gno.land

Building Dynamic Applications With Go

(and Gno!)

Presented by Dylan Boltz, Sr. Golang Developer @ Gno.land

Who am I?

Dylan Boltz

Sr. Golang Engineer @ Gno.land

Go maximalist

Working in web3 for the past three years

Enjoying my time in Serbia

https://github.com/deelawn

What is Gno and Gno.land?

- Gno is an interpreted, deterministic version of Go with a persistent data store
- Gno incorporates a select list of Go features and standard libraries from Go version 1.17
- Gno.land is soon to be released blockchain which uses Gno as its smart contract programming language
- This presentation will focus on the Gno virtual machine, not Gno.land

What will this presentation cover?

- High level Gno features and architectural overview
- Virtual machine
- Examples of Gno use cases as a virtual machine outside of the blockchain context

Repository with examples

https://github.com/deelawn/gno-workshop

Note that none of the examples depend on another, so if one doesn’t work or you skip one, don’t worry about not being able to try the next

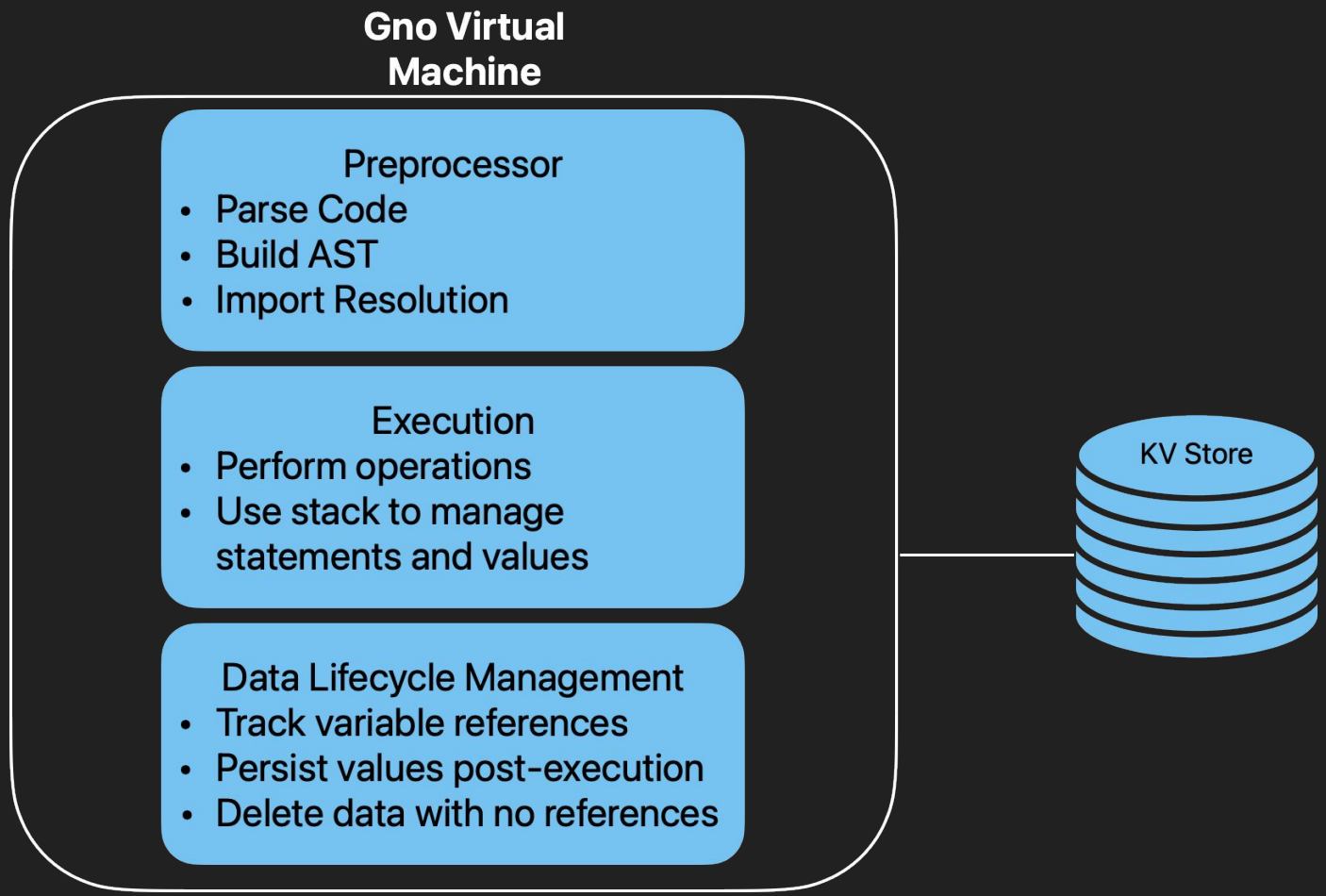
Gno features and limitations

- Determinism guaranteed
- Generics not supported (yet)
- No concurrency
- Built in application state persistence
- Import select standard libraries and other packages and applications deployed to a VM instance

