanced chain making and account types, [see here](https://monax.io/docs/chain-making).

## Step 1: Make (or get) the public keys

Everyone who interacts with an monax chain will need to have a properly formated keypair. To make a keypair we will use monax keys.

monax keys usually operates as a signing daemon, but when we use monax keys to create key pairs what we are doing effectively is writing files. As is usual with the Monax tooling, monax keys is opinionated and will work by default against the following directory: ~/.monax/keys/data. When a key pair is created, that key pair will get written into that directory.

These files will be written to a file system located inside the monax keys data container. As we go through this tutorial we will explain a bit about what that means. When we are using containers, these containers are not built to hold data, but rather are built to hold what is needed to run processes. But, if we’re making keypairs, then we definitely want to keep these.

To accomplish this, we will use the monax tooling. First we need to start the monax-keys daemon:

monax services start keys

Check that is it indeed running with:

monax ls

You’ll see something like:

SERVICE ON VERSION

keys \* 0.16.0

which indicates (\* rather than -) that the keys services is on (running). To see a more comprehensive output for your services, try monax ls --all

To see what we can do with monax keys we will run:

monax services exec keys "monax-keys -h"

This runs the monax-keys -h command “inside” the keys container.

Instead of dealing with the monax-keys service directly, however, we will use monax keys from the monax tool. The monax keys commands are basically wrappers around the monax-keys commands which are ran inside containers. To see the wrappers which the monax tooling provides around the monax-keys daemon, please type:

monax keys -h

Now it is time to generate some keys!

For the purposes of this tutorial **only** we will also create all of the necessary keys for all of the “users” of the chain and we will do so without passwords. Again, this is for demonstration purposes only, for a production system you will not do what we’re about to do.

monax keys gen --save

This will create one key for you. The output here should look something like this:

Saving key to host 49CA2456F65B524BDEF50217AE539B8E10B37421

The --save flag exported your key, which will be in the directory at ~/.monax/keys/data/49CA2456F65B524BDEF50217AE539B8E10B37421/. If omitted, you key will remain in the container and you can use monax keys export 49CA2456F65B524BDEF50217AE539B8E10B37421 to save it to host afterwards.

To see the keys which monax-keys generated both inside the container type and available on your host machine type:

monax keys ls

Each of the three participants in this chain would run these series of commands independently, on their own trusted machine then submit their public key and address to whoever is making the genesis.json. For maximum trust in the chain, each party ought to generate their own genesis.json and ensure they match.

**Note** In version 0.16, we do not have a simple method of easily creating a priv\_validator.json from an existing key. Thus, rather than creating keys via monax keys gen --save, we’re going to take advantage of monax chains make as it creates keys and some other files we’ll need. These three commands are meant to be run seperately by each participant on their own machine:

monax chains make throwawayFull --account-type=Full:1

monax chains make throwawayVal --account-type=Validator:1

monax chains make throwawayVal --account-type=Validator:1

then getting the address and exporting the key from container to host (monax keys export ADDR). This will create a priv\_validator.json in the chains directory on the host. It is needed for Step 4. Future versions will simplify this process and abstract/harden the way keys are handled.

**End Note**

Next, we’ll make the all important genesis.json file.

## Step 2: Make the genesis.json

Before we begin, let’s walk through the various files which are needed to run a chain:

1. the genesis.json which tells the chain how it should configure itself at the beginning (or, its genesis state).
2. the chain configuration file for Monax chains is called config.toml; see Step 3 for more information.
3. the keypair used by tendermint for signing blocks is the priv\_validator.json.

All three files are usually located in the ~/.monax/chains/<your\_chain>/<an\_account> directory after running monax chains make

The genesis.json is the primary file which tells monax chains how to instantiate a particular chain. It provides the “genesis” state of the chain including the accounts, permissions, and validators which will be used at the beginning of the chain. These can always be updated over the life of the chain of course, but the genesis.json provides the starting point.

With all that said, we’re ready to make a known chain. Doing so requires preparing two additional files and using three additional flags to the monax chains make command.

