

# Prompt scavenger

## Note

This tutorial needs to be updated. 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, see [setting up environment](#)
- Celestia [light 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 run the following commands to get our env vars setup (NOTE: For CORE\_IP you can select from the [list of available RPC endpoints on the Blockspace Race page](#) :

```
sh KEYNAME = "scavenger_key" NODETYPE = "light" NETWORK = "blockspacerace" AUTHTYPE = "admin" CORE_IP = "" KEYNAME = "scavenger_key" NODETYPE = "light" NETWORK = "blockspacerace" AUTHTYPE = "admin" CORE_IP = ""
```

Next, let's generate a wallet key for our light node:

```
sh ./cel-key
```

```
add KEYNAME --keyring-backend
```

```
test
```

```
--node.type NODETYPE --p2p.network NETWORK ./cel-key
```

```
add KEYNAME --keyring-backend
```

```
test
```

```
--node.type NODETYPE --p2p.network NETWORK
```

Be sure to save the mnemonics and your celestia1 public address in a safe place.

Then head over to our Discord Server and request tokens from the #faucet channel under Blockspace Race.

You can track receiving the funds on the explorer here [Interchain Explorer by Cosmostation](#)

Just make sure to paste your celestia1\*\*\*\* address in order to look it up.

We will be running this version of Celestia Node with Blockspace Race test network. First, let's initialize our node:

```
sh celestia
```

```
light
```

```
init
```

```
--p2p.network
```

```
blockspacerace celestia
```

```
light
```

```
init
```

```
--p2p.network
```

blockspacerace Next, we will start our node:

```
sh celestia
```

```
light
```

```
start
```

```
--core.ip CORE_IP --p2p.network NETWORK --gateway.deprecated-endpoints
```

```
--gateway
```

```
--gateway.addr
```

```
127.0.0.1
```

```
--gateway.port
```

```
26659
```

```
--keyring.accname KEYNAME celestia
```

```
light
```

```
start
```

```
--core.ip CORE_IP --p2p.network NETWORK --gateway.deprecated-endpoints
```

```
--gateway
```

```
--gateway.addr
```

```
127.0.0.1
```

```
--gateway.port
```

```
26659
```

```
--keyring.accname KEYNAME TIP
```

The `--core.ip` gRPC port defaults to 9090, so if you do not specify it in the command line, it will default to that port. You can add the port after the IP address or use the `--core.grpc.port` flag to specify another port if you prefer.

Refer to [the ports section of the celestia-node troubleshooting page](#) for information on which ports are required to be open on your machine. 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.

## Node API authentication key

In the same machine as where your Celestia light node is running, run the following:

```
sh export AUTH_TOKEN = ( celestia
```

```
NODETYPE auth AUTHTYPE ) export AUTH_TOKEN = ( celestia
```

```
NODETYPE auth AUTHTYPE ) Now run the following to get the auth token for your node:
```

```
sh echo AUTH_TOKEN echo AUTH_TOKEN This will be used for the env var file we setup later.
```

## OpenAI key

Make sure to [go over to OpenAI](#) in order to sign up to an account and generate an OpenAI API key. The key will be needed to communicate with OpenAI.

## Building the Prompt Scavenger

We will first need to setup a .env file with the following pasted in:

```
txt NODE_RPC_IP="http://localhost:26658" NODE_JWT_TOKEN="" OPENAI_KEY=""
```

```
NAMESPACE_ID="00000000ce1e571a" NODE_RPC_IP="http://localhost:26658" NODE_JWT_TOKEN="" OPENAI_KEY=""
```

NAMESPACE\_ID="00000000ce1e571a" The OPENAI\_KEY is the API key you got from Open AI. For NODE\_RPC\_IP , it is assumed to be the local host but it can point to a light node that is remote. NODE\_JWT\_TOKEN is the AUTH\_TOKEN you generated earlier. For Namespace ID, we made a sample one for you to use but you can generate your own.

Now, let's build!