https://docs.gno.land/reference/go-gno-compatibility

Determinism

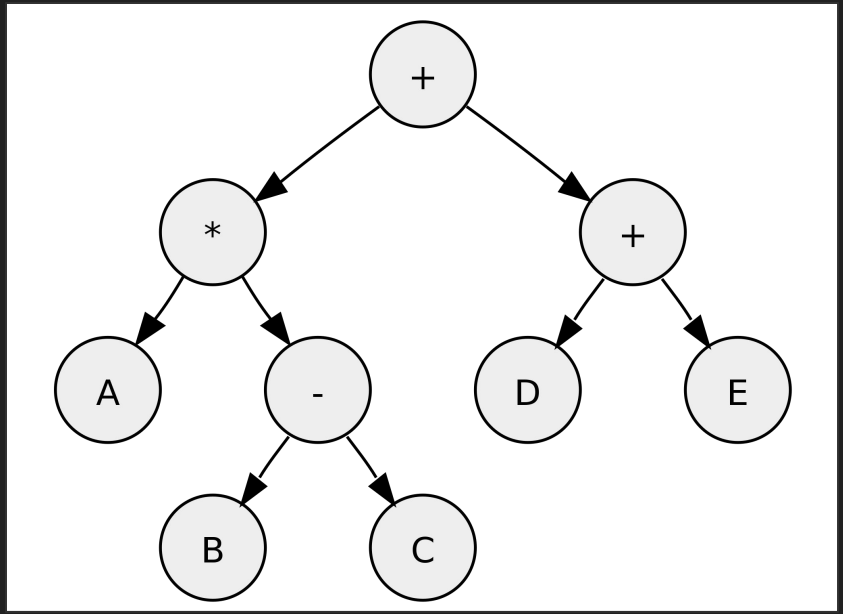
- Map types are less efficient because key ordering must be preserved for deterministic iteration
- No network packages are supported
- time.Now()
  - Can be supported at the blockchain level by using the block time
  - Non-blockchain gno requires a different solution


Gno Virtual Machine
Preprocessor
• Parse Code
• Build AST
• Import Resolution
Execution
• Perform operations
• Use stack to manage statements and values
Data Lifecycle Management
• Track variable references
• Persist values post-execution
• Delete data with no references
KV Store

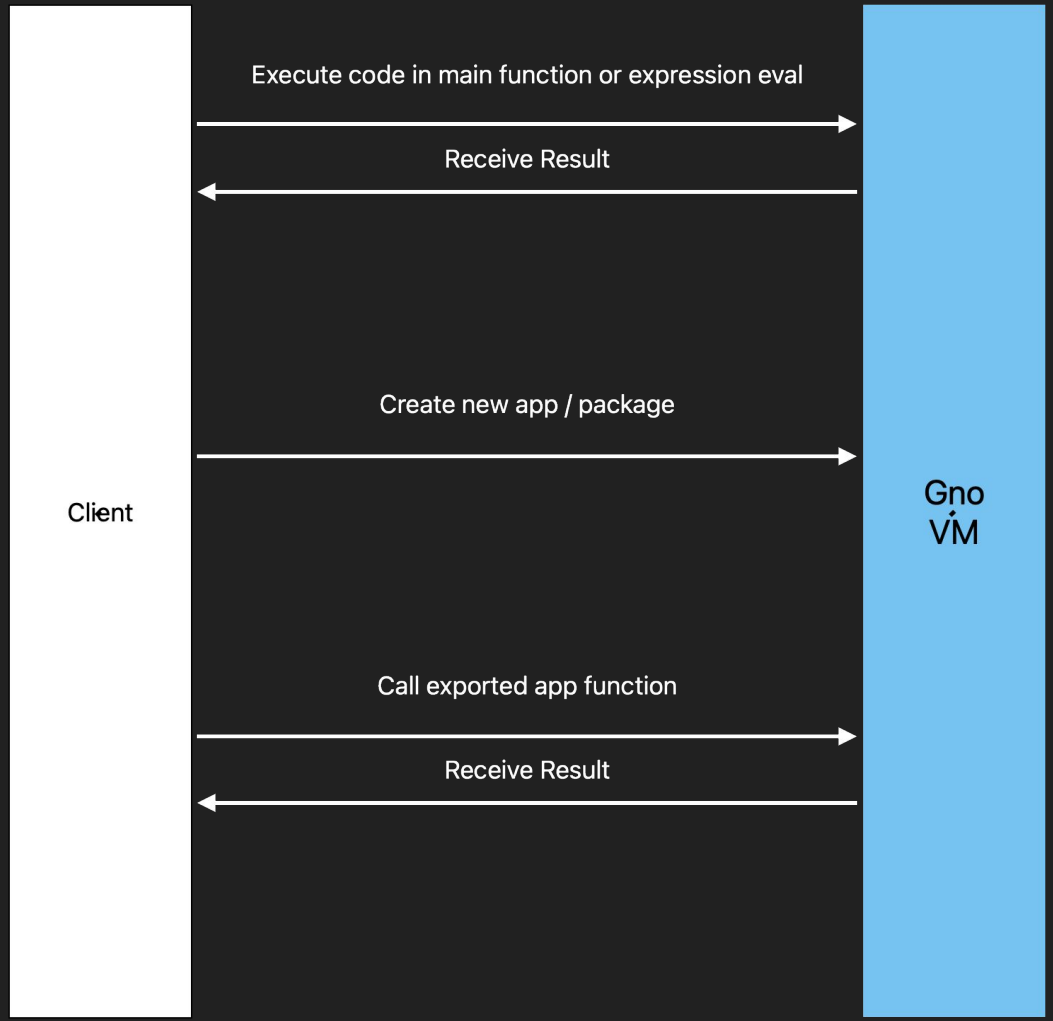
How does the VM work?

- AST postfix traversal
- Nodes are statements that contain expressions to be evaluated
- Intermediate values and expressions pushed to and popped from the stack
- Values persisted to storage
- VM Stack Trace Example

Example of evaluating
A(B-C) + (D+E)*



Source:
https://en.wikipedia.org/wiki/Stack_machine


Client
Execute code in main function or expression eval
Receive Result
Create new app / package
Call exported app function
Receive Result
Gno
VM

Hands-on: Phase 1

- Use existing or create a new .gno file with main function
- cat <filename> | go run main.go

```
package main

func main() {
	println("hello")
}

```

```
package main

import (
	"context"
	"fmt"
	"io"
	"os"

	"github.com/gnolang/gno/gno.me/gno"
)

func main() {
	vm, _ := gno.NewVM()
	input, err := io.ReadAll(os.Stdin)
	if err != nil {
		panic("failed to read input")
	}

	res, err := vm.Run(context.Background(), string(input))
	if err != nil {
		fmt.Println("run error:", err.Error())
		return
	}

	fmt.Println("result:", res)

}
```

How is this helpful?

- Let’s be honest – it’s not THAT helpful
- You can run arbitrary gno code
- Basically go code with reduced functionality
- But it’s DETERMINISTIC!!!!!

<pause for applause>

Hands-on: Phase 2

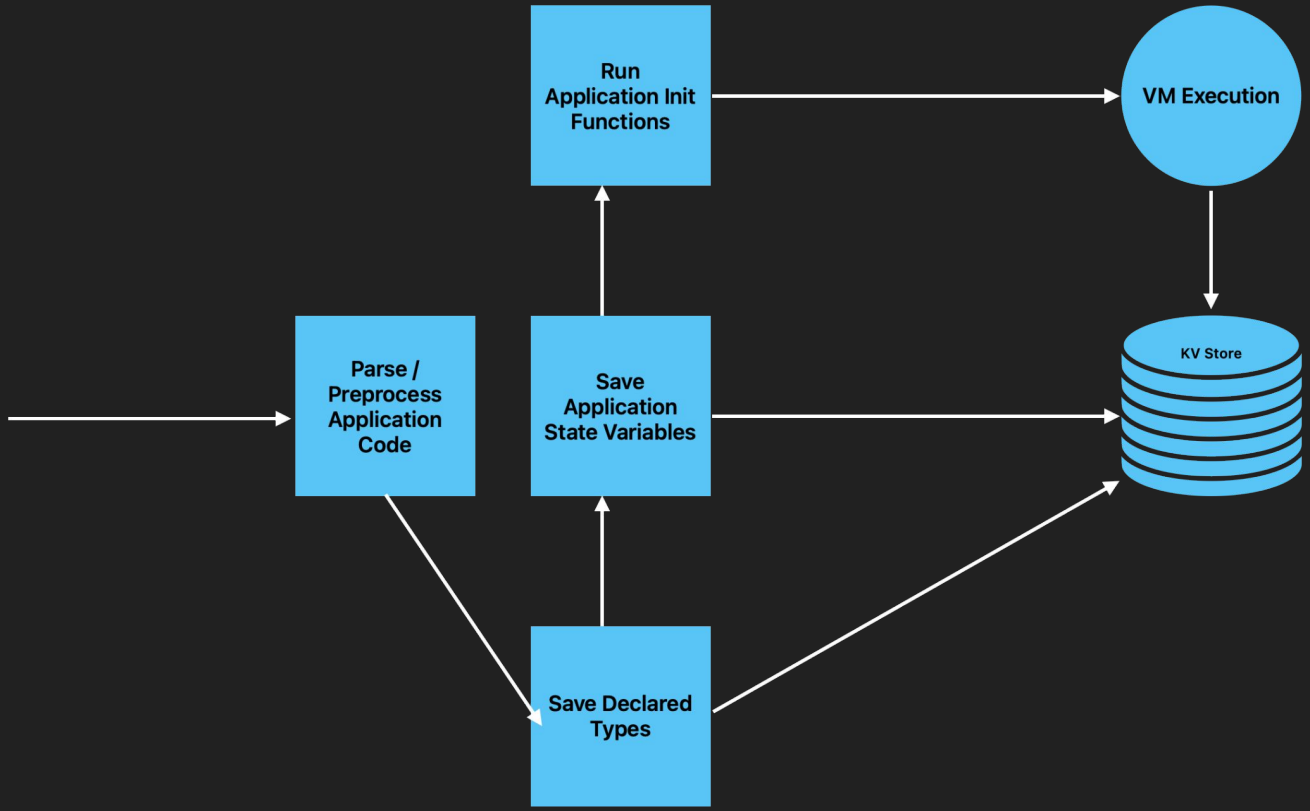
```
1   package adder
2
3   var value int
4
5   func Add(n int) {
6       value += n
7   }
8
9   func Value() int {
10      return value
11  }
12
```

```
52
53  switch os.Args[1] {
54  case "call":
55      callSet.Parse(os.Args[2:])
56      result, _, err := vm.Call(
57          context.Background(),
58          *callApp,
59          *callIsPkg,
60          *callFunc,
61          callArgs...,
62      )
63  case "create":
64      createSet.Parse(os.Args[2:])
65      var code []byte
66      code, err = os.ReadFile(*createFile)
67      if err != nil {
68          panic("failed to read file")
69      }
70
71      _, err = vm.Create(          You, 4 days
72          context.Background(),
73          string(code),
74          *createIsPkg,
75          false,
76      )
77  }
```

