```
# src/agentics/mcp/snapshot_api.py
from __future__ import annotations

"""
Snapshot GraphQL -> FastMCP tools (stdio server)

This module exposes Snapshot read-only operations as MCP tools:
- list_proposals
- list_finished_proposals
- get_proposal_by_id
- get_proposal_result_by_id
- get_votes_page
- get_votes_all
- resolve_proposal_id_from_url
- health

Run as a stdio MCP server:
export PYTHONPATH=src
python src/agentics/mcp/snapshot_api.py
"""

import os
import re
import time
import random
from typing import Any, Dict, List, Optional

import requests
from mcp.server.fastmcp import FastMCP
from pydantic import BaseModel, Field

# -----------------------------
# Config
# -----------------------------
SNAPSHOT_API = os.getenv("SNAPSHOT_API", "https://hub.snapshot.org/graphql")
TIMEOUT = int(os.getenv("SNAPSHOT_TIMEOUT", "30"))
BASE_SLEEP = float(os.getenv("SNAPSHOT_BASE_SLEEP", "0.6"))
MAX_RETRIES = int(os.getenv("SNAPSHOT_MAX_RETRIES", "5"))
BACKOFF_BASE = float(os.getenv("SNAPSHOT_BACKOFF_BASE", "1.7"))
JITTER_MIN = float(os.getenv("SNAPSHOT_JITTER_MIN", "0.10"))
JITTER_MAX = float(os.getenv("SNAPSHOT_JITTER_MAX", "0.35"))

# Shared HTTP session with a stable UA to avoid being rate-limited too aggressively.
_session = requests.Session()
_session.headers.update({"User-Agent": "mcp-snapshot-api/1.0"})

def _sleep() -> None:
"""Polite sleep with jitter between requests to avoid rate limiting."""
```

```
time.sleep(BASE_SLEEP + random.uniform(JITTER_MIN, JITTER_MAX))

# -----------------------------
# GraphQL helpers
# -----------------------------
def gql(query: str, variables: Optional[dict] = None) -> dict:
"""Call Snapshot GraphQL endpoint with retry/backoff."""
retries = 0
while True:
_sleep()
try:
r = _session.post(
SNAPSHOT_API,
json={"query": query, "variables": variables or {}},
timeout=TIMEOUT,
)
except requests.RequestException:
if retries < MAX_RETRIES:
time.sleep((BACKOFF_BASE ** retries) + random.uniform(JITTER_MIN, JITTER_MAX))
retries += 1
continue
raise
if r.status_code == 200:
return r.json()
if r.status_code in (429, 502, 503, 504) and retries < MAX_RETRIES:
ra = r.headers.get("Retry-After")
delay = float(ra) if (ra and ra.isdigit()) else (BACKOFF_BASE ** retries)
time.sleep(delay + random.uniform(JITTER_MIN, JITTER_MAX))
retries += 1
continue
r.raise_for_status()

# -----------------------------
# GraphQL queries
# -----------------------------
PROPOSALS_Q = """
query($space: String!, $first: Int!, $skip: Int!) {
proposals(
first: $first
skip: $skip
where: { space_in: [$space] }
orderBy: "created"
orderDirection: desc
) {
id
title
author
body
```

```
discussion
start
end
state
}
}
"""

PROPOSAL_BY_ID_Q = """
query($id: String!) {
proposal(id: $id) {
id
title
body
author
choices
start
end
discussion
state
}
}
"""

PROPOSAL_RESULT_Q = """
query($id: String!) {
proposal(id: $id) {
id
choices
scores
scores_total
state
}
}
"""

VOTES_BY_PROPOSAL_Q = """
query($proposal: String!, $first: Int!, $skip: Int!) {
votes(
first: $first
skip: $skip
where: { proposal: $proposal }
orderBy: "created"
orderDirection: asc
) {
id
voter
created
```

```
choice
vp
reason
}
}
"""

# -----------------------------
# Pydantic I/O models (optional but helpful for schema clarity)
# -----------------------------
class ProposalsIn(BaseModel):
space: str = Field(..., description="Snapshot space, e.g., 'aavedao.eth'")
limit: int = Field(200, ge=1, le=1000, description="Max proposals to return (client-side trim)")

class VotesPageIn(BaseModel):
proposal_id: str = Field(..., description="Snapshot proposal id")
first: int = Field(500, ge=1, le=1000, description="Page size")
skip: int = Field(0, ge=0, description="Offset for pagination")

# -----------------------------
# FastMCP app
# -----------------------------
mcp = FastMCP("SnapshotAPI")

# -----------------------------
# Core helpers (reused by tools)
# -----------------------------
def _fetch_all_proposals(space: str, batch: int = 100) -> List[dict]:
"""Fetch all proposals for a space using paged GraphQL queries."""
out: List[dict] = []
skip = 0
while True:
data = gql(PROPOSALS_Q, {"space": space, "first": batch, "skip": skip})
chunk = (data.get("data") or {}).get("proposals") or []
if not chunk:
break
out.extend(chunk)
if len(chunk) < batch:
break
skip += batch
return out

def _finished_only(proposals: List[dict]) -> List[dict]:
"""Filter proposals to those in 'closed' state whose end <= now."""
import datetime
from datetime import timezone
now_ts = int(datetime.datetime.now(timezone.utc).timestamp())
return [p for p in proposals if p.get("state") == "closed" and int(p.get("end") or 0) <= now_ts]
```

```
def _fetch_proposal_by_id(pid: str) -> dict:
data = gql(PROPOSAL_BY_ID_Q, {"id": pid})
return (data.get("data") or {}).get("proposal") or {}

def _fetch_proposal_result_by_id(pid: str) -> dict:
data = gql(PROPOSAL_RESULT_Q, {"id": pid})
return (data.get("data") or {}).get("proposal") or {}

def _fetch_votes_page(pid: str, first: int, skip: int) -> List[dict]:
data = gql(VOTES_BY_PROPOSAL_Q, {"proposal": pid, "first": first, "skip": skip})
return ( (data.get("data") or {}).get("votes") ) or []

def _fetch_votes_all(pid: str, batch: int = 500) -> List[dict]:
"""Fetch all votes for a proposal with paging (ascending by created)."""
out: List[dict] = []
skip = 0
while True:
chunk = _fetch_votes_page(pid, batch, skip)
if not chunk:
break
out.extend(chunk)
if len(chunk) < batch:
break
skip += batch
return out

# -----------------------------
# Tools
# -----------------------------
@mcp.tool()
def list_proposals(args: ProposalsIn) -> List[dict]:
"""
List proposals for a given Snapshot space (most recent first).
This returns up to 'limit' proposals (client-side truncated).
"""
all_props = _fetch_all_proposals(args.space)
return all_props[: max(1, min(args.limit, 1000))]

@mcp.tool()
def list_finished_proposals(args: ProposalsIn) -> List[dict]:
"""
List finished proposals (state='closed' and end <= now) for a space.
This returns up to 'limit' proposals (client-side truncated).
"""
all_props = _fetch_all_proposals(args.space)
fins = _finished_only(all_props)
return fins[: max(1, min(args.limit, 1000))]
```

```
@mcp.tool()
def get_proposal_by_id(proposal_id: str) -> dict:
"""Get a single proposal metadata record by id."""
return _fetch_proposal_by_id(proposal_id)

@mcp.tool()
def get_proposal_result_by_id(proposal_id: str) -> dict:
"""Get result (choices/scores/scores_total/state) for a proposal id."""
return _fetch_proposal_result_by_id(proposal_id)

@mcp.tool()
def get_votes_page(proposal_id: str, first: int = 500, skip: int = 0) -> List[dict]:
"""Get one page of votes for a proposal (ascending by created)."""
# Defensive bounds
first = max(1, min(int(first or 500), 1000))
skip = max(0, int(skip or 0))
return _fetch_votes_page(proposal_id, first, skip)

@mcp.tool()
def get_votes_all(proposal_id: str, batch: int = 500) -> List[dict]:
"""Get all votes for a proposal using pagination."""
batch = max(1, min(int(batch or 500), 1000))
return _fetch_votes_all(proposal_id, batch=batch)

@mcp.tool()
def resolve_proposal_id_from_url(snapshot_url: str) -> Optional[str]:
"""
Resolve proposal id from a full Snapshot URL.
Example: https://snapshot.org/#/aavedao.eth/proposal/0xABC... -> 0xABC...
"""
m = re.search(r"/proposal/([0-9a-zA-Z]+)", snapshot_url)
return m.group(1) if m else None

@mcp.tool()
def health() -> Dict[str, Any]:
"""Simple health check tool."""
return {
"ok": True,
"service": "SnapshotAPI",
"api": SNAPSHOT_API,
"timeout": TIMEOUT,
"retries": MAX_RETRIES,
}

# -----------------------------
# MCP stdio launcher
# -----------------------------
```

```
if __name__ == "__main__":
# Run as a stdio MCP server so orchestrators/IDEs can attach as a tool.
mcp.run(transport="stdio")
```