First, you’ll need to the public keys for each party. This can be found in the priv\_validator.json as described in the Note in Step 1. Then, make a file named accounts.csv and replace the public keys seen below with the ones submitted by each participant:

0962E87A7A75B27174FB0F2C76FE9A54B78BBD5AD0E9605BF946E08F080BC657,99999999999999,admin\_alice,16383,16383

C8E4C807152F70B5CE44E072D2CFB34F2521382CCA892AE90F1E058EE619E418,9999999999,validator\_bob,32,16383

22774A27B9471BD7B0D9015B35067020FA99E7DD017721A577864127789DA0F2,9999999999,validator\_charlie,32,16383

as well as a file named validators.csv, which is nearly identical:

0962E87A7A75B27174FB0F2C76FE9A54B78BBD5AD0E9605BF946E08F080BC657,99999999999999,admin\_alice,16383,16383

C8E4C807152F70B5CE44E072D2CFB34F2521382CCA892AE90F1E058EE619E418,9999999998,validator\_bob,32,16383

22774A27B9471BD7B0D9015B35067020FA99E7DD017721A577864127789DA0F2,9999999998,validator\_charlie,32,16383

Some notes on these two files: they are automatically generated when making unknown chains and are useful templates that can be used to regenerate a genesis.json. In this case, however, because all accounts happen to also be validators, the files are nearly identical. With a chain where some accounts are not validators, the accounts.csv will have more accounts than the validators.csv. The latter two rows in the validators.csv have one less token allocated compared to the same rows in the accounts.csv. This is because the 99999... column in the accounts.csv is the initial token allocation whereas in the validators.csv this number represent the number of tokens to bond. The third column in both the .csv’s is the account name and should, but need not, match the moniker field in the config.toml. The latter two columns handle permissions, a topic dealt with elsewhere.

We’re ready to go!

monax chains make myCustomChain --known --accounts /path/to/accounts.csv --validators /path/to/validators.csv

This will output a genesis.json to stdout, so we recommend adding >> genesis.json to the end of the above command.

Finally, distribute this genesis.json to two validators who will be participating in the chain. They should also make one and confirm that it is identical.

## Step 3: Sort the config.toml

With the exception of known chains (this tutorial), the monax chains make command automatically makes a config.toml for each account. The moniker field for each account willtake on the name of the account. Optionally, the --seeds-ip flag will take a csv string of IP:PORT combinations and is used to point your consensus engine to the peers it should connect into. Review the [advanced chain deploying tutorial](https://monax.io/docs/chain-deploying) for more information.

Since we used monax chains make rather than (ideally, and in later versions) monax keys gen to make a key for each participant, they will each have an existing config.toml. At this point, admin\_alice (the Full Account administrator) will provide his or her public IP address to the validators. Each user should now edit their config.toml to have the monikerfield with their respective account names, while validator\_bob and validator\_charlie should edit seeds = "IP-OF-ALICE:46656".

At this point, each user should have, on their own machine:

* keys service running, with one key in it
* the key saved (exported) to host
* a directory with each: priv\_validator.json, config.toml, and genesis.json

For the next step, we are assuming that each user has the files in a directory like so:

~/.monax/chains/myCustomChain/admin\_alice

~/.monax/chains/myCustomChain/validator\_bob

~/.monax/chains/myCustomChain/validator\_charlie

on their respective machines.

## Step 4: Instantiate the chain

Given the directory structure above and a chain name of myCustomChain, each user runs:

monax chains start myCustomChain --init-dir ~/.monax/chain/myCustomChain/admin\_alice

monax chains start myCustomChain --init-dir ~/.monax/chain/myCustomChain/validator\_bob

monax chains start myCustomChain --init-dir ~/.monax/chain/myCustomChain/validator\_charlie

in that order (well, as long as admin\_alice is run first). What will happen is that once admin\_alice is up and running, each validator\_, will, when started, “dial-in” to connect to admin\_alice.

And there you have it; a custom chain built using pre-generated keys!