Using Reth Execution Extensions for next generation indexing

Alexey Shekhirin
Founding Engineer, Ithaca

What is indexing?

What is indexing?

- Ethereum nodes store history, state, and receipts – sounds like a sufficient database itself?
- History is too large
  - Impossible to get the transactions for a particular user without going through all transactions in the history
- State is hard to navigate
  - Easy to get an account balance and nonce
  - Impossible to get the ERC-20 balances for a particular address just looking at the storage slots
- Receipts & logs for the rescue!
  - Data in logs is not always enough, and you need to have auxiliary infra on top (tracing the EVM execution, shadow logs, etc.)
  - And you still need to know what receipts to query, and it gets us back to the “history is too large” problem

What is indexing?

- Enter indexers!
  - Etherscan, Dora
  - Wallets that show your token balances
  - Rollups?
- Out-of-node, off-chain infrastructure that consumes the on-chain data

How do indexers work today?

- You as an indexer developer have several ways to do it today:
  1. Consume the data via JSON RPC subscriptions
  2. Inject your code into the node sources to emit the desired data into a queue/database of your choice, and process it later
  3. Build on top of the existing node components and create your own indexing node

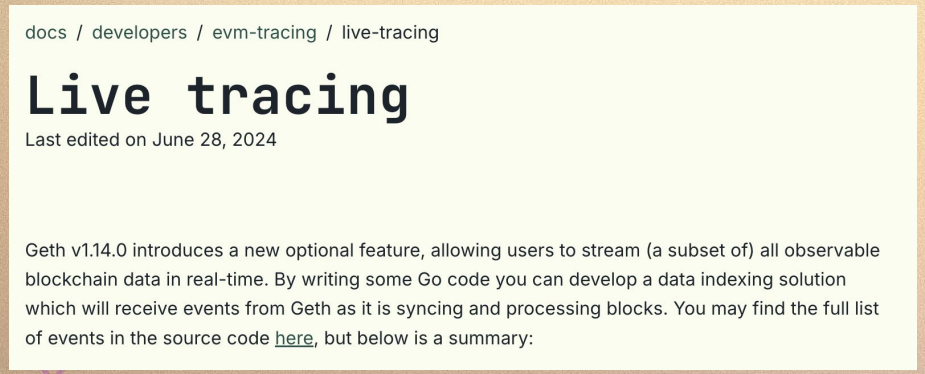
JSON RPC subscriptions

- Pros:
  - Easy
  - Every node has an eth_subscribe JSON RPC method that's part of the spec (if such exists)
- Cons:
  - Poor performance due to JSON RPC serialization and data transfer
  - Difficult to handle chain reorgs
  - Requires an ad-hoc solution that lives in a separate process

```
❯ echo 'eth_subscribe ["newHeads"]' | websocat ws://69.67.151.138:8546 --jsonrpc -n

{"jsonrpc":"2.0","id":1,"result":"0xf2a342fb8f32f3c3445e0c411ade1ae6"}

{"jsonrpc":"2.0","method":"eth_subscription","params":{"result":
{"hash":"0x090394a946b41bb941c6f52ef26f0d4b89943c158eed40bc12553bc6e9fdcae0", ... }}}
```

Inject your code into the node sources

- Pros:
  - Same process means no overhead on serialization and data transfer
  - All data is potentially accessible, even beyond the JSON RPC response fields
- Cons:
  - Maintaining a repository fork can be difficult and can lead to missing on new features
  - You're tied to the programming language the node is written in



docs / developers / evm-tracing / live-tracing

Live tracing

Last edited on June 28, 2024

Geth v1.14.0 introduces a new optional feature, allowing users to stream (a subset of) all observable blockchain data in real-time. By writing some Go code you can develop a data indexing solution which will receive events from Geth as it is syncing and processing blocks. You may find the full list of events in the source code here, but below is a summary:

Running the tracer

First compile the source by running `make geth`. Then run the following command:

```
./build/bin/geth --vmtrace supply --vmtrace.jsonconfig '{"config": "supply-logs
```

Soon you will see `supply-logs/supply.jsonl` file being populated with lines such as:

```
{"delta":973736013731113356,"reward":0,"withdrawals":466087699000000000,"burn":
{"delta":-78769000248388266,"reward":0,"withdrawals":336059502000000000,"burn"
{"delta":201614678898811488,"reward":0,"withdrawals":335502106000000000,"burn"
```

Our answer to that:
Execution Extensions

What are Execution Extensions?