## Copy over Go files

Run the following:

```
sh mkdir
```

```
test_scavenger git
```

```
clone
```

```
https://github.com/celestiaorg/PromptScavenger.git cp
```

```
PromptScavenger/go.mod
```

```
test_scavenger/ cp
```

```
PromptScavenger/go.sum
```

```
test_scavenger/ cd
```

```
test_scavenger mkdir
```

```
test_scavenger git
```

```
clone
```

```
https://github.com/celestiaorg/PromptScavenger.git cp
```

```
PromptScavenger/go.mod
```

```
test_scavenger/ cp
```

```
PromptScavenger/go.sum
```

```
test_scavenger/ cd
```

test\_scavenger This will copy over the required go.sum and go.mod files to a new directory we will use to import the Node API golang library.

## Build your import statements

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

```
go package
```

```
main
```

```
import ( " context " " fmt " " log " " os "
```

```
" github.com/celestiaorg/celestia-node/api/rpc/client " nodeheader
```

```
" github.com/celestiaorg/celestia-node/header " " github.com/celestiaorg/nmt/namespace " " github.com/joho/godotenv " cosmosmath
```

```
" cosmossdk.io/math " openai
```

```
" github.com/sashabaranov/go-openai " " encoding/base64 " " encoding/hex " )
```

```
func
```

```
main () { // TODO } package
```

```
main
```

```
import ( " context " " fmt " " log " " os "
```

```
" github.com/celestiaorg/celestia-node/api/rpc/client " nodeheader
```

```
" github.com/celestiaorg/celestia-node/header " " github.com/celestiaorg/nmt/namespace " " github.com/joho/godotenv "
cosmosmath
```

```
" cosmos-sdk.io/math " openai
```

```
" github.com/sashabaranov/go-openai " " encoding/base64 " " encoding/hex " )
```

```
func
```

```
main () { // TODO } Here we setup all required libraries we need to use plus the main function that we will use for our game.
```

## Helpful functions

First we will need to create some helpful functions that you will need later.

```
go // loadEnv loads environment variables from the .env file. func
```

```
loadEnv () { err := godotenv. Load ( ".env" ) if err !=
```

```
nil { log. Fatal ( "Error loading .env file" ) } } // loadEnv loads environment variables from the .env file. func
```

```
loadEnv () { err := godotenv. Load ( ".env" ) if err !=
```

```
nil { log. Fatal ( "Error loading .env file" ) } } The loadEnv function allows us to load up our .env file which has all the
necessary env vars needed.
```

Next, let's create a helper function that allows us to load an instance of the Celestia Node client given the correct env vars passed to it:

```
go // createClient initializes a new Celestia node client. func
```

```
createClient (ctx context.Context) * client.Client { nodeRPCIP := os. Getenv ( "NODE_RPC_IP" ) jwtToken := os. Getenv (
"NODE_JWT_TOKEN" )
```

```
rpc, err := client. NewClient (ctx, nodeRPCIP, jwtToken) if err !=
```

```
nil { log. Fatal ( "Error creating client: %v " , err ) }
```

```
return rpc } // createClient initializes a new Celestia node client. func
```

```
createClient (ctx context.Context) * client.Client { nodeRPCIP := os. Getenv ( "NODE_RPC_IP" ) jwtToken := os. Getenv (
"NODE_JWT_TOKEN" )
```

```
rpc, err := client. NewClient (ctx, nodeRPCIP, jwtToken) if err !=
```

```
nil { log. Fatal ( "Error creating client: %v " , err ) }
```

```
return rpc } As you can see, here the Celestia Node client takes in the Node RPC IP and the JWT Token we setup before.
```

Now, if we go back to our main function, we can do the following to setup and load our env and client:

```
go func
```

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

```
cancel () loadEnv ()
```

```
// Close the client when you are finished client. Close () } func
```

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

```
cancel () loadEnv ()
```

```
// Close the client when you are finished client. Close () } Here, we setup a workflow that allows us to load our env vars,
instantiate the client with it, then close the client.
```

