Prompt scavenger

Welcome to the world of Prompt Scavenger, a game where you'll be using Celestia's Node API and OpenAI's GPT-3.5 to decode hidden messages scattered throughout Celestia's blockchain. In this tutorial, we'll be using Golang to write the code for the game.

Through this tutorial, you'll gain experience using Celestia's Node API to fetch data from the blockchain, process it, and submit new transactions with that data. You'll also learn how to integrate OpenAI's GPT-3.5 API to generate fun responses based on the data you've found.

So if you're ready to embark on an adventure that combines blockchain technology with the power of AI, and learn some Golang along the way, let's get started!

Dependencies

The following dependencies are needed to be installed or obtained:

- Golang, seesetting up environment
- Celestialight node
- Getting an OpenAI API Key for GPT-3.5

Install Celestia Node and run a light node

First,install the celestia-node binary .

Let's start by initializing our light node and funding our account with some tokens. We will be using the Arabica testnet for this tutorial.

```
sh celestia

light

init

--p2p.network

arabica celestia

light

init

--p2p.network

arabica
```

You will see an output ending with something looking like this:

```
2024-05-22T14:15:49.554+0200 INFO node nodebuilder/init.go:211 NO KEY FOUND IN STORE, GENERATING NEW KEY... 2024-05-22T14:15:49.564+0200 INFO node nodebuilder/init.go:216 NEW KEY GENERATED...

NAME: my_celes_key ADDRESS: celestia1hn25k7gkfq0fy5a0vmphs6mjma2de74gsn36ef MNEMONIC (save this somewhere safe!!!):
```

2024-05-22T14:15:49.554+0200 INFO node nodebuilder/init.go:211 NO KEY FOUND IN STORE, GENERATING NEW KEY... 2024-05-22T14:15:49.564+0200 INFO node nodebuilder/init.go:216 NEW KEY GENERATED...

NAME: my_celes_key ADDRESS: celestia1hn25k7gkfq0fy5a0vmphs6mjma2de74gsn36ef MNEMONIC (save this somewhere safe!!!):

To fund your account, copy the address from the "ADDRESS" log and paste it in theArabica Faucet to request tokens.

While waiting for our account to be funded, we can start our light node.

```
sh celestia

light

start
```

--core.ip

validator-1.celestia-arabica-11.com

--p2p.network

arabica

--rpc.skip-auth celestia

light

start

--core.ip

validator-1.celestia-arabica-11.com

--p2p.network

arabica

--rpc.skip-auth You should now have a running light node on your machine. The rest of the tutorial will assume you will be building the script and running it where the light node is in your localhost.

We can now check if your account has been successfully funded by running the following command. If your balance is still 0, wait a few seconds and try again.

sh celestia

state

balance

--node.store

~/.celestia-light-arabica-11 celestia

state

balance

--node.store

~/.celestia-light-arabica-11 TIP

Make sure you run this command in a different terminal window because the node has to be running for it to work.

OpenAI key

VisitOpenAI to sign up for an account and generate an API key. order to sign up for an account and generate an OpenAI API key. The key will be needed to communicate with OpenAI.

Once you have created an API key, set it as an environment variable with the following command, pasting in your own key:

sh export OPENAI_KEY = sk-proj-xxxxxxxxxxxxxxxxxxxxxxxxx export OPENAI_KEY = sk-proj-xxxxxxxxxxxxxxxxxxxxxxxxx

Building the Prompt Scavenger

Initialize your Go project

To initialize your go project, run the following commands:

sh mkdir

test_scavenger cd

test_scavenger go

mod

init

prompt-scavenger go

get

github.com/celestiaorg/celestia-openrpc go

get

github.com/sashabaranov/go-openai mkdir

test_scavenger cd

test_scavenger go

mod

init

prompt-scavenger go

get

github.com/celestiaorg/celestia-openrpc go

get

github.com/sashabaranov/go-openai This will set up a go project in a new directory and download the required modules.