- Consume a reorg-aware stream of notifications
- Get access to all node components
- Use shared memory for communication
- Compile into the same binary as Reth and run in the same process

What are Execution Extensions?

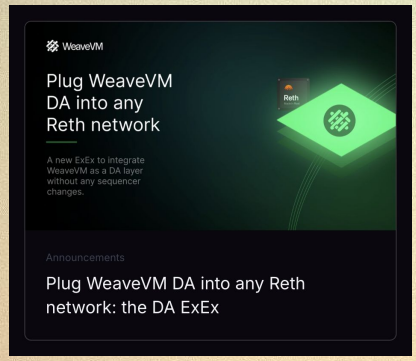
- Pros:
  - No overhead on serialization, data transfer and memory copying
  - No forking, use as a library
  - Rich access to block execution output, such as state and trie diffs
  - Full access to all node components, such as database, EVM, transaction pool, payload builder, etc.
- Cons:
  - You're tied to the programming language the node is written in (Rust ❤️)
  - ???

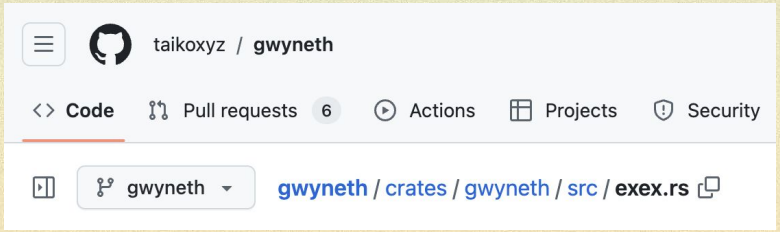
What can you do with Execution Extensions?

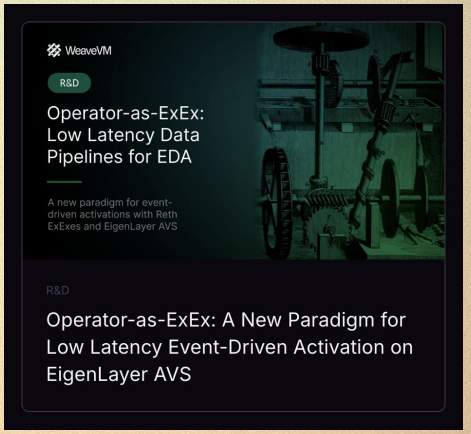
- Anything that requires deriving the on-chain data
  - Indexers
  - Rollups
  - AVSs
  - Oracles
- Use the power of Node Builder
  - Introduce a new RPC method
  - Extend the payload builder or transaction pool implementation
  - Add a new CLI argument

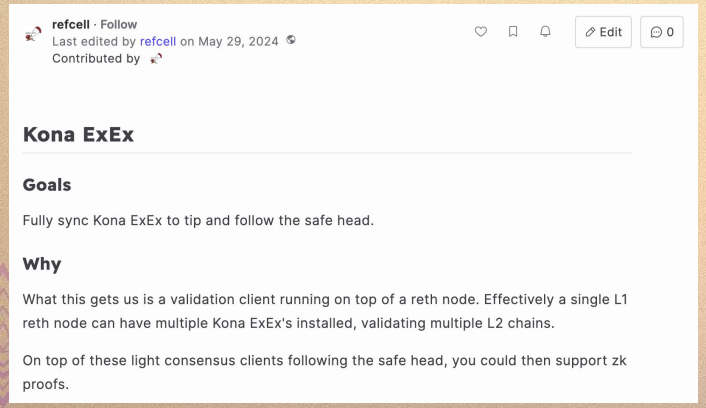
```rust
async fn exex<Node: FullNodeComponents>(mut ctx: ExExContext<Node>) -> eyre::Result<()> {
    while let Some(notification) = ctx.notifications.try_next().await? {
        match &notification {
            ExExNotification::ChainCommitted { new } => {
                info!("Received commit");
            }
            ExExNotification::ChainReorged { old, new } => {
                info!("Received reorg");
            }
            ExExNotification::ChainReverted { old } => {
                info!("Received revert");
            }
        };

        if let Some(committed_chain) = notification.committed_chain() {
            ctx.events.send(ExExEvent::FinishedHeight(committed_chain.tip().num_hash()))?;
        }
    }

    Ok(())
}

fn main() -> eyre::Result<()> {
    reth::cli::Cli::parse_args().run(|builder, _| async move {
        let handle = builder
            .node(EthereumNode::default())
            .install_exex("Minimal", exex_init)
            .launch()
            .await?;

        handle.wait_for_node_exit().await
    })
}
```

