

Rust client library tutorial

This section tutorial will guide you through using the most common RPC endpoints with [lumina](#) 's rust client library.

Install [dependencies](#) and [celestia-node](#) if you have not already.

Project setup

To start, add `celestia_rpc` and `celestia_types` as a dependency to your project:

```
bash cargo
```

```
add
```

```
celestia_rpc
```

```
celestia_types cargo
```

```
add
```

```
celestia_rpc
```

`celestia_types` To use the following methods, you will need the node URL and your auth token. To get your auth token, see this [guide](#) . To run your node without an auth token, you can use the `--rpc.skip-auth` flag when starting your node. This allows you to pass an empty string as your auth token.

The default URL is `http://localhost:26658` . If you would like to use subscription methods, such as `SubscribeHeaders` below, you must use the `ws` protocol in place of `http` : `ws://localhost:26658` .

Submitting and retrieving blobs

The [blob.Submit](#) method takes an array of blobs and a gas price, returning the height the blob was successfully posted at.

- The namespace can be generated with `Namespace::new_v0`
- .
- The blobs can be generated with `Blob::new`
- .
- You can set `GasPrice::default()`
- as the gas price to have celestia-node automatically determine an appropriate gas price.

The [blob.GetAll](#) method takes a height and array of namespaces, returning the array of blobs found in the given namespaces.

```
rust use
```

```
celestia_rpc :: { BlobClient , Client , HeaderClient , ShareClient }; use
```

```
celestia_types :: blob :: GasPrice ; use
```

```
celestia_types :: { nmt :: Namespace , Blob , ExtendedDataSquare };
```

```
async
```

```
fn
```

```
submit_blob (url :
```

```
& str , token :
```

```
& str ) { let client =
```

```
Client :: new (url, Some (token)) .await . expect ( "Failed creating rpc client" );
```

```
// let's use the DEADBEEF namespace let namespace =
```

```
Namespace :: new_v0 ( [ 0xDE , 0xAD , 0xBE , 0xEF ] ) . expect ( "Invalid namespace" );
```

```
// create a blob let blob =
```

```
Blob :: new (namespace, b"Hello, World!" . to_vec ()) . expect ( "Blob creation failed" );
```

```
// submit the blob to the network let height = client . blob_submit ( & [blob . clone ()], GasPrice :: default ()) .await . expect (
"Failed submitting blob" );

println! ( "Blob was included at height {}", height);

// fetch the blob back from the network let retrieved_blobs = client . blob_get_all (height, & [namespace]) .await . expect (
"Failed to retrieve blobs" );

assert_eq! (retrieved_blobs . len (), 1 ); assert_eq! (retrieved_blobs[ 0 ] . data, b"Hello, World!" ); assert_eq!
(retrieved_blobs[ 0 ] . commitment, blob . commitment); } use

celestia_rpc :: { BlobClient , Client , HeaderClient , ShareClient }; use

celestia_types :: blob :: GasPrice ; use

celestia_types :: { nmt :: Namespace , Blob , ExtendedDataSquare };

async

fn

submit_blob (url :

& str , token :

& str ) { let client =

Client :: new (url, Some (token)) .await . expect ( "Failed creating rpc client" );

// let's use the DEADBEEF namespace let namespace =

Namespace :: new_v0 ( & [ 0xDE , 0xAD , 0xBE , 0xEF ]) . expect ( "Invalid namespace" );

// create a blob let blob =

Blob :: new (namespace, b"Hello, World!" . to_vec ()) . expect ( "Blob creation failed" );

// submit the blob to the network let height = client . blob_submit ( & [blob . clone ()], GasPrice :: default ()) .await . expect (
"Failed submitting blob" );

println! ( "Blob was included at height {}", height);

// fetch the blob back from the network let retrieved_blobs = client . blob_get_all (height, & [namespace]) .await . expect (
"Failed to retrieve blobs" );

assert_eq! (retrieved_blobs . len (), 1 ); assert_eq! (retrieved_blobs[ 0 ] . data, b"Hello, World!" ); assert_eq!
(retrieved_blobs[ 0 ] . commitment, blob . commitment); }
```