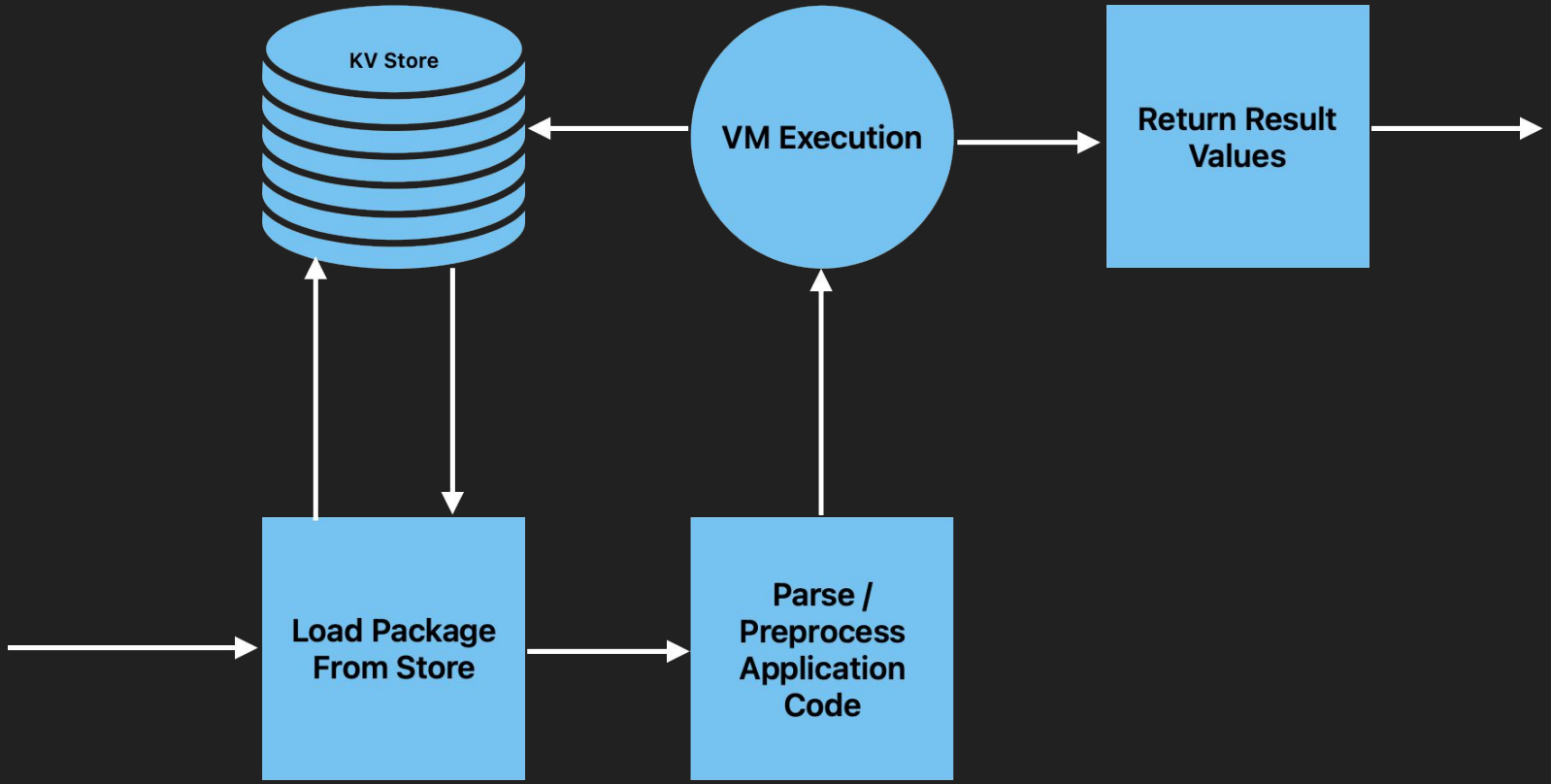
Hands-on: Phase 2

- Use the VM in a simple CLI application
- Create a new gno app
  - Use the provider “adder.gno” file or create your own
  - go run main.go create -file adder.gno
- Call the `Add` function to increment the value
  - go run main.go call -app adder -func Add -args 2
- Call the `Value` function to see the result
  - go run main.go call -app adder -func Value
- The app state was persisted and updated. How does this work?

Application Creation Flow



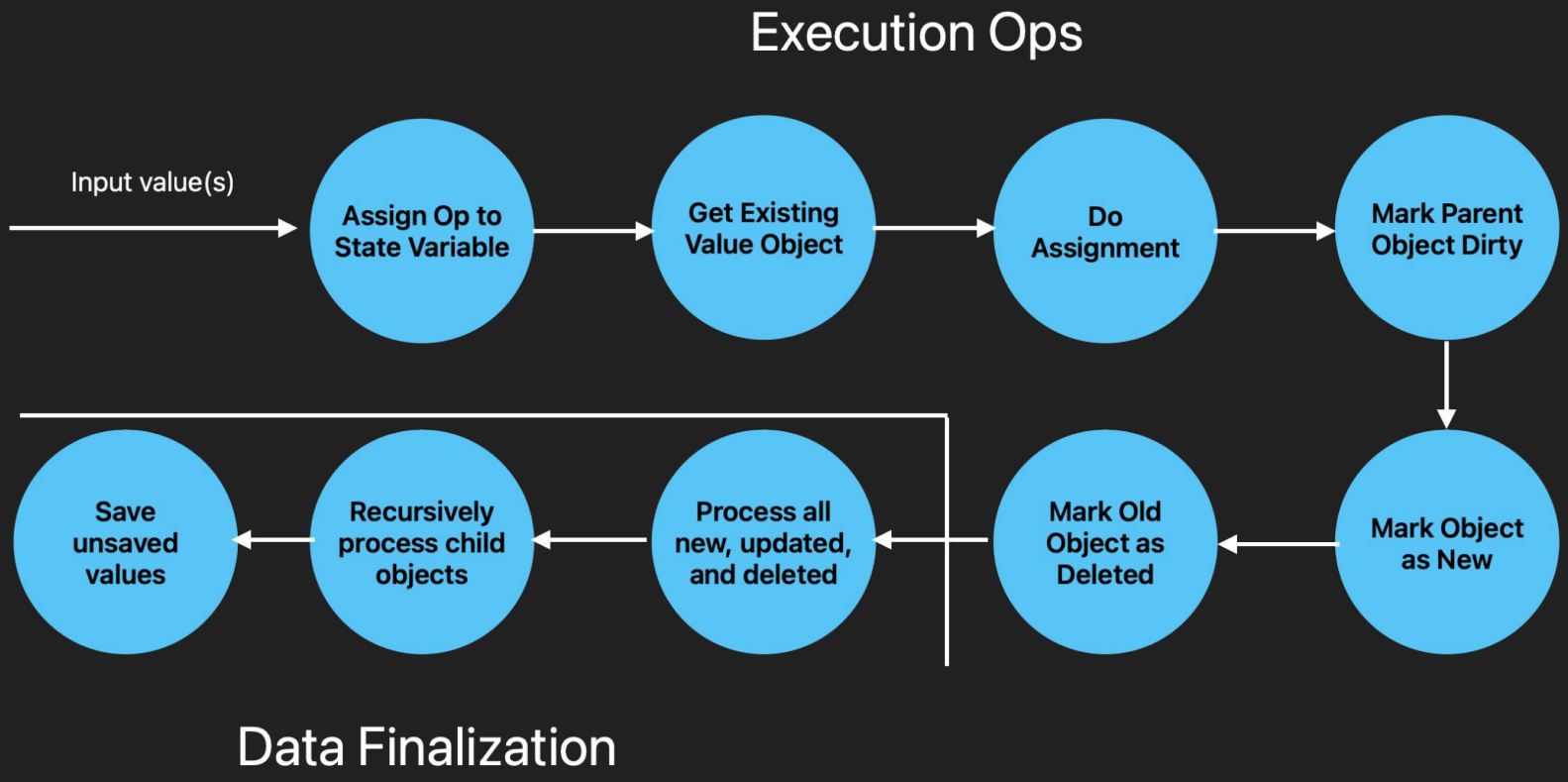
Application Call Flow



What’s special about persisting values?

- All changes made to global application variables are saved to a key-value store automatically
- Values will be loaded into memory for use the next time a call is made to the application
- Value lifecycle is managed
  - Track number of references to a value
  - Automatically delete a value if the reference count reaches zero

Application State Persistence Flow



How is persisted data structured?

- Data is represented as objects
- Each has a unique object ID
- Object values are types like:
  - Array
  - Struct
  - Map
  - Block (list of values in a code block, in this case the global var block)
- Primitive types are saved as a part of an object
- For example, an application’s primitive state variables are persisted as part of a slice (array) that is defined within a code block value

State Persistence Example

```go
package my

type Info struct {
	Name  FullName
	Age   int
	Email string
}

type FullName struct {
	First  string
	Middle string
	Last   string
}
```

```go
var info Info

func SetInfo(
	firstName,
	middleName,
	lastName string,
	age int,
	email string,
) {
	info = Info{
		Name: FullName{First: firstName, Middle: middleName, Last: lastName},
		Age: age,
		Email: email,
	}
}

func UpdateName(first, middle, last string) {
	info.Name.First = first
	info.Name.Middle = middle
	info.Name.Last = last
}
```

Hands-on: Phase 3

- Let’s use an HTTP server to route requests to the VM
- go run main.go
- Starts an HTTP server on port 4591; updates main.go to change port
- Navigate to localhost:4591/installer
- This is an app that is created upon DB initialization
- Add app code there; use the example in the phase-3 directory if you want
- Navigate to localhost:4591/myname:your-name-here

What just happened there?

- The package name when creating an app is used as the URL path name to access it
- Render is a specially named function that requires the signature `func (string) string`
- Render is called when only the app name is specified
  - The input would be the string after the colon in the URL path (following a convention from gno.land)
  - It returns a string that is displayed in the browser (if accessed via the browser)
  - In our case, it returns HTML and javascript that is rendered to allow a callback to the endpoint that allows us to create new applications

What is apps.CreatePort?

- The example requires the port app to be created in order to use the installer app. Why is that?
- The installer app needs to know the port that the server is listening on and this could change each time the binary is restarted
- The solution is to read the port from the application state whenever rendering the javascript returned by the installer app

```
1   package port
2
3   var number string
4
5   func Number() string {
6       return number
7   }
8
9   func Set(p string) {
10      number = p
11  }
```

Hands-on: Phase 4

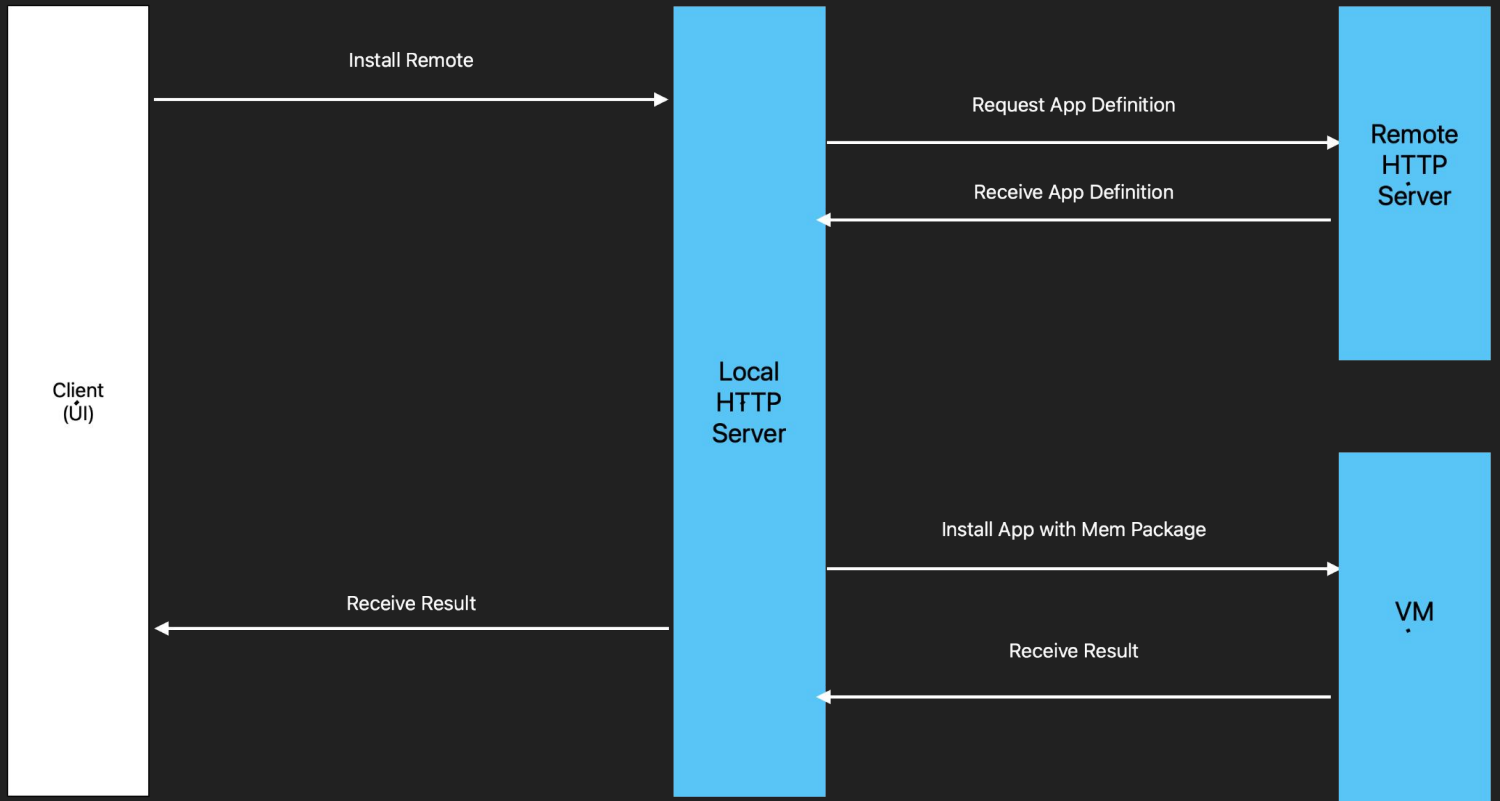
- Creating an app from the browser was cool, but copying code is a pain
- It’s much easier to point to a published app and install it locally
- Open two terminal windows and navigate to both cmd/alice and cmd/bob
- In each of them, go run main.go
- Alice is running on 4591, Bob on 4592
- Install an app on either of them via localhost:<port>/installer
- On the other, install a remote app via localhost:<port>/remoteinstaller
- Use the package name of the app created and the other’s localhost:port address

How does that work?

- Expand the VM interface to be able to retrieve the internal “mem package” representation and then create a new package using the mem package that was returned

```
type VM interface {
    ...
    QueryMemPackage(ctx context.Context, appName string) *std.MemPackage
    CreateMemPackage(ctx context.Context, memPackage *std.MemPackage) error
}
```

Install Remote App Flow



Taking it one step further

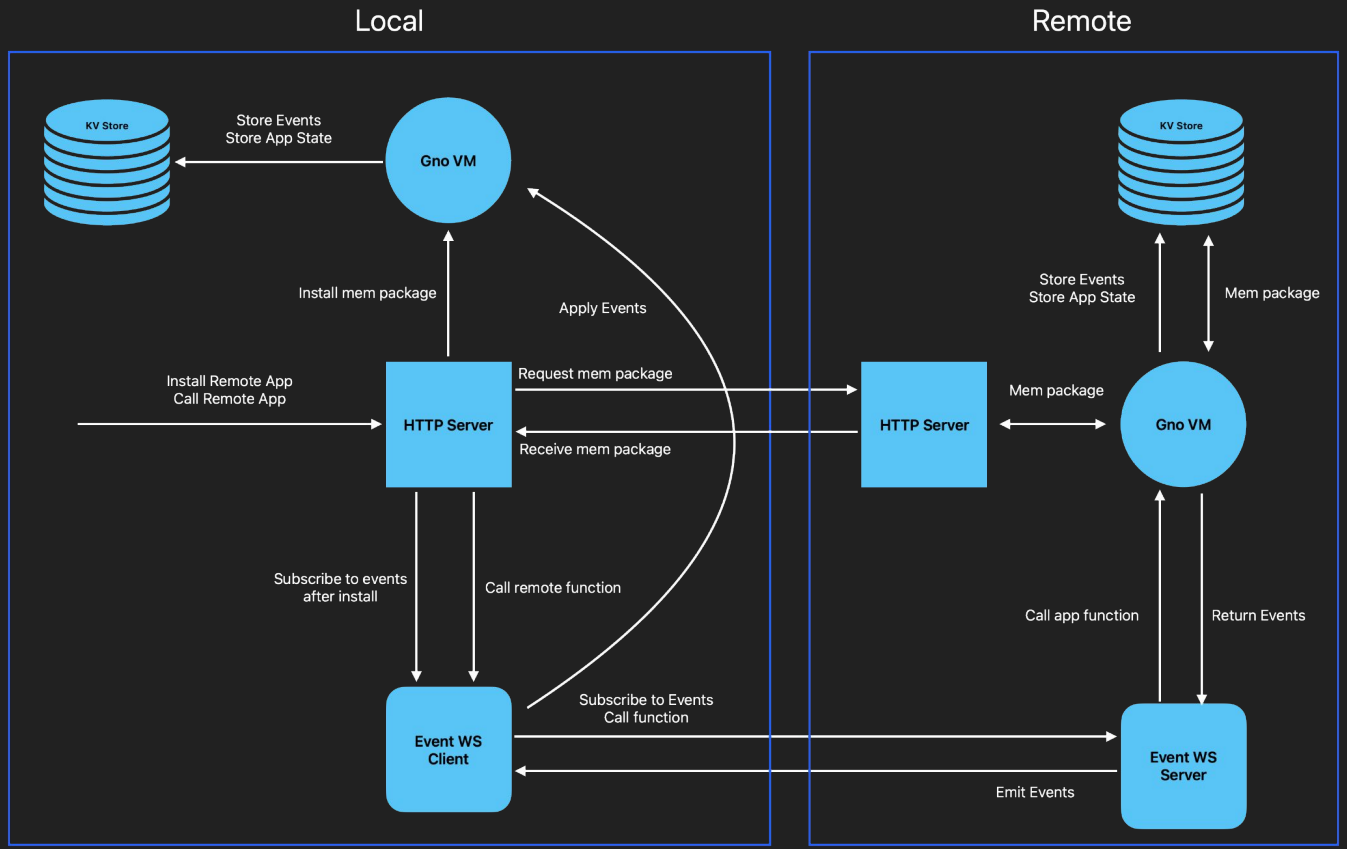
- It would be nice for users to be able to interact with the same app, meaning that they can both observe the same state
- Would require state to be synchronized between everyone using the app
- Let’s consider the simple case: an app with no other dependencies
- No need for security – this is a proof of concept
- Make it event based?
- We can bootstrap event storage on the storage already provided by the Gno VM

Event Storage

- An event contains the same information you'd provide to the VM's `Call` method when calling a function of an app
- It also includes an unsigned integer that represents the event sequence
- Ordering helps keep state in sync and keeps things deterministic

```
func NextSequence(pkgPath string) (sequence uint64)
func Store(pkgPath string, sequence uint64, funcName, args string) (sequence uint64)
func Get(pkgPath string, start, end uint64) (jsonEncodedEvents string)
```

Event-based App Architecture



Hands-on: Phase 5

- Start up both Alice and Bob from within their respective directories
  - go run main.go
- We will not use the UI to create apps or calls
  - See included postman package
  - Alternately see the curl commands in the requests file
- Alice is running on 4591 and Bob on 4592
- Create the “postit” app on Alice’s machine
- Install the remote app on Bob’s machine
- Make a calls to the postit for both Alice and Bob
- Navigate to localhost:<port>/postit to observe the application state being synced between both application instances

Improvements