Build your import statements

Inside the directory, create amain.go file and setup the import statements:

go package

main

import (" context " " encoding/hex " " fmt " " log " " os "

nodeclient

" github.com/celestiaorg/celestia-openrpc " " github.com/celestiaorg/celestia-openrpc/types/blob " " github.com/celestiaorg/celestia-openrpc/types/share " openai

" github.com/sashabaranov/go-openai ")

func

main () { // TODO: // - [ ] Load program arguments // - [ ] Initialize the node API client // - [ ] Create a namespace ID // - [ ] Create and submit a blob // - [ ] Retrieve the blob from the network // - [ ] Prompt chatgpt with the retrieved blob data } package

main

import (" context " " encoding/hex " " fmt " " log " " os "

nodeclient

" github.com/celestiaorg/celestia-openrpc " " github.com/celestiaorg/celestia-openrpc/types/blob " " github.com/celestiaorg/celestia-openrpc/types/share " openai

" github.com/sashabaranov/go-openai ")

func

main () { // TODO: // - [ ] Load program arguments // - [ ] Initialize the node API client // - [ ] Create a namespace ID // - [ ] Create and submit a blob // - [ ] Retrieve the blob from the network // - [ ] Prompt chatgpt with the retrieved blob data } Here we set up all required libraries we need to use plus themain function that we will use for our program. function that we will use for our program.

TIP

Depending on your IDE, unused import statements may be removed every time you save the file. If this is the case, come back to this section and add them one by one as they come up in the code snippets.

Main function

Let's start populating our main function. To begin, we need to load the arguments we pass to the program. and do some sanity checks. We will then initialize the node API client.

go func

main () { ctx, cancel := context. WithCancel (context. Background ()) defer

cancel ()

// Get IP, namespace, and prompt from program arguments if

len (os.Args) !=

4 { log. Fatal ("Usage: go run main.go") } nodeIP, namespaceHex, prompt := os.Args[1], os.Args[2], os.Args[3]

// We pass an empty string as the jwt token, since we // disabled auth with the --rpc.skip-auth flag client, err := nodeclient. NewClient (ctx, nodeIP, "") if err !=

nil { log. Fatalf ("Failed to create client: %v " , err) } defer client. Close ()

// TODO: // - [X] Load program arguments // - [X] Initialize the node API client // - [ ] Create a namespace ID // - [ ] Create and submit a blob // - [ ] Retrieve the blob from the network // - [ ] Prompt chatgpt with the retrieved blob data } func

main () { ctx, cancel := context. WithCancel (context. Background ()) defer

cancel ()

// Get IP, namespace, and prompt from program arguments if

len (os.Args) !=

4 { log. Fatal ("Usage: go run main.go") } nodeIP, namespaceHex, prompt := os.Args[1], os.Args[2], os.Args[3]

// We pass an empty string as the jwt token, since we // disabled auth with the --rpc.skip-auth flag client, err := nodeclient. NewClient (ctx, nodeIP, "") if err !=

nil { log. Fatalf ("Failed to create client: %v " , err) } defer client. Close ()

// TODO: // - [X] Load program arguments // - [X] Initialize the node API client // - [ ] Create a namespace ID // - [ ] Create and submit a blob // - [ ] Retrieve the blob from the network // - [ ] Prompt chatgpt with the retrieved blob data } Next, we need to create some utility functions that will help us with our next TODO items.

Utility functions

First, we need a function to convert a hex string to aNamespaceID type that is used for blob creation. This is needed because the namespace we pass in the program arguments will be in hexadecimal format.

go // createNamespaceID converts a hex string to a NamespaceID func

createNamespaceID (nIDString string) (share.Namespace, error) { // First, we parse the passed hex string into a []byte slice namespaceBytes, err := hex. DecodeString (nIDString) if err !=

nil { return

nil , fmt. Errorf ("error decoding hex string: %w " , err) }

// Next, we create a new NamespaceID using the parsed bytes return share. NewBlobNamespaceV0 (namespaceBytes) } // createNamespaceID converts a hex string to a NamespaceID func

createNamespaceID (nIDString string) (share.Namespace, error) { // First, we parse the passed hex string into a []byte slice namespaceBytes, err := hex. DecodeString (nIDString) if err !=

nil { return

nil , fmt. Errorf ("error decoding hex string: %w " , err) }

// Next, we create a new NamespaceID using the parsed bytes return share. NewBlobNamespaceV0 (namespaceBytes) }
Next, we need a utility that takes the namespace generated bycreateNamespaceID and constructs and submits a blob to the network.