Now, let's build some more helpful functions:

```
go func
```

```
createNamespaceID () [] byte { nIDString := os. Getenv ( "NAMESPACE_ID" ) data, err := hex. DecodeString (nIDString) if
err !=
```

```
nil { log. FataIf ( "Error decoding hex string:" , err) } // Encode the byte array in Base64 base64Str := base64.StdEncoding.
EncodeToString (data) namespaceID, err := base64.StdEncoding. DecodeString (base64Str) if err !=
```

```
nil { log. FataIf ( "Error decoding Base64 string:" , err) } return namespaceID } func
```

```
createNamespaceID () [] byte { nIDString := os. Getenv ( "NAMESPACE_ID" ) data, err := hex. DecodeString (nIDString) if
err !=
```

```
nil { log. FataIf ( "Error decoding hex string:" , err) } // Encode the byte array in Base64 base64Str := base64.StdEncoding.
EncodeToString (data) namespaceID, err := base64.StdEncoding. DecodeString (base64Str) if err !=
```

nil { log. FataIf ( "Error decoding Base64 string:" , err) } return namespaceID } Here, we are creating a helpful function called createNameSpaceID that given a string for a namespace ID, it can decode the hex string, encode it after to a byte array, then decode it as a base64 string which is needed by Node API.

We will need to create just a few more functions before we wrap up things.

```
go // postDataAndGetHeight submits a new transaction with the // provided data to the Celestia node. func
```

```
postDataAndGetHeight (client * client.Client, namespaceID namespace.ID, payLoad [ ] byte , fee cosmosmath.Int, gasLimit
uint64 ) uint64 { response, err := client.State. SubmitPayForBlob (context. Background (), namespa celID, payLoad, fee,
gasLimit) if err !=
```

```
nil { log. FataIf ( "Error submitting pay for blob: %v " , err) } fmt. Printf ( "Got output: %v " , response) height :=
```

```
uint64 (response.Height) fmt. Printf ( "Height that data was submitted at: %v " , height) return height } //
postDataAndGetHeight submits a new transaction with the // provided data to the Celestia node. func
```

```
postDataAndGetHeight (client * client.Client, namespaceID namespace.ID, payLoad [ ] byte , fee cosmosmath.Int, gasLimit
uint64 ) uint64 { response, err := client.State. SubmitPayForBlob (context. Background (), namespa celID, payLoad, fee,
gasLimit) if err !=
```

```
nil { log. FataIf ( "Error submitting pay for blob: %v " , err) } fmt. Printf ( "Got output: %v " , response) height :=
```

```
uint64 (response.Height) fmt. Printf ( "Height that data was submitted at: %v " , height) return height } In the
function postDataAndGetHeight , we show you how to submit a message to a specific namespace ID provided on Celestia.
After a successful submission, the function returns back to you the block height it was submitted at.
```

Next, implement the following function:

```
go func
```

```
getDataAsPrompt (client * client.Client, height uint64 , namespaceID namespace.ID) string { headerParam :=
```

```
getHeader (client, height) response, err := client.Share. GetSharesByNamespace (context. Background (), hea
derParam.DAH, namespaceID) if err !=
```

```
nil { log. FataIf ( "Error getting shares by namespace data for block height: %v . Error is %v " , height, err) } var dataString
string for _, shares :=
```

```
range response { for _, share :=
```

```
range shares.Shares { dataString =
```

```
string (share[ 8 :]) } } return dataString } func
```

```
getDataAsPrompt (client * client.Client, height uint64 , namespaceID namespace.ID) string { headerParam :=
```

```
getHeader (client, height) response, err := client.Share. GetSharesByNamespace (context. Background (), hea
derParam.DAH, namespaceID) if err !=
```

```
nil { log. FataIf ( "Error getting shares by namespace data for block height: %v . Error is %v " , height, err) } var dataString
string for _, shares :=
```

```
range response { for _, share :=
```

```
range shares.Shares { dataString =
```

```
string (share[ 8 :]) } } return dataString } In the function getDataAsPrompt , we show a helpful function that given a particular
block height and a namespace ID, it can return back the block data (called shares here) which we then convert to a string
and return it back.
```