- This is nice but there are a lot more things that can be done
- Better event syncing – catch-up, dependencies
- Handling failures and retries
- Building in a reputation system, perhaps stored on the blockchain
- Implementing a form of consensus based on whitelists and reputation to remove some of the burden from the application host
- Data encryption

Making non-deterministic operations deterministic

- time.Now() was mentioned earlier
- It’s important to be able to complete non-deterministic operations
  - time
  - randomness
  - fetching data from 3rd parties
- Can be accomplished through splitting application calls to emit multiple events
  - A call may result in an operation off the VM
  - It posts its result to the VM, say event 1
  - The original call takes place on the state that has been modified by event 1
  - It produces two events

Takeaways

- Gno is currently only used as a part of gno.land
- Internal discussions are taking place in regards to the value of making a stand-alone gno powered application, sans blockchain
- Gno is great for ordered event based applications
  - Determinism
  - Event storage can be bootstrapped on gno storage
  - No need to manage application storage
- Applications can be installed (and eventually removed) without needing to add go code nor recompile the binary
- A user running the binary is able to interact with both local, and distributed applications while remaining in full control of their data, unlike the architecture of current web applications

Official website

gno.land

Gno Docs

docs.gno.land

Gno Playground

play.gno.land

Twitter

@_gnoland

Github

gnolang/gno