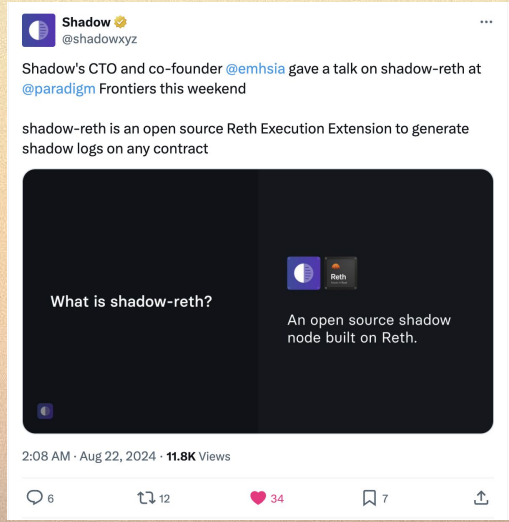
People are already building with ExExes



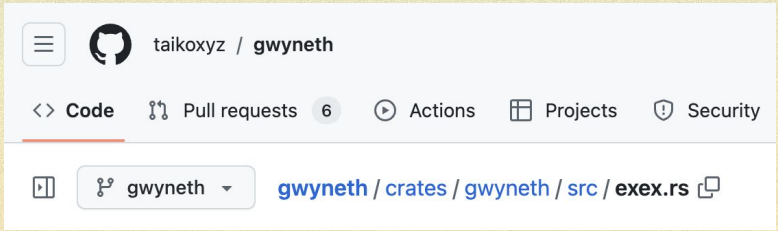


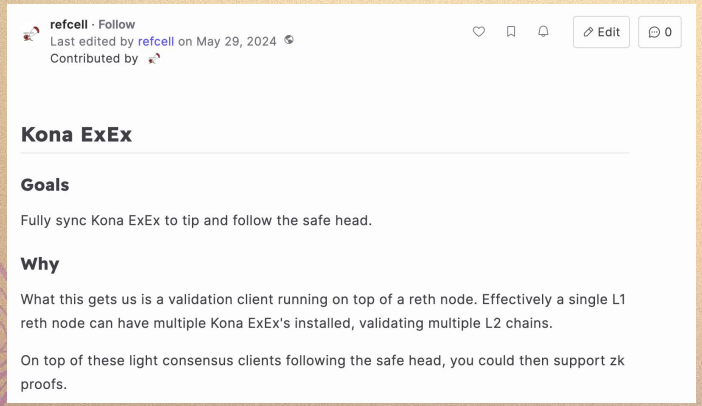






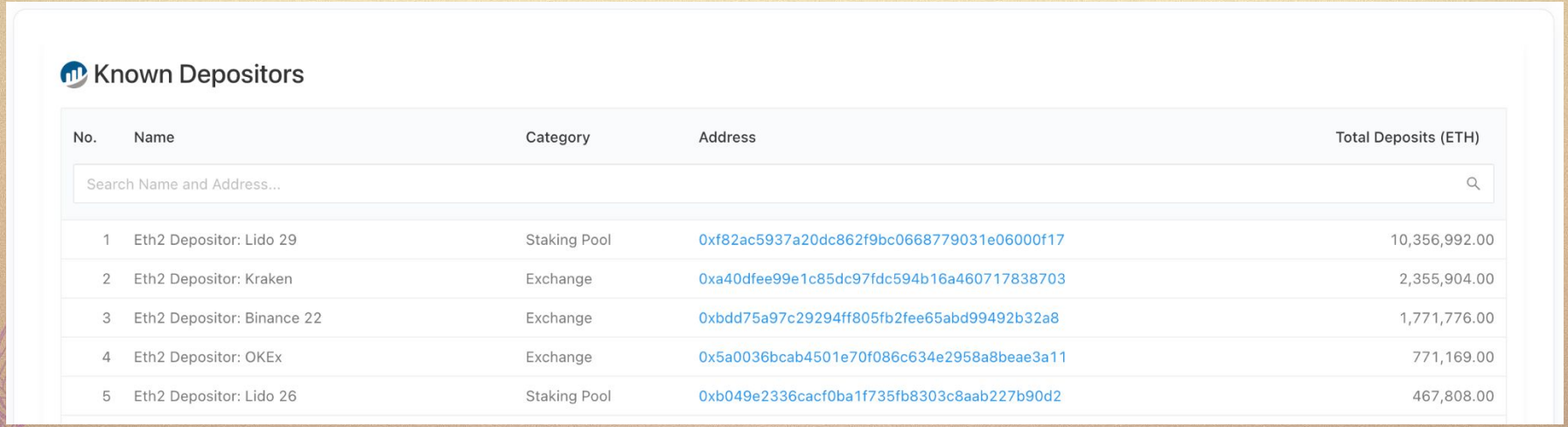
Rollups are just ExExes





Let's build an indexer

- Our goal is to index the Beacon Chain deposits and serve a list of top depositors through an API
- Deposits are recorded in the events, we can decode them
- An API will be the node's JSON RPC that we extend with our own method



Known Depositors

No.	Name	Category	Address	Total Deposits (ETH)
	Search Name and Address...			
1	Eth2 Depositor: Lido 29	Staking Pool	0xf82ac5937a20dc862f9bc0668779031e06000f17	10,356,992.00
2	Eth2 Depositor: Kraken	Exchange	0xa40dfee99e1c85dc97fdc594b16a460717838703	2,355,904.00
3	Eth2 Depositor: Binance 22	Exchange	0xbdd75a97c29294ff805fb2fee65abd99492b32a8	1,771,776.00
4	Eth2 Depositor: OKEx	Exchange	0x5a0036bcab4501e70f086c634e2958a8beae3a11	771,169.00
5	Eth2 Depositor: Lido 26	Staking Pool	0xb049e2336cacf0ba1f735fb8303c8aab227b90d2	467,808.00

The power is in the stack

Let's build an indexer

1. For listening to new blocks, we will use a custom ExEx
2. For decoding the chain events, we will use Alloy
3. For the storage, we will use SQLite
4. For the API, we will use a custom JSON RPC method
5. For bundling the ExEx and the JSON RPC together, we will use Node Builder
6. For testing, we will use cast from Foundry

```rust
async fn exex<Node: FullNodeComponents>(mut ctx: ExExContext<Node>) -> eyre::Result<()> {
    while let Some(notification) = ctx.notifications.try_next().await? {
