go -> gno

building a microblog

https://www.youtube.com/watch?v=F-_dadxcRJM

zack scholl (@schollz)

go -> gno

Gno is an interpreted version of the programming language Go.

Gno allows Go developers (like myself) a low-friction way to write **smart contracts** without having to learn an exclusive language.

Gno.land is the platform to write the smart contracts.

More information about Gno.land ecosystem: https://gno.land/about

Today I will focus getting up and running with Gno, assuming you know some Go. I will go over the workflow and syntax using a practical example of a "microblog" smart contract.

go -> gno

prerequisites

Working with Gno is as easy as working in Go - the syntax is near identical but workflow is different.

First make sure you have Go installed. Then I recommend installing Visual Studio Code w/ the Gno extension by Hariom Verma. Then install `gofumpt`:

```
go install mvdan.cc/gofumpt@latest
```

and clone the Gno project:

```
git clone https://github.com/gnolang/gno
cd gno
```

Build the Gno project utilities and `gno.land`:

```
make install
cd gno.land && make build
```

* Anytime you make changes to the Gno project you may have to rebuild `gno.land` or Gno tools.

go -> gno

prerequisites

keys

- generating

First create a key. This will be used to make transactions against the blockchain. For now it will be used for local development.

```
> gnokey generate
brush laugh sure area film ...
```

Copy the bip39 mnemonic. Now we will actually add the key:

```
gnokey add --recover yourkey
```

Enter the passphrase twice and then enter the bip39 mnemonic generated earlier.

Now you should see your key when listing them:

```
> gnokey list
0. yourkey (local) - addr: youraddress ...
```

go -> gno

prerequisites

keys

- generating

- genesis

For local development, you should add the key address ("`addr`") to the `genesis_balances.txt` so that you have tokens to make transactions. Get the address:

```
> gnokey list
0. yourkey (local) - addr: youraddress pub: ...
```

Copy the address, `youraddress` and now edit `gno.land/genesis/genesis_balances.txt` and add the line at the end with your address:

```
youraddress=10000000000ugnot # @yourkey
```

This makes development easy without having to utilize a faucet.

go -> gno

prerequisites

keys

gno.land

Gno.land is the platform to write smart contracts in Gno, providing a transition between web2 and web3.

As a Go developer, the Gno.land platform allows you to create smart contracts that can act as web servers that return Markdown.

A "realm" on Gno.land basically is a package with a `Render(path) string` function, which takes a path, processes it, and returns markdown.

The Markdown can be used to generate the webpage and can display useful information about the smart contract state (number of tokens, ownership, etc.).

go -> gno

prerequisites

keys

gno.land

- spinning up

Lets spin up a local instance of Gno.land and create a user with our address.

The realm gno.land/r/demo/users makes it easy to add and view users.

I like to make a script, `start.sh` to easily spin-up an environment:

```
#!/bin/bash

pkill -f 'build/gnoland'
pkill -f 'build/gnoweb'
rm -rf gno.land/testdir
cd gno.land && ./build/gnoland &
sleep 5
cd gno.land && ./build/gnoweb -bind 0.0.0.0:8888 &
sleep 2
```

You can run that script and wait a few seconds for the `gno.land` server and `gnoweb` interface to spin-up:

```
./start_gno.land.sh
```

Before continuing, its also easiest if you save your password to a file, e.g. "`password`" and use that when making transaction calls.

Then you can create a user with this transaction command:

```
cat password | gnokey maketx call \
    --pkgpath "gno.land/r/demo/users" --func "Register" \
    --args "" --args "yourname" --args "yourprofile" \
    --gas-fee "1000000ugnot" --gas-wanted "2000000" \
    --broadcast --chainid dev --remote localhost:26657 \
    --send "200000000ugnot" -insecure-password-stdin=true \
    yourkey
```

Now you will be able to see your user in the realm:

- gno.land/r/demo/users:yourname

The `gnokey maketx` allows you to call a function within a realm. For `/r/demo/users` the function is `Register`.

One of the cool things about `gno.land` is that the source is available for every smart contract, for example the users realm: /r/demo/users/users.gno.

The function `Register` has three arguments - `(inviter std.Address, name string, profile string)`. The arguments are inputs to to the `gnokey maktex` as `--args` arguments:

```
--args "" --args "yourname" --args "yourprofile"
```

The first argument is the address of the inviter (blank since we don't have one). The second argument is your name, as will be shown in the profile, and the final argument is any info you want to be shown on your page.

The `gno.land` exists as a repository of realms that can be utilized within your own smart contracts.

The route of the realm is given by its package path. In this example it is `/r/demo/users`.

The rendering of the realm can take other arguments, which are designated after the colon, `:`. For example, `yourname` is an argument to the render function in this path: gno.land/r/demo/users:yourname.

If we look at the `Render()` function of this realm (this is the function that is run when you go to the site), it will pull out the username using the semicolon: gno.land/r/demo/users/users.gno.

```
func Render(path string) (markdown string) {
    // path is everything after ":"
    ...
}
```

go -> gno

prerequisites

keys

gno.land

microblog

- add package

Microblog is a realm that lets users have feeds of time-dated posts. It lives at `/r/demo/microblog`.

To use a realm, we need to add to Gno.land using a transaction to add the microblog packages, and then add the microblog realms

(I will get into the details of the realm and package, but first lets try it.)

We start by adding the only package needed for microblog:

```
cat password | gnokey maketx addpkg \
    --pkgpath "gno.land/p/demo/microblog" \
    --pkgdir "examples/gno.land/p/demo/microblog" \
    --deposit 100000000ugnot --gas-fee 1000000ugnot \
    --gas-wanted 2000000 --broadcast --chainid dev \
    --remote localhost:26657 --insecure-password-stdin=true \
    yourkey
```

Then add the realm, which happens to be the same path except its `/r/` instead of `/p/`:

```
cat password | gnokey maketx addpkg \
    --pkgpath "gno.land/r/demo/microblog" \
    --pkgdir "examples/gno.land/r/demo/microblog" \
    --deposit 100000000ugnot --gas-fee 1000000ugnot \
    --gas-wanted 2000000 --broadcast --chainid dev \
    --remote localhost:26657 --insecure-password-stdin=true \
    yourkey
```

We can check to see that its up by going to its site:

- gno.land/r/demo/microblog

It will be blank because we have not added any information to it yet.

There is basically just one function: `NewPost(text string)` which you can call to add some post to your feed:

```
cat password | gnokey maketx call \
    --pkgpath "gno.land/r/demo/microblog" \
    --func "NewPost" --args "*hello*, **world**." \
    --gas-fee "1000000ugnot" --gas-wanted "2000000" \
    --broadcast --chainid dev --remote localhost:26657 \
    --send "200000000ugnot" -insecure-password-stdin=true \
    yourkey
```

Now your post will show up on the microblog realm.

The realm itself is very simple. It calls `Render` to render markdown that is used to generate the html of `gno.land` and it has a function for adding posts.

The main guts of the realm is in the package, `/p/demo/microblog`:

- gno.land/p/demo/microblog/microblog.gno.

A *realm* is Gno code with state, that represents a smart contract with storage and coins. Realms have a `Render(path string) string` function that will be called when making a transaction.

A *package* is Gno code that does not have state. Usually it is code that may be used by many realms. However you can also import realms. This can have any functions or structures exported to be used within realms.

Lets look at the microblog package to understand the differences between Gno and Go:

- gno.land/p/demo/microblog/microblog.gno

This looks just like Go code, with a few subtle differences, most notable in the imports:

```
package microblog

import (
	"errors"
	"sort"
	"std"  <- !!!
	"strings"
	"time"

	"gno.land/p/demo/avl"  <- !!!
	"gno.land/p/demo/ufmt"  <- !!!
	"gno.land/r/demo/users"  <- !!!
)
...
```

These imports are transpiled from `.gno` to `.go` code and have special properties.