Subscribing to new headers

You can subscribe to new headers using the [header.Subscribe](#) method. This method returns a `Subscription` that will receive new headers as they are produced. In this example, we will fetch all blobs at the height of the new header in the `0xDEADBEEF` namespace.

```
rust async

fn

subscribe_headers (url :

& str , token :

& str ) { let client =

Client :: new (url, Some (token)) .await . expect ( "Failed creating rpc client" );

let

mut header_sub = client . header_subscribe () .await . expect ( "Failed subscribing to incoming headers" );

// setup the namespace we will filter blobs by let namespace =

Namespace :: new_v0 ( & [ 0xDE , 0xAD , 0xBE , 0xEF ]) . expect ( "Invalid namespace" );
```

```

while
let

Some (extended_header) = header_sub . next () .await { match extended_header { Ok (header) => { let height = header .
header . height . value (); // fetch all blobs at the height of the new header

let blobs =

match client . blob_get_all (height, & [namespace]) .await { Ok (blobs) => blobs, Err (e) => { eprintln! ( "Error fetching blobs:
{}" , e); continue ; } };

println! ( "Found {} blobs at height {} in the 0xDEADBEEF namespace" , blobs . len (), height ); } Err (e) => { eprintln! ( "Error
receiving header: {}" , e); } } } } async

fn

subscribe_headers (url :

& str , token :

& str ) { let client =

Client :: new (url, Some (token)) .await . expect ( "Failed creating rpc client" );

let

mut header_sub = client . header_subscribe () .await . expect ( "Failed subscribing to incoming headers" );

// setup the namespace we will filter blobs by let namespace =

Namespace :: new_v0 ( & [ 0xDE , 0xAD , 0xBE , 0xEF ]) . expect ( "Invalid namespace" );

while
let

Some (extended_header) = header_sub . next () .await { match extended_header { Ok (header) => { let height = header .
header . height . value (); // fetch all blobs at the height of the new header

let blobs =

match client . blob_get_all (height, & [namespace]) .await { Ok (blobs) => blobs, Err (e) => { eprintln! ( "Error fetching blobs:
{}" , e); continue ; } };

println! ( "Found {} blobs at height {} in the 0xDEADBEEF namespace" , blobs . len (), height ); } Err (e) => { eprintln! ( "Error
receiving header: {}" , e); } } } } }

```

Fetching an Extended Data Square (EDS)

You can fetch an [Extended Data Square \(EDS\)](#) using the [share.GetEDS](#) method. This method takes a header and returns the EDS at the given height.

```

rust async

fn

get_eds (url :

& str , token :

& str ) ->

ExtendedDataSquare { let client =

Client :: new (url, Some (token)) .await . expect ( "Failed creating rpc client" );

// first get the header of the block you want to fetch the EDS from let latest_header = client . header_local_head () .await .
expect ( "Failed fetching header" );

client . share_get_eds ( & latest_header ) .await . expect ( "Failed to get EDS from latest header" ) } } async

fn

```

```
get_eds (url :
& str , token :
& str ) ->
ExtendedDataSquare { let client =
Client :: new (url, Some (token)) .await . expect ( "Failed creating rpc client" );
// first get the header of the block you want to fetch the EDS from let latest_header = client . header_local_head () .await .
expect ( "Failed fetching header" );
client . share_get_eds ( & latest_header) .await . expect ( "Failed to get EDS from latest header" ) }
```

API documentation

To see the full list of available methods, see the [API documentation](#) . [\[\[Edit this page on GitHub \]](#) Last updated: [Previous page Golang client tutorial](#) [Next page Prompt Scavenger](#) [\[](#)