        match &notification {
            ExExNotification::ChainCommitted { new } => {
                info!("Received commit");
            }
            ExExNotification::ChainReorged { old, new } => {
                info!("Received reorg");
            }
            ExExNotification::ChainReverted { old } => {
                info!("Received revert");
            }
        };

        if let Some(committed_chain) = notification.committed_chain() {
            ctx.events.send(ExExEvent::FinishedHeight(committed_chain.tip().num_hash()))?;
        }
    }

    Ok(())
}
```

Listening to new blocks

- The boilerplate is the same as I showed before
- The notification contains all the data that we need, which is events

```
const DEPOSIT_CONTRACT_ADDRESS: Address =
    address!("00000000219ab540356cBB839Cbe05303d7705Fa");

sol! {
    event DepositEvent(
        bytes pubkey,
        bytes withdrawal_credentials,
        bytes amount,
        bytes signature,
        bytes index
    );
}
```

Decoding the chain events

- Set the deposit contract address as a constant using the macro that represents the address in a memory-efficient way
- Generate the DepositEvent struct using the macro that parses the Solidity code and outputs the Rust code
- No codegen into separate files, all done in the same main.rs file

```
// Iterate over blocks
for block in reverted_chain.blocks_iter() {
    // Iterate over transactions
    for tx in block.transactions() {
        // Delete the deposit from the database by tx hash
    }
}
```

Storage

- Nothing fancy, SQLite for the rescue
- On every reorg
  - Delete the row by transaction hash
- On every new block:
  - Filter transactions to the deposit contract
  - Decode logs with DepositEvent
  - Insert a row with the transaction hash, transaction sender and the deposit amount

```rust
// Iterate over blocks and receipts
for (block, receipts) in committed_chain.blocks_and_receipts() {
    // Iterate over transactions and receipts
    for (tx, receipt) in block.transactions().zip(receipts) {
        // Filter only transactions to the deposit contract
        if tx.to() == Some(DEPOSIT_CONTRACT_ADDRESS) {
            // Iterate over logs
            for log in &receipt.as_ref().unwrap().logs {
                // Filter those logs that decode to the `DepositEvent`
                if let Ok(deposit) =
                    DepositEvent::decode_raw_log(log.topics(), &log.data.data, true)
                {
                    // Insert the deposit event into the database
                }
            }
        }
    }
}
```

Storage

- Nothing fancy, SQLite for the rescue
- On every reorg
  - Delete the row by transaction hash
- On every new block:
  - Filter transactions to the deposit contract
  - Decode logs with DepositEvent
  - Insert a row with the transaction hash, transaction sender and the deposit amount

```rust
#[rpc(server, namespace = "devcon")]
trait DevconRpcExtApi {
    #[method(name = "depositors")]
    async fn depositors(&self, count: usize) -> RpcResult<Vec<(Address, U256)>>;
}

#[async_trait]
impl DevconRpcExtApiServer for DevconRpcExt {
    async fn depositors(&self, count: usize) -> RpcResult<Vec<(Address, U256)>> {
        // Query depositors from highest to lowest by total amount
        let mut stmt = self.connection.prepare(
            r#"
            SELECT "from", SUM(amount) amount
            FROM deposits
            GROUP BY 1 ORDER BY 2 DESC LIMIT ?"#,
        )?;

        // Query the depositor addresses and amounts
        let depositors = stmt.query([count]);

        // Return the list of depositors as an API response
        Ok(depositors)
    }
}
```

API

- Extend the JSON RPC with the devcon_depositors method that accepts the count argument
- Query the storage for the specified numbers of depositors sorted by the sum of all deposit amounts

```rust
let handle = builder
    .node(EthereumNode::default())
    .install_exex("Devcon", move |ctx| async { Ok(exex(ctx, exex_connection)) })
    .extend_rpc_modules(move |ctx| {
        ctx.modules.merge_configured(DevconRpcExt { connection }.into_rpc())?;
        Ok(())
    })
    .launch()
    .await?;

handle.wait_for_node_exit().await
```

