SYNCING FROM SCRATCH
1) run a fetcher on a Kusama node, to get all remarks
2) run a consolidator on the fetched output, to get the consolidated set of remarks, and thus the "NFT database" in simple JSON file format
3) run any operations on this final set, like analytics, burn checks, owner checks, and more.

ATTACCARSI A POLKADOT

Download di una release dot compilata da https://github.com/paritytech/polkadot/releases

runnarla da terminale con:

chmod +x polkadot && mv polkadot kusama

./kusama --pruning archive --db rocksdb

NB If you're running this somewhere in a server of yours and want to connect to it from outside, also add --rpc-cors all to allow other origins to connect to this server.

CI METTE QUALCHE GIORNO A SINCRONIZZARSI

Ora abbiamo un KSM node che runna, siamo nella blockchain

Ora tocca fetchare i remark con i rmrk-tools

Installarli: serve versioni moderne di NodeJS e Yarn...NVM e settare a 14.16 almeno

yarn install rmrk-tools (RMRK2 DEFAULT)

fetch cmd: yarn cli:fetch --prefixes=0x726d726b,0x524d524b –ws=ws://localhost:9933

Output: file json di remark, sarà il nostro DB

Se avevamo già fetchato

yarn cli:fetch --prefixes=0x726d726b,0x524d524b --append=QmdDywgAeybKG6erv5tJmzzfADZs19aJQV1PoMDmff6jR5.json --ws=wss://node.rmrk.app --from=8000000 --to=9000000

- `--ws URL` : websocket URL to connecto to, defaults to `127.0.0.1:9944`
- `--from FROM` : block from which to start, defaults to 0 (note that for RMRK, canonically the block 4892957 is genesis)
- `--to TO` : block until which to search, defaults to latest
- `--prefixes PREFIXES` : limit return data to only remarks with these prefixes. Can be comma separated list. Prefixes can be hex or utf8. Case sensitive. Example: 0x726d726b,0x524d524b
- `--append PATH` : special mode which takes the last block in an existing dump file + 1 as FROM (overrides FROM). Appends new blocks with remarks into that file. Convenient for running via cronjob for continuous remark list building. Performance right now is 1000 blocks per 10 seconds, so processing 5000 blocks with a `* * * * *` cronjob should be doable. Example: `yarn cli:fetch --prefixes=0x726d726b,0x524d524b --append=somefile.json`
- `--collection` : filter by specific collection or part of collection ID (i.e. RMRK substring)
- `--fin` : defaults to "yes" if omitted. When "yes", fetches up to last finalized block if `to` is omitted. Otherwise, last block. `no` is useful for testing.
- `--output` : name of the file into which to save the output. Overridden if `append` is used.

CONSOLIDATION

yarn cli:consolidate --json=dump.json --ws=ws://localhost:9933

Consolidare significa fare “decode”

ORA TOCCA GESTIRE I DATI CON LE **API**

API

Le informazioni (remark) non stanno sui blocchi ma stanno storati negli hdd dei nodi, la chain rimane immutata e sono letti come external input.

RMRK E’ LO STD PER INTERPRETARE E PROCESSARE LE INFORMAZIONI NEL BLOCCO CON GLI EXTERNAL INPUT

RMRK is a set of rules dictating how to interpret blockchain graffiti in a way that lets us simulate logic on a chain without smart contracts.

Per runnare un local node, dalla src foulder eseguire:

./target/release/polkadot –- dev –-tmp

API

Installare polkadot API con `yarn add @polkadot/api`

Basics + Metadata

Importante: le interfacce in polkadot spesos non sono statiche.

Esse vengono generate automaticamente al momento della connesisone ad un nodo

Metadata

Una delle prime cose che avviene al momento della connessione ad un nodo è la lettura dei metadata, in base ad esse vengono implementate le api.

I metadati vengono rappresentati nella seguente forma: api.<type>.<module>.<section>

Dove type ricade in uno dei seguenti casi:

const: costanti runtime. **Non sono funzioni**, piuttosto accedono ad un endpoint e tornano subito il risultato. Ad es.: api.consts.balances.existentialDeposit

query: indaga lo stato della chain, estrapolando informazioni da esso

es.: api.query.system.account(<accountId>)

tx: extrinsics, ad es.: api.tx.balances.transfer(<accountId>, <value>)

Inoltre i metadata danno informazioni sugli eventi, interfaccia:
api.query.system.events()

Appaiono anche nelle transazioni...ne parliamo oltre

IMPORTANTE

Nessun endpoint di api.{consts, query, tx}.<module>.<method> è hardcodato nelle API.

Tutto viene implementato da ciò che dicono i metadati e quindi è tutto completamente dinamico. Ciò vuol dire che le API cambiano in base all chain a cui siamo connessi.

TIPI

I emtadati definiscono le chiamate a funzione, le loro signature, e dunque anche tutti i tipi definiti nelle varie interfacce. Questo significa che i tipi delle API e del nodo interrogato devono essere “allineati” (ma in futuro potrebbe non essere più necessario)

ad es.:

Substrate definisce BlockNumber come un u32, e le API sono concordi a ciò, ma se una chain ne implementa una versione differente le Api dovranno saperlo così da adattarsi

CHAIN DEFAULTS

oltre a api.[consts | query | tx], le api definiscono alcune informazioni ottenibili direttamente dall'interfaccia:

`api.genesisHash` - The genesisHash of the connected chain

`api.runtimeMetadata` - The metadata as retrieved from the chain

`api.runtimeVersion` - The chain runtime version (including spec/impl. versions and types)

`api.libraryInfo` - The version of the API, i.e. `@polkadot/api v0.90.1`

CREATE API INSTANCE

```
// Import
import { ApiPromise, WsProvider } from '@polkadot/api';

...
// Construct
const wsProvider = new WsProvider('wss://rpc.polkadot.io');
const api = await ApiPromise.create({ provider: wsProvider });

// Do something
console.log(api.genesisHash.toHex());
```

Nella costruzione osserviamo la creazione di un websocket provider.

Se non specificato, di default è "ws://127.0.0.1:9944"

WebSocketProvider unico provider totalmente supportato dalle API per ora.

"Polkadot/Substrate really comes alive with possibilities once you have access to bi-directional RPCs such as what WebSockets provide. (It is technically possible to have some limited capability via bare-HTTP, but at this point WebSockets is the only fully-operational and supported version - always remember that it is just "upgraded HTTP".)"

Notare l'uso di **await**, che ci riporta ad **async….**chiamate bloccanti

senza queto tipo di sintassi, sarebbe

```
ApiPromise
  .create({ provider: wsProvider }).isReady
  .then((api) =>
    console.log(api.genesisHash.toHex())
  );
```

Per la creazione dell’istanza si può anche non sare lo shortcut .create, ma usare una new

La soluzione dunque sarebbe così:

```
// Create the instance
const api = new ApiPromise({ provider: wsProvider });

// Wait until we are ready and connected
await api.isReady;

// Do something
console.log(api.genesisHash.toHex());
```