Finally, we implement a GPT-3.5 helper function that given a prompt, it returns back a statement:

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

```
gpt3 (msg string) { // Set the authentication header openAIKey := os.Getenv ( "OPENAI_KEY" ) client := openai. NewClient (openAIKey) resp, err := client. CreateChatCompletion ( context. Background (), openai.ChatCompletionRequest{ Model: openai.GPT3Dot5Turbo, Messages: []openai.ChatCompletionMessage{ { Role: openai.ChatMessageRoleUser, Content: msg, }, }, }, )
```

```
if err !=
```

```
nil { fmt. Printf ( "ChatCompletion error: %v\n " , err) return } fmt. Println (resp.Choices[ 0 ].Message.Content) } // gpt3 processes a given message using GPT-3 and prints the response. func
```

```
gpt3 (msg string) { // Set the authentication header openAIKey := os.Getenv ( "OPENAI_KEY" ) client := openai. NewClient (openAIKey) resp, err := client. CreateChatCompletion ( context. Background (), openai.ChatCompletionRequest{ Model: openai.GPT3Dot5Turbo, Messages: []openai.ChatCompletionMessage{ { Role: openai.ChatMessageRoleUser, Content: msg, }, }, }, )
```

```
if err !=
```

```
nil { fmt. Printf ( "ChatCompletion error: %v\n " , err) return } fmt. Println (resp.Choices[ 0 ].Message.Content) }
```

## Wrapping up the functions

Now, we will update our main function to include the logic from the functions we built where we show you how to do the following:

- Instantiate namespace ID, fee, gas limit and GPT prompt
- Submit the GPT prompt as a PayForBlob transaction and then get back the Block Height
- Get Back the Prompt from that Block Height as a Data Share and convert it to a string and return it
- Submit that string to the GPT function to get a prompt output

go func

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

```
cancel () loadEnv () var namespaceID namespace.ID =
```

```
createNamespaceID () client :=
```

```
createClient (ctx) var gasLimit uint64
```

```
=
```

```
6000000 fee := cosmosmath. NewInt ( 10000 ) var gptPrompt string
```

```
=
```

```
"What are modular blockchains?" prompt := [] byte { 0x 00 , 0x 01 , 0x 02 } prompt =
```

```
append (prompt, [] byte (gptPrompt) ... ) height :=
```

```
postDataAndGetHeight (client, namespaceID, prompt, fee, gasLimit) promptString :=
```

```
getDataAsPrompt (client, height, namespaceID) gpt3 (promptString) // Close the client when you are finished client. Close () } func
```

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

```
cancel () loadEnv () var namespaceID namespace.ID =
```

```
createNamespaceID () client :=
```

```
createClient (ctx) var gasLimit uint64
```

```
=
```

```
6000000 fee := cosmosmath. NewInt ( 10000 ) var gptPrompt string
```

```
=
```

```
"What are modular blockchains?" prompt := [] byte { 0x 00 , 0x 01 , 0x 02 } prompt =
```

```
append (prompt, [] byte (gptPrompt) ... ) height :=
```

```
postDataAndGetHeight (client, namespaceID, prompt, fee, gasLimit) promptString :=
```

```
getDataAsPrompt (client, height, namespaceID) gpt3 (promptString) // Close the client when you are finished client. Close ()  
} And now you have the final version of the game!
```

Run the go lang script with the following command:

```
sh go
```

```
run
```

```
main.go go
```

```
run
```

main.go 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 PFB transaction, submit it to Celestia, get it back from Celestia and decode its contents, then for added bonus, submit the message to GPT-3.5.

For the next steps, we will be releasing quests to this tutorial where users will need to complete challenges that help familiarize them with the Celestia Data Availability layer. [] [[Edit this page on GitHub](#)] Last updated: [Previous page Node RPC CLI tutorial](#) [Next page Overview of Blobstream](#) []