There is a special import, `std`.

The `std` package is a Gno-specific package that lets you access the caller's address, using `std.GetOrigCaller()` and store addresses using the type `std.Address`.

For example, when a `NewPost` is called to microblog it gets the user from the key:

```go
func (m *Microblog) NewPost(text string) error {
	author := std.GetOrigCaller() // <- returns address
	                              //    as std.Address
	_, found := m.Pages.Get(author.String())
	if !found {
		m.Pages.Set(author.String(), &Page{
			Author:    author,
			CreatedAt: time.Now(),
		})
	}

	page, err := m.GetPage(author.String())
	if err != nil {
		return err
	}
	return page.NewPost(text)
}
```

The `avl.Tree` is imported with `gno.land/p/demo/avl`. This data structure is a self-balancing binary search tree.

Gno is completely determistic for accountability (#452) so only one path exists between states for validators to reach consesus. The `avl.Tree` can be used as a determistic map since Go's map ordering is indeterminate (#311).

Here's a tiny demo:

```
t := avl.Tree{}
t.Set("mystring",&MyStructure)
v, found := t.Get("mystring")
if (found) {
    v2 := v.(*MyStructure) // cast it back
}
```

Make sure to cast here because it stores as an `interface{}`.

For example, in the microblog code, `avl.Tree` stores pages:

```go
func NewMicroblog(title string, prefix string) (m *Microblog)
	return &Microblog{
		Title:  title,
		Prefix: prefix,
		Pages:  avl.Tree{},
	}
}
```

which can be retrieved through `Get` or `Iterate`:

```go
func (m *Microblog) GetPages() []*Page {
	var (
		pages = make([]*Page, m.Pages.Size())
		index = 0
	)
	m.Pages.Iterate("", "", func(key string, value interface{
		pages[index] = value.(*Page)
		index++
		return false
	})
	sort.Sort(byLastPosted(pages))
	return pages
}
```

As of June 2023 Gno does not support reflection (#750) which means some of the Go standard library does not work in Gno.

For example, `fmt` uses reflection. In Gno, you can instead use `gno.land/p/demo/ufmt` which is a micro-implementation of the `fmt` library. This is the library that you can use to do formatting with basic types, like using `ufmt.Sprintf`.

For example, in microblog, the `ufmt` package is used to format the title:

```
ufmt.Sprintf("# %s\n\n", m.Title)
```

The lack of reflection affects some other packages, like `sort`. Currently you cannot use `sort.Slice` because the code uses reflection and is not ported to Gno.

However, you can use the classic method of implementing `Len()`, `Swap(i, j int)` and `Less(i, j int) bool` to do sorting.

For example in the microblog code:

```
// byLastPosted implements sort.Interface for []Page based on
// the LastPosted field.
type byLastPosted []*Page

func (a byLastPosted) Len() int           { return len(a) }
func (a byLastPosted) Swap(i, j int)      { a[i], a[j] = a[j]
func (a byLastPosted) Less(i, j int) bool { return a[i].LastPo
...
...
sort.Sort(byLastPosted(pages))
```

The other main difference between Go and Gno is the imports.

While the standard library is the same, its currently not possible to import 3rd party code, other than realms + packages. This may change in the future.

Currently, there are already many available packages and many available realms which can be imported.

In microblog we use the realm for users, the package for `avl.Tree` and the package for `ufmt`, which are accessed using the prefix `gno.land/`:

```
package microblog

import (
    ...
    "gno.land/p/demo/avl"  <-
    "gno.land/p/demo/ufmt" <-
    "gno.land/r/demo/users"  <-
)
...
```

This is just the tip of the iceberg when it comes to Gno and Go.

Most anything you can do in Go, you can do in Gno (with the caveats mentioned).

For more information checkout the resouces here:

- https://github.com/gnolang/awesome-gno