If successful, it returns the created blob, the height at which it was posted, and an empty error. Otherwise, only the error field is populated.

go // createAndSubmitBlob creates a new blob and submits it to the network. func

createAndSubmitBlob (ctx context.Context, client * nodeclient.Client, ns share.Namespace, payload string ,) (* blob.Blob, uint64 , error) { // First we can create the blob using the namespace and payload. createdBlob, err := blob. NewBlobV0 (ns, [] byte (payload)) if err !=

nil { return

nil , 0 , fmt. Errorf ("Failed to create blob: %w " , err) }

// After we've created the blob, we can submit it to the network. // Here we use the default gas price. height, err := client.Blob. Submit (ctx, [] * blob.Blob{createdBlob}, blob. DefaultGasPrice ()) if err !=

nil { return

nil , 0 , fmt. Errorf ("Failed to submit blob: %v " , err) }

log. Printf ("Blob submitted successfully at height: %d ! \n " , height) log. Printf ("Explorer link: https://arabica.celenium.io/block/ %d

\n " , height)

return createdBlob, height, nil } // createAndSubmitBlob creates a new blob and submits it to the network. func

createAndSubmitBlob (ctx context.Context, client * nodeclient.Client, ns share.Namespace, payload string ,) (* blob.Blob, uint64 , error) { // First we can create the blob using the namespace and payload. createdBlob, err := blob. NewBlobV0 (ns, [] byte (payload)) if err !=

nil { return

nil , 0 , fmt. Errorf ("Failed to create blob: %w " , err) }

// After we've created the blob, we can submit it to the network. // Here we use the default gas price. height, err := client.Blob. Submit (ctx, [] * blob.Blob{createdBlob}, blob. DefaultGasPrice ()) if err !=

nil { return

nil , 0 , fmt. Errorf ("Failed to submit blob: %v " , err) }

log. Printf ("Blob submitted successfully at height: %d ! \n " , height) log. Printf ("Explorer link: https://arabica.celenium.io/block/ %d

\n " , height)

return createdBlob, height, nil } With our updated main function, we can now call these utility functions to check off our next TODO items.

go func

main () { ctx, cancel := context. WithCancel (context. Background ()) defer

cancel ()

// Get IP, namespace, and prompt from program arguments if

len (os.Args) !=

4 { log. Fatal ("Usage: go run main.go") } nodeIP, namespaceHex, prompt := os.Args[1], os.Args[2], os.Args[3]

// We pass an empty string as the jwt token, since we // disabled auth with the --rpc.skip-auth flag client, err := nodeclient. NewClient (ctx, nodeIP, "") if err !=

nil { log. Fatalf ("Failed to create client: %v " , err) } defer client. Close ()

// Next, we convert the namespace hex string to the // concrete NamespaceID type namespaceID, err :=

createNamespaceID (namespaceHex) if err !=

nil { log. Fatalf ("Failed to decode namespace: %v " , err) }

// We can then create and submit a blob using the NamespaceID and our prompt. createdBlob, height, err :=

createAndSubmitBlob (ctx, client, namespaceID, prompt) if err !=

nil { log. Fatal (err) }

// Now we will fetch the blob back from the network, using the height, namespace, and blob commitment. fetchedBlob, err := client.Blob. Get (ctx, height, namespaceID, createdBlob.Commitment) if err !=

nil { log. Fatalf ("Failed to fetch blob: %v " , err) }

log. Printf ("Fetched blob: %s\n " , string (fetchedBlob.Data))

// TODO: // - [X] Load program arguments // - [X] Initialize the node API client // - [X] Create a namespace ID // - [X] Create and submit a blob // - [X] Retrieve the blob from the network // - [ ] Prompt chatgpt with the retrieved blob data } func

main () { ctx, cancel := context. WithCancel (context. Background ()) defer

cancel ()