Node Builder

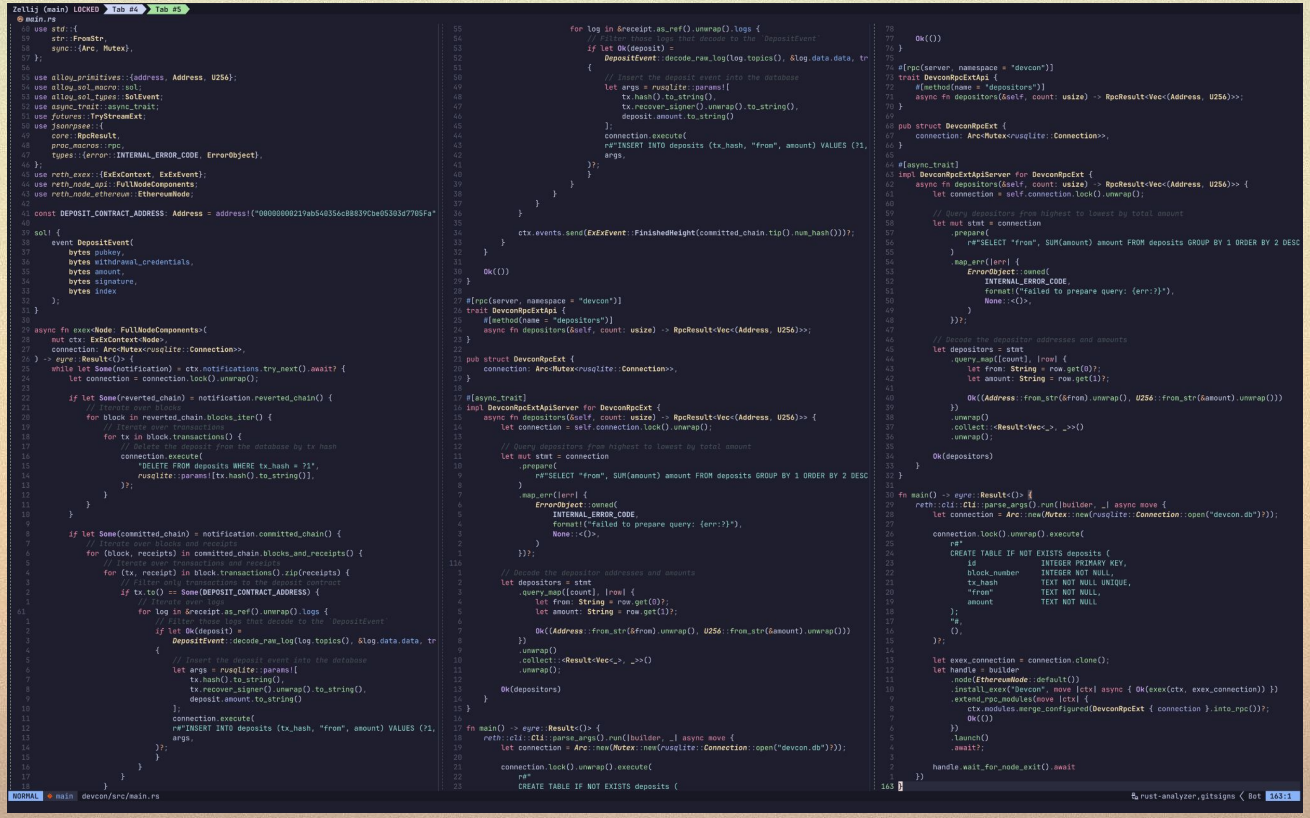
- Bundle the ExEx and the JSON RPC method into one Reth binary
  - Install ExEx
  - Extend RPC with our module
- Start the node

```
→  cast rpc -r localhost:8544 devcon_depositors 10 | jq
[
  [
    "0xdd5dfb73a16b21a6d6baff278fe05d97f71acfd3",
    320000000000
  ],
  [
    "0xc9b7a66cd66ffd62838ecb2dc49c6b67ebbb4f1b",
    32000000000
  ],
  [
    "0xb37be96e305e569f7c8a103476f3e966a8290aba",
    32000000000
  ],
  [
    "0x8ac112a5540f441cc9bebcc647041a6e0d595b94",
    32000000000
  ],
  [
    "0x22528484e68ee96c4b392fbf5a4b8fde92b5ddaf",
    32000000000
  ],
  [
    "0x143bf3a77ad31ee2502b79fa1595222722d4c0c9",
    32000000000
  ]
]
```

Testing

- Start the node and wait for it to sync if you didn’t have a synced one already
- Query the depositors via cast

The result: 1 file with <200 LOC



What else can you build with ExExes?

- Remote ExEx
  - Streams the data out via a gRPC server
- Oracle ExEx
  - Listen to an off-chain data source
  - Sign attestations to this data
  - Gossip attestations
  - Submit the result of consensus on-chain when the quorum is reached (think AVSs)
- Based rollup
  - Expose your own RPC to send transactions
  - Execute and batch them together
  - Submit on-chain

- We want your ideas!
- Scan the QR code and checkout the examples repo

What else can you build with ExExes?

- Remote ExEx
  - Streams the data out via a gRPC server
- Oracle ExEx
  - Listen to an off-chain data source
  - Sign attestations to this data
  - Gossip attestations
  - Submit the result of consensus on-chain when the quorum is reached (think AVSs)
- Based rollup
  - Expose your own RPC to send transactions
  - Execute and batch them together
  - Submit on-chain

- We want your ideas!
- Scan the QR code and checkout the examples repo

Thank you. Let's build.

Alexey Shekhirin
Founding Engineer, Ithaca
@ashekhirin