// Get IP, namespace, and prompt from program arguments if

len (os.Args) !=

4 { log. Fatal ("Usage: go run main.go") } nodeIP, namespaceHex, prompt := os.Args[1], os.Args[2], os.Args[3]

// We pass an empty string as the jwt token, since we // disabled auth with the --rpc.skip-auth flag client, err := nodeclient. NewClient (ctx, nodeIP, "") if err !=

nil { log. Fatalf ("Failed to create client: %v " , err) } defer client. Close ()

// Next, we convert the namespace hex string to the // concrete NamespaceID type namespaceID, err :=

createNamespaceID (namespaceHex) if err !=

nil { log. Fatalf ("Failed to decode namespace: %v " , err) }

// We can then create and submit a blob using the NamespaceID and our prompt. createdBlob, height, err :=

createAndSubmitBlob (ctx, client, namespaceID, prompt) if err !=

nil { log. Fatal (err) }

// Now we will fetch the blob back from the network, using the height, namespace, and blob commitment. fetchedBlob, err := client.Blob. Get (ctx, height, namespaceID, createdBlob.Commitment) if err !=

nil { log. Fatalf ("Failed to fetch blob: %v " , err) }

log. Printf ("Fetched blob: %s\n " , string (fetchedBlob.Data))

// TODO: // - [X] Load program arguments // - [X] Initialize the node API client // - [X] Create a namespace ID // - [X] Create and submit a blob // - [X] Retrieve the blob from the network // - [ ] Prompt chatgpt with the retrieved blob data } TIP

Alternatively toclient.Blob.Get , you could also useclient.Blob.GetAll(ctx, height, []share.Namespace{namespaceID}) which fetches all blobs in the namespace at the given height. Now our program is able to create the namespace and blob, then submit and fetch it from the arabica network. The next step is to prompt ChatGPT with the fetched blob data.

Prompting ChatGPT

First, we need one more utility function to help us prompt GPT-3.5. It reads theOPENAI_KEY environment variable and uses it to create a new GPT-3 client, which it uses to prompt and retrieve the answer.

go // gpt3 processes a given message using GPT-3 and returns the response. func

gpt3 (ctx context.Context, msg string) (string , error) { // Set the authentication header openAIKey := os. Getenv ("OPENAI_KEY") if openAIKey ==

"" { return

"" , fmt. Errorf ("OPENAI_KEY environment variable not set") } client := openai. NewClient (openAIKey) resp, err := client. CreateChatCompletion (ctx, openai.ChatCompletionRequest{ Model: openai.GPT3Dot5Turbo, Messages: []openai.ChatCompletionMessage{ { Role: openai.ChatMessageRoleUser, Content: msg, }, }, },)

```
if err !=

nil { return

"" , fmt. Errorf ( "ChatCompletion error: %w " , err) }

return resp.Choices[ 0 ].Message.Content, nil } // gpt3 processes a given message using GPT-3 and returns the response.
func

gpt3 (ctx context.Context, msg string ) ( string , error ) { // Set the authentication header openAIKey := os. Getenv (
"OPENAI_KEY" ) if openAIKey ==

"" { return

"" , fmt. Errorf ( "OPENAI_KEY environment variable not set" ) } client := openai. NewClient (openAIKey) resp, err := client.
CreateChatCompletion ( ctx, openai.ChatCompletionRequest{ Model: openai.GPT3Dot5Turbo, Messages:
[]openai.ChatCompletionMessage{ { Role: openai.ChatMessageRoleUser, Content: msg, }, }, }, )

if err !=

nil { return

"" , fmt. Errorf ( "ChatCompletion error: %w " , err) }

return resp.Choices[ 0 ].Message.Content, nil }
```

Wrapping things up

Now, we will update ourmain function to finish our last TODO item: prompting CHATGPT with the fetched blob data.

```
go func

main () { ctx, cancel := context. WithCancel (context. Background ()) defer

cancel ()

// Get IP, namespace, and prompt from program arguments if

len (os.Args) !=

4 { log. Fatal ( "Usage: go run main.go" ) } nodeIP, namespaceHex, prompt := os.Args[ 1 ], os.Args[ 2 ], os.Args[ 3 ]

// We pass an empty string as the jwt token, since we // disabled auth with the --rpc.skip-auth flag client, err := nodeclient.
NewClient (ctx, nodeIP, "" ) if err !=

nil { log. Fatalf ( "Failed to create client: %v " , err) } defer client. Close ()

// Next, we convert the namespace hex string to the // concrete NamespaceID type namespaceID, err :=

createNamespaceID (namespaceHex) if err !=

nil { log. Fatalf ( "Failed to decode namespace: %v " , err) }

// We can then create and submit a blob using the NamespaceID and our prompt. createdBlob, height, err :=

createAndSubmitBlob (ctx, client, namespaceID, prompt) if err !=

nil { log. Fatal (err) }

// Now we will fetch the blob back from the network. fetchedBlob, err := client.Blob. Get (ctx, height, namespaceID,
createdBlob.Commitment) if err !=

nil { log. Fatalf ( "Failed to fetch blob: %v " , err) }

log. Printf ( "Fetched blob: %s\n " , string (fetchedBlob.Data)) promptAnswer, err :=

gpt3 (ctx, string (fetchedBlob.Data)) if err !=

nil { log. Fatalf ( "Failed to process message with GPT-3: %v " , err) }

log. Printf ( "GPT-3 response: %s\n " , promptAnswer) } func

main () { ctx, cancel := context. WithCancel (context. Background ()) defer
```

cancel ()

// Get IP, namespace, and prompt from program arguments if

len (os.Args) !=

4 { log. Fatal ("Usage: go run main.go") } nodeIP, namespaceHex, prompt := os.Args[1], os.Args[2], os.Args[3]

// We pass an empty string as the jwt token, since we // disabled auth with the --rpc.skip-auth flag client, err := nodeclient. NewClient (ctx, nodeIP, "") if err !=

nil { log. Fatalf ("Failed to create client: %v " , err) } defer client. Close ()

// Next, we convert the namespace hex string to the // concrete NamespaceID type namespaceID, err :=

createNamespaceID (namespaceHex) if err !=

nil { log. Fatalf ("Failed to decode namespace: %v " , err) }

// We can then create and submit a blob using the NamespaceID and our prompt. createdBlob, height, err :=

createAndSubmitBlob (ctx, client, namespaceID, prompt) if err !=

nil { log. Fatal (err) }

// Now we will fetch the blob back from the network. fetchedBlob, err := client.Blob. Get (ctx, height, namespaceID, createdBlob.Commitment) if err !=

nil { log. Fatalf ("Failed to fetch blob: %v " , err) }

log. Printf ("Fetched blob: %s\n " , string (fetchedBlob.Data)) promptAnswer, err :=

gpt3 (ctx, string (fetchedBlob.Data)) if err !=

nil { log. Fatalf ("Failed to process message with GPT-3: %v " , err) }

log. Printf ("GPT-3 response: %s\n " , promptAnswer) } And now you have the final version of the prompt scavenger!

Run the golang script with the following command:

sh go

run

main.go

< nodeI P

< namespac e

< promp t

go

run

main.go

< nodeI P

< namespac e

< promp t

For example, you could run:

sh go

run

main.go

ws://localhost:26658

ce1e5714

'What is a modular blockchain?' go

run

main.go

ws://localhost:26658

ce1e5714

'What is a modular blockchain?' After some time, it’ll post the output of the prompt you submitted to OpenAI that you pulled from Celestia’s blockchain.

Next steps

With this tutorial, you were able to construct a blob, submit it to Celestia, get it back from Celestia, decode its contents, then for added bonus, submit the message to GPT-3.5.

If you're up for a challenge, you can refer to the Node API client guide and try to implement more advanced features, such as:

- Subscribing to new prompts inside thece1e5714
- namespace, submitting each one to GPT-3.5
- Posting the responses back to Celestia under a different namespace. [[Edit this page on GitHub] Last updated: