{
"memo_type": "monolith_structure",
"source_file": "fortuna.py",
"part": 1,
"total_parts": 3,
"blocks": [
{
"type": "import",
"content": "from __future__ import annotations\n"
},
{
"type": "miscellaneous",
"content": "#
\u2501\u250
CRITICAL: Fix for Playwright + PyInstaller + Windows\n# Must be at the very top, before any other imports\n#
\u2501\u250
},
{
"type": "import",
"content": "import sys\n"
},
{
"type": "import",
"content": "import platform\n"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "unknown",
"content": "if platform.system() == 'Windows' and getattr(sys, 'frozen', False):\n # Running as frozen EXE on Windows - force
ProactorEventLoop\n import asyncio\n try:\n asyncio.set_event_loop_policy(asyncio.WindowsProactorEventLoopPolicy())\n except
AttributeError:\n # Fallback for older Python versions if needed, though 3.8+ is standard now\n pass\n"
},
{
"type": "miscellaneous",
"content": "#
\u2501\u250
fortuna_discovery_engine.py\n# Aggregated monolithic discovery adapters for Fortuna\n# This engine serves as a
high-reliability fallback for the Fortuna discovery system.\n\n"
},
{
"type": "docstring",
"content": "\"\"\"\nFortuna Discovery Engine - Production-grade racing data aggregation.\n\nThis module provides a unified
collection of adapters for fetching racecard data\nfrom various racing websites. It serves as a high-reliability fallback
system.\n\"\"\"\n"
},
{
"type": "import",
"content": "import argparse\n"
},
{
"type": "import",
"content": "import asyncio\n"
},
{
"type": "import",
"content": "import functools\n"
},
{
"type": "import",
"content": "from functools import lru_cache\n"
},
{
"type": "import",
"content": "import html\n"
},
{
"type": "import",
"content": "import json\n"
},
{
"type": "import",
"content": "import logging\n"
},
{
"type": "import",
"content": "import os\n"
},
{
"type": "import",
"content": "import random\n"
},
{

```
"type": "import",
"content": "import re\n"
},
{
"type": "import",
"content": "import time\n"
},
{
"type": "import",
"content": "from abc import ABC, abstractmethod\n"
},
{
"type": "import",
"content": "from collections import defaultdict\n"
},
{
"type": "import",
"content": "from dataclasses import dataclass, field\n"
},
{
"type": "import",
"content": "from datetime import date, datetime, timedelta, timezone\n"
},
{
"type": "import",
"content": "from decimal import Decimal\n"
},
{
"type": "import",
"content": "from enum import Enum\n"
},
{
"type": "import",
"content": "from io import StringIO\n"
},
{
"type": "import",
"content": "from pathlib import Path\n"
},
{
"type": "import",
"content": "from typing import (\n Any,\n Annotated,\n Callable,\n ClassVar,\n Dict,\n Final,\n List,\n Optional,\n Tuple,\n
Type,\n TypeVar,\n Union,\n)\n"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "import",
"content": "import httpx\n"
},
{
"type": "import",
"content": "import pandas as pd\n"
},
{
"type": "import",
"content": "import sqlite3\n"
},
{
"type": "import",
"content": "from zoneinfo import ZoneInfo\n"
},
{
"type": "import",
"content": "from concurrent.futures import ThreadPoolExecutor\n"
},
{
"type": "import",
"content": "from contextlib import asynccontextmanager\n"
},
{
"type": "import",
"content": "import structlog\n"
},
{
"type": "import",
"content": "import subprocess\n"
},
{
"type": "import",
"content": "import sys\n"
},
{
```

```
"type": "import",
"content": "import threading\n"
},
{
"type": "import",
"content": "import webbrowser\n"
},
{
"type": "import",
"content": "from pydantic import (\n BaseModel,\n ConfigDict,\n Field,\n WrapSerializer,\n field_validator,\n)\n"
},
{
"type": "import",
"content": "from selectolax.parser import HTMLParser, Node\n"
},
{
"type": "import",
"content": "from tenacity import (\n RetryError,\n retry,\n retry_if_exception_type,\n stop_after_attempt,\n
wait_exponential,\n)\n"
},
{
"type": "miscellaneous",
"content": "\n# --- OPTIONAL IMPORTS ---\n"
},
{
"type": "unknown",
"content": "try:\n from curl_cffi import requests as curl_requests\nexcept Exception:\n curl_requests = None\n"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "unknown",
"content": "try:\n import tomli\n HAS_TOML = True\nexcept ImportError:\n HAS_TOML = False\n"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "unknown",
"content": "try:\n from scrapling import AsyncFetcher, Fetcher\n from scrapling.parser import Selector\n
ASYNC_SESSIONS_AVAILABLE = True\nexcept Exception:\n ASYNC_SESSIONS_AVAILABLE = False\n Selector = None # type: ignore\n"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "unknown",
"content": "try:\n from scrapling.fetchers import AsyncDynamicSession, AsyncStealthySession\nexcept Exception:\n
ASYNC_SESSIONS_AVAILABLE = False\n"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "unknown",
"content": "try:\n from scrapling.core.custom_types import StealthMode\nexcept Exception:\n class StealthMode: # type:
ignore\n FAST = \"fast\"\n CAMOUFLAGE = \"camouflage\"\n"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "unknown",
"content": "try:\n import winsound\nexcept (ImportError, RuntimeError):\n winsound = None\n"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def get_resp_status(resp: Any) -> Union[int, str]:\n if hasattr(resp, \"status_code\"): return resp.status_code\n
return getattr(resp, \"status\", \"unknown\")\n",
"name": "get_resp_status"
},
{
"type": "miscellaneous",
"content": "\n"
},
```

```
{
"type": "function",
"content": "def is_frozen() -> bool:\n \"\"\"Check if running as a frozen executable (PyInstaller, cx_Freeze, etc.)\"\"\"\n
return getattr(sys, 'frozen', False) and hasattr(sys, '_MEIPASS')\n",
"name": "is_frozen"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "function",
"content": "def get_base_path() -> Path:\n \"\"\"Returns the base path of the application (frozen or source).\"\"\"\n if
is_frozen():\n return Path(sys._MEIPASS)\n return Path(__file__).parent\n",
"name": "get_base_path"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "function",
"content": "def load_config() -> Dict[str, Any]:\n \"\"\"Loads configuration from config.toml with intelligent
fallback.\"\"\"\n config = {\n \"analysis\": {\"trustworthy_ratio_min\": 0.7, \"max_field_size\": 11},\n \"region\":
{\"default\": \"GLOBAL\"},\n \"ui\": {\"auto_open_report\": True, \"show_status_card\": True},\n \"logging\": {\"level\":
\"INFO\", \"save_to_file\": True}\n }\n\n config_paths = [Path(\"config.toml\")]\n if is_frozen():\n config_paths.insert(0,
Path(sys.executable).parent / \"config.toml\")\n config_paths.append(Path(sys._MEIPASS) / \"config.toml\")\n\n selected_config
= None\n for cp in config_paths:\n if cp.exists():\n selected_config = cp\n break\n\n if selected_config and HAS_TOML:\n
try:\n with open(selected_config, \"rb\") as f:\n toml_data = tomli.load(f)\n # Deep merge simple dict\n for section, values
in toml_data.items():\n if section in config and isinstance(values, dict):\n config[section].update(values)\n else:\n
config[section] = values\n except Exception as e:\n print(f\"Warning: Failed to load config.toml: {e} - using default
configuration\")\n else:\n # Explicitly log if we are falling back to defaults due to missing config or parser\n if not
selected_config:\n structlog.get_logger().debug(\"No config.toml found, using default configuration\")\n elif not HAS_TOML:\n
structlog.get_logger().warning(\"tomli not installed, using default configuration\")\n\n return config\n",
"name": "load_config"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "function",
"content": "def print_status_card(config: Dict[str, Any]):\n \"\"\"Prints a friendly status card with application health and
latest metrics.\"\"\"\n if not config.get(\"ui\", {}).get(\"show_status_card\", True):\n return\n\n version = \"Unknown\"\n
version_file = get_base_path() / \"VERSION\"\n if version_file.exists():\n version = version_file.read_text().strip()\n\n
try:\n from rich.console import Console\n console = Console()\n print_func = console.print\n except ImportError:\n # Fallback
to structlog for telemetry (GPT5 Improvement)\n sl = structlog.get_logger()\n print_func = lambda msg: sl.info(msg)\n\n
print_func(\"\\n\" + \"\u2550\" * 60)\n print_func(f\" \ud83d\udc0e FORTUNA FAUCET INTELLIGENCE - v{version} \".center(60,
\"\u2550\"))\n print_func(\"\u2550\" * 60)\n\n # Region and active mode\n region = config.get(\"region\", {}).get(\"default\",
\"GLOBAL\")\n print_func(f\" \ud83d\udccd Region: [bold cyan]{region}[/] | \ud83d\udd0d Status: [bold green]READY[/]\")\n\n #
Database status\n db = FortunaDB()\n # We'll use a sync helper or just run it\n try:\n # Simple sqlite check\n conn =
sqlite3.connect(db.db_path)\n cursor = conn.cursor()\n cursor.execute(\"SELECT COUNT(*) FROM tips\")\n total_tips =
cursor.fetchone()[0]\n cursor.execute(\"SELECT COUNT(*) FROM tips WHERE audit_completed = 1\")\n audited =
cursor.fetchone()[0]\n cursor.execute(\"SELECT COUNT(*) FROM tips WHERE is_goldmine = 1\")\n goldmines =
cursor.fetchone()[0]\n conn.close()\n\n print_func(f\" \ud83d\udcca Database: {total_tips} tips | \u2705 {audited} audited |
\ud83d\udc8e {goldmines} goldmines\")\n except Exception:\n print_func(\" \ud83d\udcca Database: INITIALIZING\")\n\n # Odds
Hygiene\n trust_min = config.get(\"analysis\", {}).get(\"trustworthy_ratio_min\", 0.7)\n print_func(f\" \ud83d\udee1\ufe0f
Odds Hygiene: >{int(trust_min*100)}% trust ratio required\")\n\n # Reports\n reports = []\n if
Path(\"summary_grid.txt\").exists(): reports.append(\"Summary\")\n if Path(\"fortuna_report.html\").exists():
reports.append(\"HTML\")\n if reports:\n print_func(f\" \ud83d\udcc1 Latest Reports: {', '.join(reports)}\")\n\n
print_func(\"\u2550\" * 60 + \"\\n\")\n",
"name": "print_status_card"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "function",
"content": "def print_quick_help():\n \"\"\"Prints a friendly onboarding guide for new users.\"\"\"\n try:\n from rich.console
import Console\n from rich.panel import Panel\n console = Console()\n print_func = console.print\n except ImportError:\n #
Fallback to structlog for telemetry (GPT5 Improvement)\n sl = structlog.get_logger()\n print_func = lambda msg:
sl.info(msg)\n\n help_text = \"\"\"\n [bold yellow]Welcome to Fortuna Faucet Intelligence![/]\n\n This app helps you discover
\"Goldmine\" racing opportunities where the\n second favorite has strong odds and a significant gap from the favorite.\n\n
[bold]Common Commands:[/]\n \u2022 [cyan]Discovery:[/] Just run the app! It will fetch latest races and find goldmines.\n
\u2022 [cyan]Monitor:[/] Run with [green]--monitor[/] for a live-updating dashboard.\n \u2022 [cyan]Analytics:[/] Run
[green]fortuna_analytics.py[/] to see how past predictions performed.\n\n [bold]Useful Flags:[/]\n \u2022 [green]--status:[/]
See your database stats and application health.\n \u2022 [green]--show-log:[/] See highlights from recent fetching and
auditing.\n \u2022 [green]--region:[/] Force a region (USA, INT, or GLOBAL).\n\n [italic]Predictions are saved to
fortuna_report.html and summary_grid.txt[/]\n \"\"\"\n if 'Console' in globals() or 'console' in locals():\n
print_func(Panel(help_text, title=\"\ud83d\ude80 Quick Start Guide\", border_style=\"yellow\"))\n else:\n
print_func(help_text)\n",
"name": "print_quick_help"
},
{
```

```
"type": "miscellaneous",
"content": "\n"
},
{
"type": "async_function",
"content": "async def print_recent_logs():\n \"\"\"Prints recent fetch and audit highlights from the database.\"\"\"\n db =
FortunaDB()\n try:\n # We need to use sync connection here as it's called from main which is not in loop yet\n # Actually
main_all_in_one is async and called via asyncio.run\n conn = sqlite3.connect(db.db_path)\n conn.row_factory = sqlite3.Row\n\n
print(\"\\n\" + \"\u2500\" * 60)\n print(\" \ud83d\udd0d RECENT ACTIVITY LOG \".center(60, \"\u2500\"))\n print(\"\u2500\" *
60)\n\n # Recent Harvests\n cursor = conn.execute(\"SELECT timestamp, adapter_name, race_count, region FROM harvest_logs ORDER
BY id DESC LIMIT 5\")\n print(\"\\n [bold]Latest Fetches:[/]\")\n for row in cursor.fetchall():\n ts =
row['timestamp'][:16].replace('T', ' ')\n print(f\" \u2022 {ts} | {row['adapter_name']:<20} | {row['race_count']} races
({row['region']})\")\n\n # Recent Audits\n cursor = conn.execute(\"SELECT audit_timestamp, venue, race_number, verdict,
net_profit FROM tips WHERE audit_completed = 1 ORDER BY audit_timestamp DESC LIMIT 5\")\n rows = cursor.fetchall()\n if
rows:\n print(\"\\n [bold]Latest Audits:[/]\")\n for row in rows:\n ts = row['audit_timestamp'][:16].replace('T', ' ')\n emoji
= \"\u2705\" if row['verdict'] == \"CASHED\" else \"\u274c\"\n print(f\" \u2022 {ts} | {row['venue']:<15}
R{row['race_number']} | {emoji} {row['verdict']} (${row['net_profit']:+.2f})\")\n\n conn.close()\n print(\"\\n\" + \"\u2500\"
* 60 + \"\\n\")\n except Exception as e:\n print(f\"Error reading activity log: {e}\")\n",
"name": "print_recent_logs"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "function",
"content": "def open_report_in_browser():\n \"\"\"Opens the HTML report in the default system browser.\"\"\"\n html_path =
Path(\"fortuna_report.html\")\n if html_path.exists():\n print(f\"Opening {html_path} in your browser...\")\n try:\n abs_path
= html_path.absolute()\n if sys.platform == \"win32\":\n os.startfile(abs_path)\n else:\n import webbrowser\n
webbrowser.open(f\"file://{abs_path}\")\n except Exception as e:\n print(f\"Failed to open report: {e}\")\n else:\n print(\"No
report found. Run discovery first!\")\n",
"name": "open_report_in_browser"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "unknown",
"content": "try:\n from notifications import DesktopNotifier\n HAS_NOTIFICATIONS = True\nexcept Exception:\n HAS_NOTIFICATIONS
= False\n"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "unknown",
"content": "try:\n from browserforge.headers import HeaderGenerator\n from browserforge.fingerprints import
FingerprintGenerator\n # Smoke test: HeaderGenerator often fails if data files are missing (frozen app issue)\n _hg =
HeaderGenerator()\n BROWSERFORGE_AVAILABLE = True\nexcept Exception:\n BROWSERFORGE_AVAILABLE = False\n"
},
{
"type": "miscellaneous",
"content": "\n\n# --- TYPE VARIABLES ---\n"
},
{
"type": "assignment",
"content": "T = TypeVar(\"T\")\n",
"name": "T"
},
{
"type": "assignment",
"content": "RaceT = TypeVar(\"RaceT\", bound=\"Race\")\n"
},
{
"type": "miscellaneous",
"content": "\n# --- CONSTANTS ---\n"
},
{
"type": "assignment",
"content": "EASTERN = ZoneInfo(\"America/New_York\")\n",
"name": "EASTERN"
},
{
"type": "unknown",
"content": "DEFAULT_REGION: Final[str] = \"GLOBAL\"\n"
},
{
"type": "miscellaneous",
"content": "\n# Region-based adapter lists (Refined by Council of Superbrains Directive)\n# Single-continent adapters remain
in USA/INT jobs.\n# Multi-continental adapters move to the GLOBAL parallel fetch job.\n# AtTheRaces is duplicated into USA as
per explicit request.\n"
},
{
```

```
"type": "unknown",
"content": "USA_DISCOVERY_ADAPTERS: Final[set] = {\"Equibase\", \"TwinSpires\", \"RacingPostB2B\", \"StandardbredCanada\",
\"AtTheRaces\"}\n"
},
{
"type": "unknown",
"content": "INT_DISCOVERY_ADAPTERS: Final[set] = {\"TAB\", \"BetfairDataScientist\"}\n"
},
{
"type": "unknown",
"content": "GLOBAL_DISCOVERY_ADAPTERS: Final[set] = {\n \"SkyRacingWorld\", \"AtTheRaces\", \"AtTheRacesGreyhound\",
\"RacingPost\",\n \"Oddschecker\", \"Timeform\", \"BoyleSports\", \"SportingLife\", \"SkySports\",\n \"RacingAndSports\"\n}\n"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "unknown",
"content": "USA_RESULTS_ADAPTERS: Final[set] = {\"EquibaseResults\", \"SportingLifeResults\",
\"StandardbredCanadaResults\"}\n"
},
{
"type": "unknown",
"content": "INT_RESULTS_ADAPTERS: Final[set] = {\n \"RacingPostResults\", \"RacingPostTote\", \"AtTheRacesResults\",\n
\"SportingLifeResults\", \"SkySportsResults\", \"RacingAndSportsResults\",\n \"TimeformResults\"\n}\n"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "unknown",
"content": "MAX_VALID_ODDS: Final[float] = 1000.0\n"
},
{
"type": "unknown",
"content": "MIN_VALID_ODDS: Final[float] = 1.01\n"
},
{
"type": "unknown",
"content": "DEFAULT_ODDS_FALLBACK: Final[float] = 2.75\n"
},
{
"type": "unknown",
"content": "COMMON_PLACEHOLDERS: Final[set] = {2.75}\n"
},
{
"type": "unknown",
"content": "DEFAULT_CONCURRENT_REQUESTS: Final[int] = 5\n"
},
{
"type": "unknown",
"content": "DEFAULT_REQUEST_TIMEOUT: Final[int] = 30\n"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "unknown",
"content": "DEFAULT_BROWSER_HEADERS: Final[Dict[str, str]] = {\n \"Accept\":
\"text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8\",\n \"Accept-Language\":
\"en-US,en;q=0.9\",\n \"Cache-Control\": \"no-cache\",\n \"Connection\": \"keep-alive\",\n \"Pragma\": \"no-cache\",\n
\"Sec-Fetch-Dest\": \"document\",\n \"Sec-Fetch-Mode\": \"navigate\",\n \"Sec-Fetch-Site\": \"none\",\n \"Sec-Fetch-User\":
\"?1\",\n \"Upgrade-Insecure-Requests\": \"1\",\n}\n"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "unknown",
"content": "CHROME_USER_AGENT: Final[str] = (\n \"Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 \"\n \"(KHTML,
like Gecko) Chrome/125.0.0.0 Safari/537.36\"\n)\n"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "unknown",
"content": "CHROME_SEC_CH_UA: Final[str] = (\n '\"Google Chrome\";v=\"125\", \"Chromium\";v=\"125\",
\"Not.A/Brand\";v=\"24\"'\n)\n"
},
```

```
{
"type": "miscellaneous",
"content": "\n# Bet type keywords mapping (lowercase key -> display name)\n"
},
{
"type": "unknown",
"content": "BET_TYPE_KEYWORDS: Final[Dict[str, str]] = {\n \"superfecta\": \"Superfecta\",\n \"spr\": \"Superfecta\",\n
\"trifecta\": \"Trifecta\",\n \"tri\": \"Trifecta\",\n \"exacta\": \"Exacta\",\n \"ex\": \"Exacta\",\n \"quinella\":
\"Quinella\",\n \"qn\": \"Quinella\",\n \"daily double\": \"Daily Double\",\n \"dbl\": \"Daily Double\",\n \"pick 3\": \"Pick
3\",\n \"pick 4\": \"Pick 4\",\n \"pick 5\": \"Pick 5\",\n \"pick 6\": \"Pick 6\",\n \"first 4\": \"Superfecta\",\n
\"forecast\": \"Exacta\",\n \"tricast\": \"Trifecta\",\n}\n"
},
{
"type": "miscellaneous",
"content": "\n# Discipline detection keywords\n"
},
{
"type": "unknown",
"content": "DISCIPLINE_KEYWORDS: Final[Dict[str, List[str]]] = {\n \"Harness\": [\"harness\", \"trotter\", \"pacer\",
\"standardbred\", \"trot\", \"pace\"],\n \"Greyhound\": [\"greyhound\", \"dog\", \"dogs\"],\n \"Quarter Horse\": [\"quarter
horse\", \"quarterhorse\"],\n}\n"
},
{
"type": "miscellaneous",
"content": "\n\n# --- EXCEPTIONS ---\n"
},
{
"type": "class",
"content": "class FortunaException(Exception):\n \"\"\"Base exception for all Fortuna-related errors.\"\"\"\n pass\n",
"name": "FortunaException"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class ErrorCategory(Enum):\n \"\"\"Categories for classifying adapter errors.\"\"\"\n BOT_DETECTION =
\"bot_detection\"\n NETWORK = \"network\"\n STRUCTURE_CHANGE = \"structure_change\"\n TIMEOUT = \"timeout\"\n AUTHENTICATION =
\"authentication\"\n CONFIGURATION = \"configuration\"\n PARSING = \"parsing\"\n UNKNOWN = \"unknown\"\n",
"name": "ErrorCategory"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class AdapterError(FortunaException):\n \"\"\"Base error for adapter-specific issues.\"\"\"\n def __init__(self,
adapter_name: str, message: str, category: ErrorCategory = ErrorCategory.UNKNOWN):\n self.adapter_name = adapter_name\n
self.category = category\n super().__init__(f\"[{adapter_name}] {message}\")\n",
"name": "AdapterError"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class AdapterRequestError(AdapterError):\n def __init__(self, adapter_name: str, message: str):\n
super().__init__(adapter_name, message, ErrorCategory.NETWORK)\n",
"name": "AdapterRequestError"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class AdapterHttpError(AdapterRequestError):\n def __init__(self, adapter_name: str, status_code: int, url:
str):\n self.status_code = status_code\n self.url = url\n super().__init__(adapter_name, f\"Received HTTP {status_code} from
{url}\")\n",
"name": "AdapterHttpError"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class AdapterParsingError(AdapterError):\n def __init__(self, adapter_name: str, message: str):\n
super().__init__(adapter_name, message, ErrorCategory.PARSING)\n",
"name": "AdapterParsingError"
},
{
```

```
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class FetchError(Exception):\n def __init__(self, message: str, response: Optional[Any] = None, category:
ErrorCategory = ErrorCategory.UNKNOWN):\n super().__init__(message)\n self.response = response\n self.category = category\n",
"name": "FetchError"
},
{
"type": "miscellaneous",
"content": "\n\n# --- MODELS ---\n"
},
{
"type": "function",
"content": "def decimal_serializer(value: Any, handler: Callable[[Any], Any]) -> Any:\n if value is None: return None\n try:\n
return float(value)\n except (TypeError, ValueError):\n return handler(value)\n",
"name": "decimal_serializer"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "assignment",
"content": "JsonDecimal = Annotated[Any, WrapSerializer(decimal_serializer, when_used=\"json\")]\n"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class FortunaBaseModel(BaseModel):\n model_config = ConfigDict(\n populate_by_name=True,\n
arbitrary_types_allowed=True,\n str_strip_whitespace=True,\n )\n",
"name": "FortunaBaseModel"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class OddsData(FortunaBaseModel):\n win: Optional[JsonDecimal] = None\n place: Optional[JsonDecimal] = None\n
source: str\n last_updated: datetime = Field(default_factory=lambda: datetime.now(EASTERN))\n\n
@field_validator(\"last_updated\", mode=\"after\")\n @classmethod\n def validate_eastern(cls, v: datetime) -> datetime:\n
return ensure_eastern(v)\n",
"name": "OddsData"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class Runner(FortunaBaseModel):\n id: Optional[str] = None\n name: str\n number: Optional[int] = Field(None,
alias=\"saddleClothNumber\")\n scratched: bool = False\n odds: Dict[str, OddsData] = Field(default_factory=dict)\n win_odds:
Optional[float] = Field(None, alias=\"winOdds\")\n metadata: Dict[str, Any] = Field(default_factory=dict)\n\n
@field_validator(\"name\", mode=\"before\")\n @classmethod\n def clean_name(cls, v: Any) -> str:\n if not v:\n return
\"Unknown\"\n name = str(v).strip()\n # Handle non-breaking spaces\n name = name.replace('\\xa0', ' ')\n # Remove country
suffixes in parentheses, e.g., \"Jay Bee (IRE)\" -> \"Jay Bee\"\n name = re.sub(r\"\\s*\\([^)]*\\)\\s*$\", \"\", name)\n #
Remove leading numbers followed by a dot and space, e.g., \"1. Horse\" -> \"Horse\"\n name = re.sub(r\"^\\d+\\.\\s*\", \"\",
name)\n # Remove unwanted punctuation/marks that might break parsing or Excel\n # Keep letters, numbers, spaces, hyphens, and
apostrophes.\n name = re.sub(r\"[^a-zA-Z0-9\\s\\-\\'\\\\\\\"]\", \"\", name)\n # Collapse multiple spaces\n name =
re.sub(r\"\\s+\", \" \", name)\n return name.strip() or \"Unknown\"\n",
"name": "Runner"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class Race(FortunaBaseModel):\n id: str\n venue: str\n race_number: int = Field(..., alias=\"raceNumber\", ge=1,
le=100)\n start_time: datetime = Field(..., alias=\"startTime\")\n runners: List[Runner] = Field(default_factory=list)\n\n
@field_validator(\"start_time\", mode=\"after\")\n @classmethod\n def validate_eastern(cls, v: datetime) -> datetime:\n
\"\"\"Ensures all race start times are in US Eastern Time.\"\"\"\n return ensure_eastern(v)\n source: str\n discipline:
Optional[str] = None\n distance: Optional[str] = None\n field_size: Optional[int] = None\n available_bets: List[str] =
Field(default_factory=list, alias=\"availableBets\")\n metadata: Dict[str, Any] = Field(default_factory=dict)\n
qualification_score: Optional[float] = None\n is_error_placeholder: bool = False\n top_five_numbers: Optional[str] = None\n
error_message: Optional[str] = None\n",
"name": "Race"
},
{
"type": "miscellaneous",
```

```
"content": "\n# --- UTILITIES ---\n"
},
{
"type": "function",
"content": "def get_field(obj: Any, field_name: str, default: Any = None) -> Any:\n \"\"\"Helper to get a field from either an
object or a dictionary.\"\"\"\n if isinstance(obj, dict):\n return obj.get(field_name, default)\n return getattr(obj,
field_name, default)\n",
"name": "get_field"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def clean_text(text: Any) -> str:\n \"\"\"Strips leading/trailing whitespace and collapses internal
whitespace.\"\"\"\n if not text:\n return \"\"\n return \" \".join(str(text).strip().split())\n",
"name": "clean_text"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def node_text(n: Any) -> str:\n \"\"\"Consistently extracts text from Scrapling Selectors and Selectolax
Nodes.\"\"\"\n if n is None:\n return \"\"\n # Selectolax nodes have a .text() method, Scrapling Selectors have a .text
property\n txt = getattr(n, \"text\", None)\n if txt is None:\n return \"\"\n return txt().strip() if callable(txt) else
str(txt).strip()\n",
"name": "node_text"
},
{
"type": "miscellaneous",
"content": "\n\n@lru_cache(maxsize=1024)\n"
},
{
"type": "function",
"content": "def get_canonical_venue(name: Optional[str]) -> str:\n \"\"\"Returns a sanitized canonical form for deduplication
keys.\"\"\"\n if not name:\n return \"unknown\"\n # Call normalization first to strip race titles and ads\n norm =
normalize_venue_name(name)\n # Remove everything in parentheses (extra safety)\n norm =
re.sub(r\"[\\(\\[\uff08].*?[\\)\\]\uff09]\", \"\", norm)\n # Remove special characters, lowercase, strip\n res =
re.sub(r\"[^a-z0-9]\", \"\", norm.lower())\n return res or \"unknown\"\n",
"name": "get_canonical_venue"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def now_eastern() -> datetime:\n \"\"\"Returns the current time in US Eastern Time.\"\"\"\n return
datetime.now(EASTERN)\n",
"name": "now_eastern"
},
{
"type": "miscellaneous",
"content": "\n\n\n\n"
},
{
"type": "function",
"content": "def to_eastern(dt: datetime) -> datetime:\n \"\"\"Converts a datetime object to US Eastern Time.\"\"\"\n if
dt.tzinfo is None:\n return dt.replace(tzinfo=EASTERN)\n return dt.astimezone(EASTERN)\n",
"name": "to_eastern"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def ensure_eastern(dt: datetime) -> datetime:\n \"\"\"Ensures datetime is timezone-aware and in Eastern time. More
strict than to_eastern.\"\"\"\n if dt.tzinfo is None:\n return dt.replace(tzinfo=EASTERN)\n if dt.tzinfo is not EASTERN:\n
try:\n return dt.astimezone(EASTERN)\n except Exception:\n # Fallback for rare cases where conversion fails (e.g. invalid
times during DST transitions)\n return dt.replace(tzinfo=EASTERN)\n return dt\n",
"name": "ensure_eastern"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "assignment",
"content": "RACING_KEYWORDS = [\n\"PRIX\", \"CHASE\", \"HURDLE\", \"HANDICAP\", \"STAKES\", \"CUP\", \"LISTED\",
\"GBB\",\n\"RACE\", \"MEETING\", \"NOVICE\", \"TRIAL\", \"PLATE\", \"TROPHY\", \"CHAMPIONSHIP\",\n\"JOCKEY\", \"TRAINER\",
\"BEST ODDS\", \"GUARANTEED\", \"PRO/AM\", \"AUCTION\",\n\"HUNT\", \"MARES\", \"FILLIES\", \"COLTS\", \"GELDINGS\",
```

```
\"JUVENILE\", \"SELLING\",\n\"CLAIMING\", \"OPTIONAL\", \"ALLOWANCE\", \"MAIDEN\", \"OPEN\", \"INVITATIONAL\",\n\"CLASS \",
\"GRADE \", \"GROUP \", \"DERBY\", \"OAKS\", \"GUINEAS\", \"ELIE DE\",\n\"FREDERIK\", \"CONNOLLY'S\", \"QUINNBET\", \"RED
MILLS\", \"IRISH EBF\", \"SKY BET\",\n\"CORAL\", \"BETFRED\", \"WILLIAM HILL\", \"UNIBET\", \"PADDY POWER\",
\"BETFAIR\",\n\"GET THE BEST\", \"CHELTENHAM TRIALS\", \"PORSCHE\", \"IMPORTED\", \"IMPORTE\", \"THE JOC\",\n\"PREMIO\",
\"GRANDE\", \"CLASSIC\", \"SPRINT\", \"DASH\", \"MILE\", \"STAYERS\",\n\"BOWL\", \"MEMORIAL\", \"PURSE\", \"CONDITION\",
\"NIGHT\", \"EVENING\", \"DAY\",\n\"4RACING\", \"WILGERBOSDRIFT\", \"YOUCANBETONUS\", \"FOR HOSPITALITY\", \"SA \", \"TAB
\",\n\"DE \", \"DU \", \"DES \", \"LA \", \"LE \", \"AU \", \"WELCOME\", \"BET \", \"WITH \", \"AND \",\n\"NEXT\", \"WWW\",
\"GAMBLE\", \"BETMGM\", \"TV\", \"ONLINE\", \"LUCKY\", \"RACEWAY\",\n\"SPEEDWAY\", \"DOWNS\", \"PARK\", \"HARNESS\", \"
STANDARDBRED\", \"FORM GUIDE\", \"FULL FIELDS\"\n]\n",
"name": "RACING_KEYWORDS"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "assignment",
"content": "VENUE_MAP = {\n\"ABU DHABI\": \"Abu Dhabi\",\n\"AQU\": \"Aqueduct\",\n\"AQUEDUCT\": \"Aqueduct\",\n\"ARGENTAN\":
\"Argentan\",\n\"ASCOT\": \"Ascot\",\n\"AYR\": \"Ayr\",\n\"BAHRAIN\": \"Bahrain\",\n\"BANGOR ON DEE\":
\"Bangor-on-Dee\",\n\"CATTERICK\": \"Catterick\",\n\"CATTERICK BRIDGE\": \"Catterick\",\n\"CT\": \"Charles Town\",\n\"CENTRAL
PARK\": \"Central Park\",\n\"CHELMSFORD\": \"Chelmsford\",\n\"CHELMSFORD CITY\": \"Chelmsford\",\n\"CURRAGH\":
\"Curragh\",\n\"DEAUVILLE\": \"Deauville\",\n\"DED\": \"Delta Downs\",\n\"DELTA DOWNS\": \"Delta Downs\",\n\"DONCASTER\":
\"Doncaster\",\n\"DOVER DOWNS\": \"Dover Downs\",\n\"DOWN ROYAL\": \"Down Royal\",\n\"DUNDALK\": \"Dundalk\",\n\"DUNSTALL
PARK\": \"Wolverhampton\",\n\"EPSOM\": \"Epsom\",\n\"EPSOM DOWNS\": \"Epsom\",\n\"FG\": \"Fair Grounds\",\n\"FAIR GROUNDS\":
\"Fair Grounds\",\n\"FONTWELL\": \"Fontwell Park\",\n\"FONTWELL PARK\": \"Fontwell Park\",\n\"GREAT YARMOUTH\": \"Great
Yarmouth\",\n\"GP\": \"Gulfstream Park\",\n\"GULFSTREAM\": \"Gulfstream Park\",\n\"GULFSTREAM PARK\": \"Gulfstream
Park\",\n\"HAYDOCK\": \"Haydock Park\",\n\"HAYDOCK PARK\": \"Haydock Park\",\n\"HOOSIER PARK\": \"Hoosier Park\",\n\"HOVE\":
\"Hove\",\n\"KEMPTON\": \"Kempton Park\",\n\"KEMPTON PARK\": \"Kempton Park\",\n\"LRL\": \"Laurel Park\",\n\"LAUREL PARK\":
\"Laurel Park\",\n\"LINGFIELD\": \"Lingfield Park\",\n\"LINGFIELD PARK\": \"Lingfield Park\",\n\"LOS ALAMITOS\": \"Los
Alamitos\",\n\"MARONAS\": \"Maronas\",\n\"MEADOWLANDS\": \"Meadowlands\",\n\"MEYDAN\": \"Meydan\",\n\"MIAMI VALLEY\": \"Miami
Valley\",\n\"MIAMI VALLEY RACEWAY\": \"Miami Valley\",\n\"MVR\": \"Mahoning Valley\",\n\"MOHAWK\": \"Mohawk\",\n\"MOHAWK
PARK\": \"Mohawk\",\n\"MUSSELBURGH\": \"Musselburgh\",\n\"NAAS\": \"Naas\",\n\"NEWCASTLE\": \"Newcastle\",\n\"NEWMARKET\":
\"Newmarket\",\n\"NORTHFIELD PARK\": \"Northfield Park\",\n\"OXFORD\": \"Oxford\",\n\"PAU\": \"Pau\",\n\"OP\": \"Oaklawn
Park\",\n\"PEN\": \"Penn National\",\n\"POCONO DOWNS\": \"Pocono Downs\",\n\"SAM HOUSTON\": \"Sam Houston\",\n\"SAM HOUSTON
RACE PARK\": \"Sam Houston\",\n\"SANDOWN\": \"Sandown Park\",\n\"SANDOWN PARK\": \"Sandown Park\",\n\"SA\": \"Santa
Anita\",\n\"SANTA ANITA\": \"Santa Anita\",\n\"SARATOGA\": \"Saratoga\",\n\"SARATOGA HARNESS\": \"Saratoga
Harness\",\n\"SCIOTO DOWNS\": \"Scioto Downs\",\n\"SHEFFIELD\": \"Sheffield\",\n\"STRATFORD\":
\"Stratford-on-Avon\",\n\"SUN\": \"Sunland Park\",\n\"SUNLAND PARK\": \"Sunland Park\",\n\"TAM\": \"Tampa Bay
Downs\",\n\"TAMPA BAY DOWNS\": \"Tampa Bay Downs\",\n\"THURLES\": \"Thurles\",\n\"TP\": \"Turfway Park\",\n\"TUP\": \"Turf
Paradise\",\n\"TURF PARADISE\": \"Turf Paradise\",\n\"TURFFONTEIN\": \"Turffontein\",\n\"UTTOXETER\":
\"Uttoxeter\",\n\"VINCENNES\": \"Vincennes\",\n\"WARWICK\": \"Warwick\",\n\"WETHERBY\": \"Wetherby\",\n\"WOLVERHAMPTON\":
\"Wolverhampton\",\n\"WO\": \"Woodbine\",\n\"WOODBINE\": \"Woodbine\",\n\"WOODBINE MOHAWK\": \"Mohawk\",\n\"WOODBINE MOHAWK
PARK\": \"Mohawk\",\n\"YARMOUTH\": \"Great Yarmouth\",\n\"YONKERS\": \"Yonkers\",\n\"YONKERS RACEWAY\": \"Yonkers\",\n}\n",
"name": "VENUE_MAP"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def normalize_venue_name(name: Optional[str]) -> str:\n \"\"\"\n Normalizes a racecourse name to a standard
format.\n Aggressively strips race names, sponsorships, and country noise.\n \"\"\"\n if not name:\n return \"Unknown\"\n\n #
1. Initial Cleaning: Replace dashes and strip all parenthetical info\n # Handle full-width parentheses and brackets often
found in international data\n name = str(name).replace(\"-\", \" \")\n name = re.sub(r\"[\\(\\[\uff08].*?[\\)\\]\uff09]\", \"
\", name)\n\n cleaned = clean_text(name)\n if not cleaned:\n return \"Unknown\"\n\n # 2. Aggressive Race/Meeting Name
Stripping\n # If these keywords are found, assume everything after is the race name.\n\n upper_name = cleaned.upper()\n
earliest_idx = len(cleaned)\n for kw in RACING_KEYWORDS:\n # Check for keyword with leading space\n idx = upper_name.find(\"
\" + kw)\n if idx != -1:\n earliest_idx = min(earliest_idx, idx)\n\n track_part = cleaned[:earliest_idx].strip()\n if not
track_part:\n track_part = cleaned\n\n # Handle repetition check (e.g., \"Bahrain Bahrain\" -> \"Bahrain\")\n words =
track_part.split()\n if len(words) > 1 and words[0].lower() == words[1].lower():\n track_part = words[0]\n\n upper_track =
track_part.upper()\n\n # 3. High-Confidence Mapping\n # Map raw/cleaned names to canonical display names.\n\n # Direct match\n
if upper_track in VENUE_MAP:\n return VENUE_MAP[upper_track]\n\n # Prefix match (sort by length desc to avoid partial matches
on shorter names)\n for known_track in sorted(VENUE_MAP.keys(), key=len, reverse=True):\n if
upper_name.startswith(known_track):\n return VENUE_MAP[known_track]\n\n return track_part.title()\n",
"name": "normalize_venue_name"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def parse_odds_to_decimal(odds_str: Any) -> Optional[float]:\n \"\"\"\n Parses various odds formats (fractional,
decimal) into a float decimal.\n Uses advanced heuristics to extract odds from noisy strings.\n \"\"\"\n if odds_str is None:
return None\n s = str(odds_str).strip().upper()\n\n # Remove common non-odds noise and currency symbols\n s =
re.sub(r\"[$\\s\\xa0]\", \"\", s)\n s = re.sub(r\"(ML|MTP|AM|PM|LINE|ODDS|PRICE)[:=]*\", \"\", s)\n\n if s in (\"EVN\",
\"EVEN\", \"EVS\", \"EVENS\"): return 2.0\n if any(kw in s for kw in (\"SCR\", \"SCRATCHED\", \"N/A\", \"NR\", \"VOID\")):
return None\n\n try:\n # 1. Fractional Format: \"7/4\", \"7-4\", \"7 TO 4\"\n groups =
re.search(r\"(\\d+)\\s*(?:[/\\-]|TO)\\s*(\\d+)\", s)\n if groups:\n num, den = int(groups.group(1)), int(groups.group(2))\n if
den > 0: return round((num / den) + 1.0, 2)\n\n # 2. Decimal Format: \"5.00\", \"10.5\"\n decimal_match =
re.search(r\"(\\d+\\.\\d+)\", s)\n if decimal_match:\n value = float(decimal_match.group(1))\n if MIN_VALID_ODDS <= value <
MAX_VALID_ODDS: return round(value, 2)\n\n # 3. Simple Integer as fractional odds (e.g., \"5\" often means \"5/1\")\n # Only
apply if it's a likely odds value (not saddle cloth 1-20)\n int_match = re.match(r\"^(\\d+)$\", s)\n if int_match:\n val =
int(int_match.group(1))\n # Heuristic: only treat as fractional odds if it's in a likely range (1-50)\n # to avoid
```

misinterpreting horse numbers or race numbers.\n if 1 <= val <= 50:\n return float(val + 1)\n\n except Exception: pass\n return None\n",
"name": "parse_odds_to_decimal"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def is_placeholder_odds(value: Optional[Union[float, Decimal]]) -> bool:\n \"\"\"Detects if odds value is a known placeholder or default.\"\"\"\n if value is None:\n return True\n try:\n val_float = round(float(value), 2)\n return val_float in COMMON_PLACEHOLDERS\n except (ValueError, TypeError):\n return True\n",
"name": "is_placeholder_odds"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def is_valid_odds(odds: Any) -> bool:\n if odds is None: return False\n try:\n odds_float = float(odds)\n if not (MIN_VALID_ODDS <= odds_float < MAX_VALID_ODDS):\n return False\n return not is_placeholder_odds(odds_float)\n except Exception: return False\n",
"name": "is_valid_odds"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def create_odds_data(source_name: str, win_odds: Any, place_odds: Any = None) -> Optional[OddsData]:\n if not is_valid_odds(win_odds):\n if win_odds is not None and is_placeholder_odds(win_odds):\n structlog.get_logger().warning(\"placeholder_odds_detected\", source=source_name, odds=win_odds)\n return None\n return OddsData(win=float(win_odds), place=float(place_odds) if is_valid_odds(place_odds) else None, source=source_name)\n",
"name": "create_odds_data"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def scrape_available_bets(html_content: str) -> List[str]:\n if not html_content: return []\n available_bets: List[str] = []\n html_lower = html_content.lower()\n for kw, bet_name in BET_TYPE_KEYWORDS.items():\n if re.search(rf\"\\b{re.escape(kw)}\\b\", html_lower) and bet_name not in available_bets:\n available_bets.append(bet_name)\n return available_bets\n",
"name": "scrape_available_bets"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "function",
"content": "def detect_discipline(html_content: str) -> str:\n if not html_content: return \"Thoroughbred\"\n html_lower = html_content.lower()\n for disc, keywords in DISCIPLINE_KEYWORDS.items():\n if any(kw in html_lower for kw in keywords): return disc\n return \"Thoroughbred\"\n",
"name": "detect_discipline"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class SmartOddsExtractor:\n \"\"\"\n Advanced heuristics for extracting odds from noisy HTML or text.\n Scans for various patterns and returns the first plausible odds found.\n \"\"\"\n @staticmethod\n def extract_from_text(text: str) -> Optional[float]:\n if not text: return None\n # Try to find common odds patterns in the text\n # 1. Decimal odds (e.g. 5.00, 10.5)\n decimals = re.findall(r\"(\\d+\\.\\d+)\", text)\n for d in decimals:\n val = float(d)\n if MIN_VALID_ODDS <= val < MAX_VALID_ODDS: return round(val, 2)\n\n # 2. Fractional odds (e.g. 7/4, 10-1)\n fractions = re.findall(r\"(\\d+)\\s*[/\\-]\\s*(\\d+)\", text)\n for num, den in fractions:\n n, d = int(num), int(den)\n if d > 0 and (n/d) > 0.1: return round((n / d) + 1.0, 2)\n\n return None\n\n @staticmethod\n def extract_from_node(node: Any) -> Optional[float]:\n \"\"\"Scans a selectolax node for odds using multiple strategies.\"\"\"\n # Strategy 1: Look at text content of the entire node\n if hasattr(node, 'text'):\n if val := SmartOddsExtractor.extract_from_text(node.text()):\n return val\n\n # Strategy 2: Look at attributes\n if hasattr(node, 'attributes'):\n for attr in [\"data-odds\", \"data-price\", \"data-bestprice\", \"title\"]:\n if val_str := node.attributes.get(attr):\n if val := parse_odds_to_decimal(val_str):\n return val\n\n return None\n",
"name": "SmartOddsExtractor"
},
{
"type": "miscellaneous",
"content": "\n\n"
},

```
{
"type": "function",
"content": "def generate_race_id(prefix: str, venue: str, start_time: datetime, race_number: int, discipline: Optional[str] =
None) -> str:\n venue_slug = re.sub(r\"[^a-z0-9]\", \"\", venue.lower())\n date_str = start_time.strftime(\"%Y%m%d\")\n
time_str = start_time.strftime(\"%H%M\")\n\n # Always include a discipline suffix for consistency and better matching\n dl =
(discipline or \"Thoroughbred\").lower()\n if \"harness\" in dl: disc_suffix = \"_h\"\n elif \"greyhound\" in dl: disc_suffix
= \"_g\"\n elif \"quarter\" in dl: disc_suffix = \"_q\"\n else: disc_suffix = \"_t\"\n\n return
f\"{prefix}_{venue_slug}_{date_str}_{time_str}_R{race_number}{disc_suffix}\"\n",
"name": "generate_race_id"
},
{
"type": "miscellaneous",
"content": "\n\n# --- VALIDATORS ---\n"
},
{
"type": "class",
"content": "class RaceValidator(BaseModel):\n venue: str = Field(..., min_length=1)\n race_number: int = Field(..., ge=1,
le=100)\n start_time: datetime\n runners: List[Runner] = Field(..., min_length=2)\n",
"name": "RaceValidator"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class DataValidationPipeline:\n @staticmethod\n def validate_raw_response(adapter_name: str, raw_data: Any) ->
tuple[bool, str]:\n if raw_data is None: return False, \"Null response\"\n return True, \"OK\"\n @staticmethod\n def
validate_parsed_races(races: List[Race], adapter_name: str = \"Unknown\") -> tuple[List[Race], List[str]]:\n valid_races:
List[Race] = []\n warnings: List[str] = []\n for i, race in enumerate(races):\n try:\n data = race.model_dump() if
hasattr(race, \"model_dump\") else race.dict()\n RaceValidator(**data)\n valid_races.append(race)\n except Exception as e:\n
err_msg = f\"[{adapter_name}] Race {i} ({getattr(race, 'venue', 'Unknown')} R{getattr(race, 'race_number', '?')}) validation
failed: {str(e)}\"\n warnings.append(err_msg)\n structlog.get_logger().error(\"race_validation_failed\", adapter=adapter_name,
error=str(e), race_index=i, venue=getattr(race, 'venue', 'Unknown'))\n continue\n return valid_races, warnings\n",
"name": "DataValidationPipeline"
},
{
"type": "miscellaneous",
"content": "\n\n# --- CORE INFRASTRUCTURE ---\n"
},
{
"type": "class",
"content": "class GlobalResourceManager:\n \"\"\"Manages shared resources like HTTP clients and semaphores.\"\"\"\n
_httpx_client: Optional[httpx.AsyncClient] = None\n _locks: ClassVar[dict[asyncio.AbstractEventLoop, asyncio.Lock]] = {}\n
_lock_initialized: ClassVar[threading.Lock] = threading.Lock()\n _global_semaphore: Optional[asyncio.Semaphore] = None\n\n
@classmethod\n async def _get_lock(cls) -> asyncio.Lock:\n loop = asyncio.get_running_loop()\n if loop not in cls._locks:\n
with cls._lock_initialized:\n if loop not in cls._locks:\n cls._locks[loop] = asyncio.Lock()\n return cls._locks[loop]\n\n
@classmethod\n async def get_httpx_client(cls, timeout: Optional[int] = None) -> httpx.AsyncClient:\n \"\"\"\n Returns a
shared httpx client.\n If timeout is provided and differs from current client, the client is recreated.\n \"\"\"\n lock =
await cls._get_lock()\n async with lock:\n if cls._httpx_client is not None:\n # Guard against None in timeout comparison
(GPT5 Fix)\n current_timeout = getattr(cls._httpx_client.timeout, \"read\", None)\n if timeout is not None and current_timeout
is not None and abs(current_timeout - timeout) > 0.001:\n try:\n await cls._httpx_client.aclose()\n except Exception:\n pass\n
cls._httpx_client = None\n\n if cls._httpx_client is None:\n use_timeout = timeout or DEFAULT_REQUEST_TIMEOUT\n
cls._httpx_client = httpx.AsyncClient(\n follow_redirects=True,\n timeout=httpx.Timeout(use_timeout),\n
headers={**DEFAULT_BROWSER_HEADERS, \"User-Agent\": CHROME_USER_AGENT},\n limits=httpx.Limits(max_connections=100,
max_keepalive_connections=20)\n )\n return cls._httpx_client\n\n @classmethod\n def get_global_semaphore(cls) ->
asyncio.Semaphore:\n if cls._global_semaphore is None:\n try:\n # Attempt to get running loop to ensure we are in async
context\n asyncio.get_running_loop()\n cls._global_semaphore = asyncio.Semaphore(DEFAULT_CONCURRENT_REQUESTS * 2)\n except
RuntimeError:\n # Fallback if called outside a loop\n cls._global_semaphore = asyncio.Semaphore(DEFAULT_CONCURRENT_REQUESTS *
2)\n return cls._global_semaphore\n return cls._global_semaphore\n\n @classmethod\n async def cleanup(cls):\n if
cls._httpx_client:\n try:\n await cls._httpx_client.aclose()\n except (AttributeError, RuntimeError):\n pass\n
cls._httpx_client = None\n",
"name": "GlobalResourceManager"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class BrowserEngine(Enum):\n CAMOUFOX = \"camoufox\"\n PLAYWRIGHT = \"playwright\"\n CURL_CFFI = \"curl_cffi\"\n
PLAYWRIGHT_LEGACY = \"playwright_legacy\"\n HTTPX = \"httpx\"\n",
"name": "BrowserEngine"
},
{
"type": "miscellaneous",
"content": "\n\n@dataclass\n"
},
{
"type": "class",
"content": "class UnifiedResponse:\n \"\"\"Unified response object to normalize data across different fetch engines.\"\"\"\n
text: str\n status: int\n status_code: int\n url: str\n headers: Dict[str, str] = field(default_factory=dict)\n\n def
json(self) -> Any:\n return json.loads(self.text)\n",
"name": "UnifiedResponse"
},
```

```
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class FetchStrategy(FortunaBaseModel):\n primary_engine: BrowserEngine = BrowserEngine.PLAYWRIGHT\n enable_js:
bool = True\n stealth_mode: str = \"fast\"\n block_resources: bool = True\n max_retries: int = Field(3, ge=0, le=10)\n
timeout: int = Field(DEFAULT_REQUEST_TIMEOUT, ge=1, le=300)\n page_load_strategy: str = \"domcontentloaded\"\n wait_until:
Optional[str] = None\n network_idle: bool = False\n wait_for_selector: Optional[str] = None\n",
"name": "FetchStrategy"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class SmartFetcher:\n BOT_DETECTION_KEYWORDS: ClassVar[List[str]] = [\"datadome\", \"perimeterx\", \"access
denied\", \"captcha\", \"cloudflare\", \"please verify\"]\n def __init__(self, strategy: Optional[FetchStrategy] = None):\n
self.strategy = strategy or FetchStrategy()\n self.logger = structlog.get_logger(self.__class__.__name__)\n
self._engine_health = {\n BrowserEngine.CAMOUFOX: 0.9,\n BrowserEngine.CURL_CFFI: 0.8,\n BrowserEngine.PLAYWRIGHT: 0.7,\n
BrowserEngine.PLAYWRIGHT_LEGACY: 0.6,\n BrowserEngine.HTTPX: 0.5\n }\n self.last_engine: str = \"unknown\"\n if
BROWSERFORGE_AVAILABLE:\n self.header_gen = HeaderGenerator()\n self.fingerprint_gen = FingerprintGenerator()\n else:\n
self.header_gen = None\n self.fingerprint_gen = None\n\n async def fetch(self, url: str, **kwargs: Any) -> Any:\n method =
kwargs.pop(\"method\", \"GET\").upper()\n kwargs.pop(\"url\", None)\n # Check if engines are available before sorting\n
available_engines = [e for e in self._engine_health.keys()]\n if not curl_requests and BrowserEngine.CURL_CFFI in
available_engines:\n available_engines.remove(BrowserEngine.CURL_CFFI)\n if not ASYNC_SESSIONS_AVAILABLE:\n for e in
[BrowserEngine.CAMOUFOX, BrowserEngine.PLAYWRIGHT]:\n if e in available_engines: available_engines.remove(e)\n\n if not
available_engines:\n self.logger.error(\"no_fetch_engines_available\", url=url)\n raise FetchError(\"No fetch engines
available (install curl_cffi or scrapling)\")\n\n strategy = kwargs.get(\"strategy\", self.strategy)\n engines =
sorted(available_engines, key=lambda e: self._engine_health[e], reverse=True)\n if strategy.primary_engine in engines:\n
engines.remove(strategy.primary_engine)\n engines.insert(0, strategy.primary_engine)\n self.logger.debug(\"Fetch engines
ordered\", url=url, engines=[e.value for e in engines], primary=strategy.primary_engine.value)\n last_error:
Optional[Exception] = None\n for engine in engines:\n try:\n response = await self._fetch_with_engine(engine, url,
method=method, **kwargs)\n self._engine_health[engine] = min(1.0, self._engine_health[engine] + 0.1)\n self.last_engine =
engine.value\n return response\n except Exception as e:\n self.logger.debug(f\"Engine {engine.value} failed\", error=str(e))\n
self._engine_health[engine] = max(0.0, self._engine_health[engine] - 0.2)\n last_error = e\n continue\n err_msg =
repr(last_error) if last_error else \"All fetch engines failed\"\n self.logger.error(\"all_engines_failed\", url=url,
error=err_msg)\n raise last_error or FetchError(\"All fetch engines failed\")\n\n \n async def _fetch_with_engine(self,
engine: BrowserEngine, url: str, method: str, **kwargs: Any) -> Any:\n # Generate browserforge headers if available\n if
BROWSERFORGE_AVAILABLE:\n try:\n # Generate headers and a corresponding user agent\n fingerprint =
self.fingerprint_gen.generate()\n bf_headers = self.header_gen.generate()\n # Ensure User-Agent is consistent between
fingerprint and headers\n ua = getattr(fingerprint.navigator, 'userAgent', getattr(fingerprint.navigator, 'user_agent',
CHROME_USER_AGENT))\n bf_headers['User-Agent'] = ua\n\n if \"headers\" in kwargs:\n # Merge - browserforge headers complement
provided ones\n for k, v in bf_headers.items():\n if k not in kwargs[\"headers\"]:\n kwargs[\"headers\"][k] = v\n else:\n
kwargs[\"headers\"] = bf_headers\n self.logger.debug(\"Applied browserforge headers\", engine=engine.value)\n except Exception
as e:\n self.logger.warning(\"Failed to generate browserforge headers\", error=str(e))\n\n # Define browser-specific arguments
to strip for non-browser engines\n BROWSER_SPECIFIC_KWARGS = [\n \"network_idle\", \"wait_selector\", \"wait_until\",
\"impersonate\",\n \"stealth\", \"block_resources\", \"wait_for_selector\", \"stealth_mode\",\n \"strategy\"\n ]\n\n strategy
= kwargs.get(\"strategy\", self.strategy)\n if engine == BrowserEngine.HTTPX:\n # Pass strategy timeout if present in kwargs
or use default\n timeout = kwargs.get(\"timeout\", strategy.timeout)\n client = await
GlobalResourceManager.get_httpx_client(timeout=timeout)\n\n # Remove timeout and browser-specific keys from kwargs\n
req_kwargs = {\n k: v for k, v in kwargs.items()\n if k != \"timeout\" and k not in BROWSER_SPECIFIC_KWARGS\n }\n resp = await
client.request(method, url, timeout=timeout, **req_kwargs)\n return UnifiedResponse(resp.text, resp.status_code,
resp.status_code, str(resp.url), resp.headers)\n \n if engine == BrowserEngine.CURL_CFFI:\n if not curl_requests:\n raise
ImportError(\"curl_cffi is not available\")\n \n self.logger.debug(f\"Using curl_cffi for {url}\")\n timeout =
kwargs.get(\"timeout\", strategy.timeout)\n\n # Default headers if still not present after browserforge attempt\n headers =
kwargs.get(\"headers\", {**DEFAULT_BROWSER_HEADERS, \"User-Agent\": CHROME_USER_AGENT})\n # Respect impersonate if provided,
otherwise default\n impersonate = kwargs.get(\"impersonate\", \"chrome110\")\n \n # Remove keys that
curl_requests.AsyncSession.request doesn't like\n clean_kwargs = {\n k: v for k, v in kwargs.items()\n if k not in
[\"timeout\", \"headers\", \"impersonate\"] + BROWSER_SPECIFIC_KWARGS\n }\n \n async with curl_requests.AsyncSession() as s:\n
resp = await s.request(\n method, \n url, \n timeout=timeout, \n headers=headers, \n impersonate=impersonate,\n
**clean_kwargs\n )\n return UnifiedResponse(resp.text, resp.status_code, resp.status_code, resp.url, resp.headers)\n\n if not
ASYNC_SESSIONS_AVAILABLE:\n raise ImportError(\"scrapling not available\")\n\n # Scrapling specific kwargs\n SCRAPLING_KWARGS
= [\"network_idle\", \"wait_selector\", \"wait_until\", \"stealth_mode\", \"block_resources\", \"timeout\"]\n scrapling_kwargs
= {k: v for k, v in kwargs.items() if k in SCRAPLING_KWARGS}\n\n # Propagate strategy values to scrapling if not explicitly
overridden in kwargs\n if \"timeout\" not in scrapling_kwargs:\n timeout_val = kwargs.get(\"timeout\", strategy.timeout)\n #
Scrapling/Playwright uses milliseconds for timeout\n scrapling_kwargs[\"timeout\"] = timeout_val * 1000\n if \"wait_until\"
not in scrapling_kwargs:\n scrapling_kwargs[\"wait_until\"] = strategy.wait_until or strategy.page_load_strategy\n if
\"network_idle\" not in scrapling_kwargs:\n scrapling_kwargs[\"network_idle\"] = strategy.network_idle\n if \"stealth_mode\"
not in scrapling_kwargs:\n scrapling_kwargs[\"stealth_mode\"] = strategy.stealth_mode\n if \"block_resources\" not in
scrapling_kwargs:\n scrapling_kwargs[\"block_resources\"] = strategy.block_resources\n \n # For other engines, we use
AsyncFetcher from scrapling\n if engine == BrowserEngine.CAMOUFOX:\n async with AsyncStealthySession(headless=True) as s:\n
resp = await s.fetch(url, method=method, **scrapling_kwargs)\n content = str(getattr(resp, 'body', getattr(resp,
'html_content', \"\")))\n return UnifiedResponse(content, resp.status, resp.status, resp.url, resp.headers)\n\n elif engine ==
BrowserEngine.PLAYWRIGHT_LEGACY:\n # Direct Playwright usage for cases where scrapling/camoufox fail\n from
playwright.async_api import async_playwright\n async with async_playwright() as p:\n browser = await
p.chromium.launch(headless=True)\n # Apply impersonation via context\n ua = kwargs.get(\"headers\", {}).get(\"User-Agent\",
CHROME_USER_AGENT)\n context = await browser.new_context(user_agent=ua)\n page = await context.new_page()\n\n timeout =
kwargs.get(\"timeout\", strategy.timeout) * 1000\n wait_until = \"networkidle\" if strategy.network_idle else
\"domcontentloaded\"\n\n # Apply headers\n if \"headers\" in kwargs:\n await
context.set_extra_http_headers(kwargs[\"headers\"])\n\n resp_obj = await page.goto(url, wait_until=wait_until,
timeout=timeout)\n content = await page.content()\n status = resp_obj.status if resp_obj else 0\n headers = resp_obj.headers
if resp_obj else {}\n\n await browser.close()\n return UnifiedResponse(content, status, status, url, headers)\n\n elif engine
```

```
== BrowserEngine.PLAYWRIGHT:\n async with AsyncDynamicSession(headless=True) as s:\n resp = await s.fetch(url, method=method,
**scrapling_kwargs)\n # Scrapling responses have a .text object that sometimes returns length 0\n # We ensure it's a string
from .body or .html_content\n content = str(getattr(resp, 'body', getattr(resp, 'html_content', \"\")))\n return
UnifiedResponse(content, resp.status, resp.status, resp.url, resp.headers)\n else:\n # Fallback to simple fetcher\n async with
AsyncFetcher() as fetcher:\n if method.upper() == \"GET\":\n resp = await fetcher.get(url, **kwargs)\n else:\n resp = await
fetcher.post(url, **kwargs)\n\n content = str(getattr(resp, 'body', getattr(resp, 'html_content', \"\")))\n return
UnifiedResponse(content, resp.status, resp.status, resp.url, resp.headers)\n\n\n async def close(self) -> None:\n \"\"\"\n
Shared resources are managed by GlobalResourceManager.\n This remains for API compatibility.\n \"\"\"\n pass\n",
"name": "SmartFetcher"
},
{
"type": "miscellaneous",
"content": "\n\n@dataclass\n"
},
{
"type": "class",
"content": "class CircuitBreaker:\n failure_threshold: int = 5\n recovery_timeout: float = 60.0\n state: str = \"closed\"\n
failure_count: int = 0\n last_failure_time: Optional[float] = None\n async def record_success(self) -> None:\n
self.failure_count = 0\n self.state = \"closed\"\n async def record_failure(self) -> None:\n self.failure_count += 1\n
self.last_failure_time = time.time()\n if self.failure_count >= self.failure_threshold: self.state = \"open\"\n async def
allow_request(self) -> bool:\n if self.state == \"closed\": return True\n if self.state == \"open\" and
self.last_failure_time:\n if time.time() - self.last_failure_time > self.recovery_timeout:\n self.state = \"half-open\"\n
return True\n return self.state == \"half-open\"\n",
"name": "CircuitBreaker"
},
{
"type": "miscellaneous",
"content": "\n\n@dataclass\n"
},
{
"type": "class",
"content": "class RateLimiter:\n requests_per_second: float = 10.0\n _tokens: float = field(default=10.0, init=False)\n
_last_update: float = field(default_factory=time.time, init=False)\n _lock: Optional[asyncio.Lock] = field(default=None,
init=False)\n _lock_sentinel: ClassVar[threading.Lock] = threading.Lock()\n\n def _get_lock(self) -> asyncio.Lock:\n if
self._lock is None:\n with self._lock_sentinel:\n if self._lock is None:\n try:\n loop = asyncio.get_running_loop()\n
self._lock = asyncio.Lock()\n except RuntimeError:\n pass\n return self._lock\n\n async def acquire(self) -> None:\n lock =
self._get_lock()\n for _ in range(1000): # Iteration limit to prevent potential hangs\n wait_time = 0\n async with lock:\n now
= time.time()\n elapsed = now - self._last_update\n self._tokens = min(self.requests_per_second, self._tokens + (elapsed *
self.requests_per_second))\n self._last_update = now\n if self._tokens >= 1:\n self._tokens -= 1\n return\n wait_time = (1 -
self._tokens) / self.requests_per_second\n\n if wait_time >= 0:\n await asyncio.sleep(max(wait_time, 0.01))\n",
"name": "RateLimiter"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class AdapterMetrics:\n def __init__(self) -> None:\n self._lock = threading.Lock()\n self.total_requests = 0\n
self.successful_requests = 0\n self.failed_requests = 0\n self.total_latency_ms = 0.0\n self.consecutive_failures = 0\n
self.last_failure_reason: Optional[str] = None\n @property\n def success_rate(self) -> float:\n return
self.successful_requests / self.total_requests if self.total_requests > 0 else 1.0\n async def record_success(self,
latency_ms: float) -> None:\n with self._lock:\n self.total_requests += 1\n self.successful_requests += 1\n
self.total_latency_ms += latency_ms\n self.consecutive_failures = 0\n self.last_failure_reason: Optional[str] = None\n async
def record_failure(self, error: str) -> None:\n with self._lock:\n self.total_requests += 1\n self.failed_requests += 1\n
self.consecutive_failures += 1\n self.last_failure_reason = error\n def snapshot(self) -> Dict[str, Any]:\n return {\n
\"total_requests\": self.total_requests,\n \"success_rate\": self.success_rate,\n \"failed_requests\": self.failed_requests,\n
\"consecutive_failures\": self.consecutive_failures,\n \"last_failure_reason\": getattr(self, \"last_failure_reason\", None)\n
}\n",
"name": "AdapterMetrics"
},
{
"type": "miscellaneous",
"content": "\n\n# --- MIXINS ---\n"
},
{
"type": "class",
"content": "class JSONParsingMixin:\n \"\"\"Mixin for safe JSON extraction from HTML and scripts.\"\"\"\n def
_parse_json_from_script(self, parser: HTMLParser, selector: str, context: str = \"script\") -> Optional[Any]:\n script =
parser.css_first(selector)\n if not script:\n return None\n try:\n return json.loads(script.text())\n except
json.JSONDecodeError as e:\n if hasattr(self, 'logger'):\n self.logger.error(\"failed_parsing_json\", context=context,
selector=selector, error=str(e))\n return None\n\n def _parse_json_from_attribute(self, parser: HTMLParser, selector: str,
attribute: str, context: str = \"attribute\") -> Optional[Any]:\n el = parser.css_first(selector)\n if not el:\n return None\n
raw = el.attributes.get(attribute)\n if not raw:\n return None\n try:\n return json.loads(html.unescape(raw))\n except
json.JSONDecodeError as e:\n if hasattr(self, 'logger'):\n self.logger.error(\"failed_parsing_json\", context=context,
selector=selector, attribute=attribute, error=str(e))\n return None\n\n def _parse_all_jsons_from_scripts(self, parser:
HTMLParser, selector: str, context: str = \"scripts\") -> List[Any]:\n results = []\n for script in parser.css(selector):\n
try:\n results.append(json.loads(script.text()))\n except json.JSONDecodeError as e:\n if hasattr(self, 'logger'):\n
self.logger.error(\"failed_parsing_json_in_list\", context=context, selector=selector, error=str(e))\n return results\n",
"name": "JSONParsingMixin"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
```

```
{
"type": "class",
"content": "class BrowserHeadersMixin:\n def _get_browser_headers(self, host: Optional[str] = None, referer: Optional[str] =
None, **extra: str) -> Dict[str, str]:\n h = {**DEFAULT_BROWSER_HEADERS, \"User-Agent\": CHROME_USER_AGENT, \"sec-ch-ua\":
CHROME_SEC_CH_UA, \"sec-ch-ua-mobile\": \"?0\", \"sec-ch-ua-platform\": '\"Windows\"'}\n if host: h[\"Host\"] = host\n if
referer: h[\"Referer\"] = referer\n h.update(extra)\n return h\n",
"name": "BrowserHeadersMixin"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class DebugMixin:\n def _save_debug_snapshot(self, content: str, context: str, url: Optional[str] = None) ->
None:\n if not content or not os.getenv(\"DEBUG_SNAPSHOTS\"): return\n try:\n d = Path(\"debug_snapshots\")\n
d.mkdir(parents=True, exist_ok=True)\n f = d / f\"{context}_{datetime.now(EASTERN).strftime('%Y%m%d_%H%M%S')}.html\"\n with
open(f, \"w\", encoding=\"utf-8\") as out:\n if url: out.write(f\"<!-- URL: {url} -->\\n\")\n out.write(content)\n except
Exception: pass\n def _save_debug_html(self, content: str, filename: str, **kwargs) -> None:\n
self._save_debug_snapshot(content, filename)\n",
"name": "DebugMixin"
},
{
"type": "miscellaneous",
"content": "\n\n"
},
{
"type": "class",
"content": "class RacePageFetcherMixin:\n async def _fetch_race_pages_concurrent(self, metadata: List[Dict[str, Any]],
headers: Dict[str, str], semaphore_limit: int = 5, delay_range: tuple[float, float] = (0.5, 1.5)) -> List[Dict[str, Any]]:\n
local_sem = asyncio.Semaphore(semaphore_limit)\n async def fetch_single(item):\n url = item.get(\"url\")\n if not url: return
None\n\n async with local_sem:\n # Stagger requests by sleeping inside the semaphore (Project Convention)\n await
asyncio.sleep(delay_range[0] + random.random() * (delay_range[1] - delay_range[0]))\n try:\n if hasattr(self, 'logger'):\n
self.logger.debug(\"fetching_race_page\", url=url)\n # make_request handles global_sem internally\n resp = None\n for attempt
in range(2): # 1 retry\n resp = await self.make_request(\"GET\", url, headers=headers)\n if resp and hasattr(resp, \"text\")
and resp.text and len(resp.text) > 500:\n break\n await asyncio.sleep(1 * (attempt + 1))\n\n if resp and hasattr(resp,
\"text\") and resp.text:\n if hasattr(self, 'logger'):\n self.logger.debug(\"fetched_race_page\", url=url,
status=getattr(resp, 'status', 'unknown'))\n return {**item, \"html\": resp.text}\n elif resp:\n if hasattr(self, 'logger'):\n
self.logger.warning(\"failed_fetching_race_page_unexpected_status\", url=url, status=getattr(resp, 'status', 'unknown'))\n
except Exception as e:\n if hasattr(self, 'logger'):\n self.logger.error(\"failed_fetching_race_page\", url=url,
error=str(e))\n return None\n tasks = [fetch_single(m) for m in metadata]\n results = await asyncio.gather(*tasks,
return_exceptions=True)\n return [r for r in results if not isinstance(r, Exception) and r is not None]\n",
"name": "RacePageFetcherMixin"
},
{
"type": "miscellaneous",
"content": "\n\n# --- BASE ADAPTER ---\n"
},
{
"type": "class",
"content": "class BaseAdapterV3(ABC):\n ADAPTER_TYPE: ClassVar[str] = \"discovery\"\n\n def __init__(self, source_name: str,
base_url: str, rate_limit: float = 10.0, config: Optional[Dict[str, Any]] = None, **kwargs: Any) -> None:\n self.source_name =
source_name\n self.base_url = base_url.rstrip(\"/\")\n self.config = config or {}\n # Merge kwargs into config\n
self.config.update(kwargs)\n self.trust_ratio = 0.0 # Tracking odds quality ratio (0.0 to 1.0)\n\n # Override rate_limit from
config if present\n actual_rate_limit = float(self.config.get(\"rate_limit\", rate_limit))\n\n self.logger =
structlog.get_logger(adapter_name=self.source_name)\n self.circuit_breaker = CircuitBreaker(\n
failure_threshold=int(self.config.get(\"failure_threshold\", 5)),\n
recovery_timeout=float(self.config.get(\"recovery_timeout\", 60.0))\n )\n self.rate_limiter =
RateLimiter(requests_per_second=actual_rate_limit)\n self.metrics = AdapterMetrics()\n self.smart_fetcher =
SmartFetcher(strategy=self._configure_fetch_strategy())\n self.last_race_count = 0\n self.last_duration_s = 0.0\n\n
@abstractmethod\n def _configure_fetch_strategy(self) -> FetchStrategy: pass\n @abstractmethod\n async def _fetch_data(self,
date: str) -> Optional[Any]: pass\n @abstractmethod\n def _parse_races(self, raw_data: Any) -> List[Race]: pass\n\n async def
get_races(self, date: str) -> List[Race]:\n start = time.time()\n try:\n # Check for browser requirement in monolith mode\n
strategy = self.smart_fetcher.strategy\n if is_frozen() and strategy.primary_engine in [BrowserEngine.PLAYWRIGHT,
BrowserEngine.CAMOUFOX]:\n self.logger.info(\"Skipping browser-dependent adapter in monolith mode\")\n return []\n\n if not
await self.circuit_breaker.allow_request(): return []\n await self.rate_limiter.acquire()\n raw = await
self._fetch_data(date)\n if not raw:\n await self.circuit_breaker.record_failure()\n return []\n races =
self._validate_and_parse_races(raw)\n self.last_race_count = len(races)\n self.last_duration_s = time.time() - start\n await
self.circuit_breaker.record_success()\n await self.metrics.record_success(self.last_duration_s * 1000)\n return races\n except
Exception as e:\n self.logger.error(\"Adapter failed\", error=str(e))\n await self.circuit_breaker.record_failure()\n await
self.metrics.record_failure(str(e))\n return []\n\n def _validate_and_parse_races(self, raw_data: Any) -> List[Race]:\n races
= self._parse_races(raw_data)\n total_runners = 0\n trustworthy_runners = 0\n\n for r in races:\n # Global heuristic for
runner numbers (addressing \"impossible\" high numbers)\n active_runners = [run for run in r.runners if not run.scratched]\n
field_size = len(active_runners)\n\n # If any runner has a number > 20 and it's also > field_size + 10 (buffer)\n # or if it's
extremely high (> 100), re-index everything as it's likely a parsing error (horse IDs).\n # Also re-index if all numbers are
missing/zero.\n suspicious = all(run.number == 0 or run.number is None for run in r.runners)\n if not suspicious:\n for run in
r.runners:\n if run.number:\n if run.number > 100 or (run.number > 20 and run.number > field_size + 10):\n suspicious = True\n
break\n\n if suspicious:\n self.logger.warning(\"suspicious_runner_numbers\", venue=r.venue, field_size=field_size)\n for i,
run in enumerate(r.runners):\n run.number = i + 1\n\n for runner in r.runners:\n if not runner.scratched:\n # Explicitly
enrich win_odds using all available sources (including fallbacks)\n best = _get_best_win_odds(runner)\n # Untrustworthy odds
should be flagged (Memory Directive Fix)\n is_trustworthy = best is not None\n runner.metadata[\"odds_source_trustworthy\"] =
is_trustworthy\n if best:\n runner.win_odds = float(best)\n trustworthy_runners += 1\n total_runners += 1\n\n if total_runners
> 0:\n self.trust_ratio = round(trustworthy_runners / total_runners, 2)\n self.logger.info(\"adapter_odds_quality\",
ratio=self.trust_ratio, source=self.source_name)\n\n valid, warnings = DataValidationPipeline.validate_parsed_races(races,
adapter_name=self.source_name)\n return valid\n\n async def make_request(self, method: str, url: str, **kwargs: Any) -> Any:\n
```

```
full_url = url if url.startswith(\"http\") else f\"{self.base_url}/{url.lstrip('/')}\"\n self.logger.debug(\"Requesting\",
method=method, url=full_url)\n # Apply global concurrency limit (Memory Directive Fix)\n async with
GlobalResourceManager.get_global_semaphore():\n try:\n # Use adapter-specific strategy\n kwargs.setdefault(\"strategy\",
self.smart_fetcher.strategy)\n resp = await self.smart_fetcher.fetch(full_url, method=method, **kwargs)\n status =
get_resp_status(resp)\n self.logger.debug(\"Response received\", method=method, url=full_url, status=status)\n return resp\n
except Exception as e:\n self.logger.error(\"Request failed\", method=method, url=full_url, error=str(e))\n return None\n\n
async def close(self) -> None: await self.smart_fetcher.close()\n async def shutdown(self) -> None: await self.close()\n",
"name": "BaseAdapterV3"
},
{
"type": "miscellaneous",
"content": "\n# ============================================================================\n# ADAPTER IMPLEMENTATIONS\n#
============================================================================\n\n# ---------------------------------------\n#
EquibaseAdapter\n# ---------------------------------------\n"
},
{
"type": "class",
"content": "class RacingAndSportsAdapter(BrowserHeadersMixin, DebugMixin, RacePageFetcherMixin, BaseAdapterV3):\n \"\"\"\n
Adapter for Racing & Sports (RAS).\n Note: Highly protected by Cloudflare; requires advanced impersonation.\n \"\"\"\n
SOURCE_NAME: ClassVar[str] = \"RacingAndSports\"\n BASE_URL: ClassVar[str] = \"https://www.racingandsports.com.au\"\n\n def
__init__(self, config: Optional[Dict[str, Any]] = None) -> None:\n super().__init__(source_name=self.SOURCE_NAME,
base_url=self.BASE_URL, config=config)\n\n def _configure_fetch_strategy(self) -> FetchStrategy:\n return FetchStrategy(\n
primary_engine=BrowserEngine.CURL_CFFI,\n enable_js=True,\n stealth_mode=\"camouflage\",\n timeout=60\n )\n\n def
_get_headers(self) -> Dict[str, str]:\n return self._get_browser_headers(host=\"www.racingandsports.com.au\")\n\n async def
_fetch_data(self, date: str) -> Optional[Dict[str, Any]]:\n url = f\"/racing-index?date={date}\"\n resp = await
self.make_request(\"GET\", url, headers=self._get_headers())\n if not resp or not resp.text:\n return None\n\n
self._save_debug_snapshot(resp.text, f\"ras_index_{date}\")\n parser = HTMLParser(resp.text)\n metadata = []\n\n # RAS uses
tables for different regions (Australia, UK, etc.)\n for table in parser.css(\"table.table-index\"):\n for row in
table.css(\"tbody tr\"):\n venue_cell = row.css_first(\"td.venue-name\")\n if not venue_cell: continue\n venue_name =
venue_cell.text(strip=True)\n\n for link in row.css(\"td a.race-link\"):\n race_url = link.attributes.get(\"href\", \"\")\n if
not race_url: continue\n if not race_url.startswith(\"http\"):\n race_url = self.BASE_URL + race_url\n\n r_num_match =
re.search(r\"R(\\d+)\", link.text(strip=True))\n r_num = int(r_num_match.group(1)) if r_num_match else 0\n\n
metadata.append({\n \"url\": race_url,\n \"venue\": venue_name,\n \"race_number\": r_num\n })\n\n if not metadata:\n return
None\n\n # Limit for sanity\n pages = await self._fetch_race_pages_concurrent(metadata[:40], self._get_headers())\n return
{\"pages\": pages, \"date\": date}\n\n def _parse_races(self, raw_data: Any) -> List[Race]:\n if not raw_data or not
raw_data.get(\"pages\"): return []\n try: race_date = datetime.strptime(raw_data[\"date\"], \"%Y-%m-%d\").date()\n except
Exception: return []\n\n races: List[Race] = []\n for item in raw_data[\"pages\"]:\n html_content = item.get(\"html\")\n if
not html_content: continue\n try:\n race = self._parse_single_race(html_content, item.get(\"url\", \"\"), race_date,
item.get(\"venue\"), item.get(\"race_number\"))\n if race: races.append(race)\n except Exception: pass\n return races\n\n def
_parse_single_race(self, html_content: str, url: str, race_date: date, venue: str, race_num: int) -> Optional[Race]:\n tree =
HTMLParser(html_content)\n\n runners = []\n for row in tree.css(\"tr.runner-row\"):\n name_node =
row.css_first(\".runner-name\")\n if not name_node: continue\n name = clean_text(name_node.text())\n\n num_node =
row.css_first(\".runner-number\")\n number = int(\"\".join(filter(str.isdigit, num_node.text()))) if num_node else 0\n\n
odds_node = row.css_first(\".odds-win\")\n win_odds = parse_odds_to_decimal(clean_text(odds_node.text())) if odds_node else
None\n\n odds_data = {}\n if ov := create_odds_data(self.SOURCE_NAME, win_odds):\n odds_data[self.SOURCE_NAME] = ov\n\n
runners.append(Runner(name=name, number=number, odds=odds_data, win_odds=win_odds))\n\n if not runners: return None\n\n #
Start time from page if available, else guess\n start_time = datetime.combine(race_date, datetime.min.time())\n # Try to find
time in text\n time_match = re.search(r\"(\\d{1,2}:\\d{2})\", html_content)\n if time_match:\n try:\n start_time =
datetime.combine(race_date, datetime.strptime(time_match.group(1), \"%H:%M\").time())\n except Exception: pass\n\n return
Race(\n id=generate_race_id(\"ras\", venue, start_time, race_num),\n venue=venue,\n race_number=race_num,\n
start_time=ensure_eastern(start_time),\n runners=runners,\n source=self.SOURCE_NAME,\n
available_bets=scrape_available_bets(html_content)\n )\n",
"name": "RacingAndSportsAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class SkyRacingWorldAdapter(BrowserHeadersMixin, DebugMixin, RacePageFetcherMixin, BaseAdapterV3):\n SOURCE_NAME:
ClassVar[str] = \"SkyRacingWorld\"\n BASE_URL: ClassVar[str] = \"https://www.skyracingworld.com\"\n\n def __init__(self,
config: Optional[Dict[str, Any]] = None) -> None:\n super().__init__(source_name=self.SOURCE_NAME, base_url=self.BASE_URL,
config=config)\n\n def _configure_fetch_strategy(self) -> FetchStrategy:\n return FetchStrategy(\n
primary_engine=BrowserEngine.CURL_CFFI,\n enable_js=True,\n stealth_mode=\"camouflage\",\n timeout=60\n )\n\n def
_get_headers(self) -> Dict[str, str]:\n return self._get_browser_headers(host=\"www.skyracingworld.com\")\n\n async def
make_request(self, method: str, url: str, **kwargs: Any) -> Any:\n kwargs.setdefault(\"impersonate\", \"chrome120\")\n return
await super().make_request(method, url, **kwargs)\n\n async def _fetch_data(self, date: str) -> Optional[Dict[str, Any]]:\n #
Index for the day\n index_url = f\"/form-guide/thoroughbred/{date}\"\n resp = await self.make_request(\"GET\", index_url,
headers=self._get_headers())\n if not resp or not resp.text:\n if resp: self.logger.warning(\"Unexpected status\",
status=resp.status, url=index_url)\n return None\n self._save_debug_snapshot(resp.text, f\"skyracing_index_{date}\")\n\n
parser = HTMLParser(resp.text)\n track_links = defaultdict(list)\n now = now_eastern()\n today_str =
now.strftime(\"%Y-%m-%d\")\n\n # Optimization: If it's late in ET, skip countries that are finished\n # Europe/Turkey/SA
usually finished by 18:00 ET\n skip_finished_countries = (now.hour >= 18 or now.hour < 6) and (date == today_str)\n
finished_keywords = [\"turkey\", \"south-africa\", \"united-kingdom\", \"france\", \"germany\", \"dubai\", \"bahrain\"]\n\n
for link in parser.css(\"a.fg-race-link\"):\n url = link.attributes.get(\"href\")\n if url:\n if not
url.startswith(\"http\"):\n url = self.BASE_URL + url\n\n if skip_finished_countries:\n if any(kw in url.lower() for kw in
finished_keywords):\n continue\n\n # Group by track (everything before R#)\n track_key = re.sub(r'/R\\d+$', '', url)\n
track_links[track_key].append(url)\n\n metadata = []\n for t_url in track_links:\n # For discovery, we usually only care about
upcoming races.\n # Without times in index, we pick R1 as a guess, but if we have multiple,\n # R1 might be in the past.
However, picking R1 is the safest if we want \"one per track\".\n if track_links[t_url]:\n metadata.append({\"url\":
track_links[t_url][0]})\n\n if not metadata:\n self.logger.warning(\"No metadata found\", context=\"SRW Index Parsing\",
url=index_url)\n return None\n # Limit to first 50 to avoid hammering\n pages = await
self._fetch_race_pages_concurrent(metadata[:50], self._get_headers(), semaphore_limit=5)\n return {\"pages\": pages, \"date\":
date}\n\n def _parse_races(self, raw_data: Any) -> List[Race]:\n if not raw_data or not raw_data.get(\"pages\"): return []\n
```

```
try: race_date = datetime.strptime(raw_data[\"date\"], \"%Y-%m-%d\").date()\n except Exception: return []\n races: List[Race] = []\n for item in raw_data[\"pages\"]:\n html_content = item.get(\"html\")\n if not html_content: continue\n try:\n race = self._parse_single_race(html_content, item.get(\"url\", \"\"), race_date)\n if race: races.append(race)\n except Exception: pass\n return races\n\n def _parse_single_race(self, html_content: str, url: str, race_date: date) -> Optional[Race]:\n parser = HTMLParser(html_content)\n\n # Extract venue and time from header\n # Format usually: \"14:30 LINGFIELD\" or similar\n header = parser.css_first(\".sdc-site-racing-header__name\") or parser.css_first(\"h1\") or parser.css_first(\"h2\")\n if not header: return None\n\n header_text = clean_text(header.text())\n match = re.search(r\"(\\d{1,2}:\\d{2})\\s+(.+)\", header_text)\n if match:\n time_str = match.group(1)\n venue = normalize_venue_name(match.group(2))\n else:\n venue = normalize_venue_name(header_text)\n time_str = \"12:00\" # Fallback\n\n try:\n start_time = datetime.combine(race_date, datetime.strptime(time_str, \"%H:%M\").time())\n except Exception:\n start_time = datetime.combine(race_date, datetime.min.time())\n\n # Race number from URL\n race_num = 1\n num_match = re.search(r'/R(\\d+)$', url)\n if num_match:\n race_num = int(num_match.group(1))\n\n runners = []\n # Try different selectors for runners\n for row in parser.css(\".runner_row\") or parser.css(\".mobile-runner\"):\n try:\n name_node = row.css_first(\".horseName\") or row.css_first(\"a[href*='/horse/']\")\n if not name_node: continue\n name = clean_text(name_node.text())\n\n num_node = row.css_first(\".tdContent b\") or row.css_first(\"[data-tab-no]\")\n number = 0\n if num_node:\n if num_node.attributes.get(\"data-tab-no\"):\n number = int(num_node.attributes.get(\"data-tab-no\"))\n else:\n digits = \"\".join(filter(str.isdigit, num_node.text()))\n if digits: number = int(digits)\n\n scratched = \"strikeout\" in (row.attributes.get(\"class\") or \"\").lower() or row.attributes.get(\"data-scratched\") == \"True\"\n\n win_odds = None\n odds_node = row.css_first(\".pa_odds\") or row.css_first(\".odds\")\n if odds_node:\n win_odds = parse_odds_to_decimal(clean_text(odds_node.text()))\n\n if win_odds is None:\n win_odds = SmartOddsExtractor.extract_from_node(row)\n\n od = {}\n if ov := create_odds_data(self.SOURCE_NAME, win_odds):\n od[self.SOURCE_NAME] = ov\n\n runners.append(Runner(name=name, number=number, scratched=scratched, odds=od, win_odds=win_odds))\n except Exception: continue\n\n if not runners: return None\n\n disc = detect_discipline(html_content)\n return Race(\n id=generate_race_id(\"srw\", venue, start_time, race_num, disc),\n venue=venue,\n race_number=race_num,\n start_time=start_time,\n runners=runners,\n discipline=disc,\n source=self.SOURCE_NAME,\n available_bets=scrape_available_bets(html_content)\n )\n",
"name": "SkyRacingWorldAdapter"
},
{
"type": "miscellaneous",
"content": "\n# ----------------------------------------\n# AtTheRacesAdapter\n# ----------------------------------------\n"
},
{
"type": "class",
"content": "class AtTheRacesAdapter(BrowserHeadersMixin, DebugMixin, RacePageFetcherMixin, BaseAdapterV3):\n SOURCE_NAME: ClassVar[str] = \"AtTheRaces\"\n BASE_URL: ClassVar[str] = \"https://www.attheraces.com\"\n\n def _configure_fetch_strategy(self) -> FetchStrategy:\n return FetchStrategy(primary_engine=BrowserEngine.CURL_CFFI, enable_js=True, stealth_mode=\"camouflage\")\n\n async def make_request(self, method: str, url: str, **kwargs: Any) -> Any:\n kwargs.setdefault(\"impersonate\", \"chrome120\")\n return await super().make_request(method, url, **kwargs)\n\n SELECTORS: ClassVar[Dict[str, List[str]]] = {\n \"race_links\": ['a.race-navigation-link', 'a.sidebar-racecardsigation-link', 'a[href^=\"/racecard/\"]', 'a[href*=\"/racecard/\"]'],\n \"details_container\": [\".race-header__details--primary\", \"atr-racecard-race-header .container\", \".racecard-header .container\"],\n \"track_name\": [\"h2\", \"h1 a\", \"h1\"],\n \"race_time\": [\"h2 b\", \"h1 span\", \".race-time\"],\n \"distance\": [\".race-header__details--secondary .p--large\", \".race-header__details--secondary div\"],\n \"runners\": [\".card-cell--horse\", \".odds-grid-horse\", \"atr-horse-in-racecard\", \".horse-in-racecard\"],\n }\n\n def __init__(self, config: Optional[Dict[str, Any]] = None) -> None:\n super().__init__(source_name=self.SOURCE_NAME, base_url=self.BASE_URL, config=config)\n\n def _get_headers(self) -> Dict[str, str]:\n return self._get_browser_headers(host=\"www.attheraces.com\", referer=\"https://www.attheraces.com/racecards\")\n\n async def _fetch_data(self, date: str) -> Optional[Dict[str, Any]]:\n index_url = f\"/racecards/{date}\"\n intl_url = f\"/racecards/international/{date}\"\n\n resp = await self.make_request(\"GET\", index_url, headers=self._get_headers())\n intl_resp = await self.make_request(\"GET\", intl_url, headers=self._get_headers())\n\n metadata = []\n if resp and resp.text:\n self._save_debug_snapshot(resp.text, f\"atr_index_{date}\")\n parser = HTMLParser(resp.text)\n metadata.extend(self._extract_race_metadata(parser, date))\n\n elif resp:\n self.logger.warning(\"Unexpected status\", status=resp.status, url=index_url)\n\n if intl_resp and intl_resp.text:\n self._save_debug_snapshot(intl_resp.text, f\"atr_intl_index_{date}\")\n intl_parser = HTMLParser(intl_resp.text)\n metadata.extend(self._extract_race_metadata(intl_parser, date))\n elif intl_resp:\n self.logger.warning(\"Unexpected status\", status=intl_resp.status, url=intl_url)\n\n if not metadata:\n self.logger.warning(\"No metadata found\", context=\"ATR Index Parsing\", date=date)\n return None\n pages = await self._fetch_race_pages_concurrent(metadata, self._get_headers(), semaphore_limit=5)\n return {\"pages\": pages, \"date\": date}\n\n def _extract_race_metadata(self, parser: HTMLParser, date_str: str) -> List[Dict[str, Any]]:\n meta: List[Dict[str, Any]] = []\n track_map = defaultdict(list)\n\n try:\n target_date = datetime.strptime(date_str, \"%Y-%m-%d\").date()\n except Exception:\n target_date = datetime.now(EASTERN).date()\n\n for link in parser.css('a[href*=\"/racecard/\"]'):\n url = link.attributes.get(\"href\")\n if not url: continue\n # Look for time at end of URL: /racecard/venue/date/1330\n time_match = re.search(r\"/(\\d{4})$\", url)\n if not time_match:\n # Might be just a race number: /racecard/venue/date/1\n if not re.search(r\"/\\d{1,2}$\", url): continue\n\n parts = url.split(\"/\")\n if len(parts) >= 3:\n track_name = parts[2]\n time_str = time_match.group(1) if time_match else None\n track_map[track_name].append({\"url\": url, \"time_str\": time_str})\n\n # Site usually shows UK time\n site_tz = ZoneInfo(\"Europe/London\")\n now_site = datetime.now(site_tz)\n\n for track, race_infos in track_map.items():\n # Broaden window to capture multiple races (Memory Directive Fix)\n for r in race_infos:\n if r[\"time_str\"]:\n try:\n rt = datetime.strptime(r[\"time_str\"], \"%H%M\").replace(\n year=target_date.year, month=target_date.month, day=target_date.day, tzinfo=site_tz\n )\n diff = (rt - now_site).total_seconds() / 60\n if not (-45 < diff <= 1080):\n continue\n meta.append({\"url\": r[\"url\"], \"race_number\": 1, \"venue_raw\": track})\n except Exception: pass\n\n if not meta:\n for meeting in (parser.css(\".meeting-summary\") or parser.css(\".p-meetings__item\")):\n for link in meeting.css('a[href*=\"/racecard/\"]'):\n if url := link.attributes.get(\"href\"):\n meta.append({\"url\": url, \"race_number\": 1})\n return meta\n\n def _parse_races(self, raw_data: Any) -> List[Race]:\n if not raw_data or not raw_data.get(\"pages\"): return []\n try: race_date = datetime.strptime(raw_data[\"date\"], \"%Y-%m-%d\").date()\n except Exception: return []\n races: List[Race] = []\n for item in raw_data[\"pages\"]:\n html_content = item.get(\"html\")\n if not html_content: continue\n try:\n race = self._parse_single_race(html_content, item.get(\"url\", \"\"), race_date, item.get(\"race_number\"))\n if race: races.append(race)\n except Exception: pass\n return races\n\n def _parse_single_race(self, html_content: str, url_path: str, race_date: date, race_number_fallback: Optional[int]) -> Optional[Race]:\n parser = HTMLParser(html_content)\n track_name, time_str, header_text = None, None, \"\"\n header = parser.css_first(\".race-header__details\") or parser.css_first(\".racecard-header\")\n if header:\n header_text = clean_text(header.text()) or \"\"\n time_match = re.search(r\"(\\d{1,2}:\\d{2})\", header_text)\n if time_match:\n time_str = time_match.group(1)\n track_raw = re.sub(r\"\\d{1,2}\\s+[A-Za-z]{3}\\s+\\d{4}\", \"\", header_text.replace(time_str, \"\")).strip()\n track_raw = re.split(r\"\\s+Race\\s+\\d+\", track_raw, flags=re.I)[0]\n track_raw = re.sub(r\"^\\d+\\s+\", \"\", track_raw).split(\" - \")[0].split(\"|\")[0].strip()\n track_name = normalize_venue_name(track_raw)\n if not track_name:\n details = parser.css_first(\".race-header__details--primary\")\n if details:\n track_node =
```

```
details.css_first(\"h2\") or details.css_first(\"h1 a\") or details.css_first(\"h1\")\n if track_node: track_name =
normalize_venue_name(clean_text(track_node.text()))\n if not time_str:\n time_node = details.css_first(\"h2 b\") or
details.css_first(\".race-time\")\n if time_node: time_str = clean_text(time_node.text()).replace(\" ATR\", \"\")\n if not
track_name:\n parts = url_path.split(\"/\")\n if len(parts) >= 3: track_name = normalize_venue_name(parts[2])\n if not
time_str:\n parts = url_path.split(\"/\")\n if len(parts) >= 5 and re.match(r\"\\d{4}\", parts[-1]):\n raw_time = parts[-1]\n
time_str = f\"{raw_time[:2]}:{raw_time[2:]}\"\n if not track_name or not time_str: return None\n try: start_time =
datetime.combine(race_date, datetime.strptime(time_str, \"%H:%M\").time())\n except Exception: return None\n race_number =
race_number_fallback or 1\n distance = None\n dist_match = re.search(r\"\\|\\s*(\\d+[mfy].*)\", header_text, re.I)\n if
dist_match: distance = dist_match.group(1).strip()\n runners = self._parse_runners(parser)\n if not runners: return None\n
return Race(discipline=\"Thoroughbred\", id=generate_race_id(\"atr\", track_name, start_time, race_number), venue=track_name,
race_number=race_number, start_time=start_time, runners=runners, distance=distance, source=self.source_name,
available_bets=scrape_available_bets(html_content))\n\n def _parse_runners(self, parser: HTMLParser) -> List[Runner]:\n
odds_map: Dict[str, float] = {}\n for row in parser.css(\".odds-grid__row--horse\"):\n if m := re.search(r\"row-(\\d+)\",
row.attributes.get(\"id\", \"\")):\n if price := row.attributes.get(\"data-bestprice\"):\n try:\n p_val = float(price)\n if
is_valid_odds(p_val): odds_map[m.group(1)] = p_val\n except Exception: pass\n runners: List[Runner] = []\n for selector in
self.SELECTORS[\"runners\"]:\n nodes = parser.css(selector)\n if nodes:\n for i, node in enumerate(nodes):\n runner =
self._parse_runner(node, odds_map, i + 1)\n if runner: runners.append(runner)\n break\n return runners\n\n def
_parse_runner(self, row: Node, odds_map: Dict[str, float], fallback_number: int = 0) -> Optional[Runner]:\n try:\n name_node =
row.css_first(\"h3\") or row.css_first(\"a.horse__link\") or row.css_first('a[href*=\"/form/horse/\"]')\n if not name_node:
return None\n name = clean_text(name_node.text())\n if not name: return None\n num_node =
row.css_first(\".horse-in-racecard__saddle-cloth-number\") or row.css_first(\".odds-grid-horse__no\")\n number = 0\n if
num_node:\n ns = clean_text(num_node.text())\n if ns:\n digits = \"\".join(filter(str.isdigit, ns))\n if digits: number =
int(digits)\n\n if number == 0 or number > 40:\n number = fallback_number\n win_odds = None\n if horse_link :=
row.css_first('a[href*=\"/form/horse/\"]'):\n if m := re.search(r\"/(\\d+)(\\?|$)\", horse_link.attributes.get(\"href\",
\"\")):\n win_odds = odds_map.get(m.group(1))\n if win_odds is None:\n if odds_node :=
row.css_first(\".horse-in-racecard__odds\"):\n win_odds = parse_odds_to_decimal(clean_text(odds_node.text()))\n\n # Advanced
heuristic fallback\n if win_odds is None:\n win_odds = SmartOddsExtractor.extract_from_node(row)\n\n odds: Dict[str, OddsData]
= {}\n if od := create_odds_data(self.source_name, win_odds): odds[self.source_name] = od\n return Runner(number=number,
name=name, odds=odds, win_odds=win_odds)\n except Exception: return None\n",
"name": "AtTheRacesAdapter"
},
{
"type": "miscellaneous",
"content": "\n# ----------------------------------------\n# AtTheRacesGreyhoundAdapter\n#
----------------------------------------\n"
},
{
"type": "class",
"content": "class AtTheRacesGreyhoundAdapter(JSONParsingMixin, BrowserHeadersMixin, DebugMixin, RacePageFetcherMixin,
BaseAdapterV3):\n SOURCE_NAME: ClassVar[str] = \"AtTheRacesGreyhound\"\n BASE_URL: ClassVar[str] =
\"https://greyhounds.attheraces.com\"\n\n def __init__(self, config: Optional[Dict[str, Any]] = None) -> None:\n
super().__init__(source_name=self.SOURCE_NAME, base_url=self.BASE_URL, config=config)\n\n def _configure_fetch_strategy(self)
-> FetchStrategy:\n return FetchStrategy(primary_engine=BrowserEngine.CURL_CFFI, enable_js=True, stealth_mode=\"camouflage\",
timeout=45)\n\n def _get_headers(self) -> Dict[str, str]:\n return
self._get_browser_headers(host=\"greyhounds.attheraces.com\", referer=\"https://greyhounds.attheraces.com/racecards\")\n\n
async def _fetch_data(self, date: str) -> Optional[Dict[str, Any]]:\n index_url = f\"/racecards/{date}\" if date else
\"/racecards\"\n resp = await self.make_request(\"GET\", index_url, headers=self._get_headers())\n if not resp or not
resp.text:\n if resp: self.logger.warning(\"Unexpected status\", status=resp.status, url=index_url)\n return None\n
self._save_debug_snapshot(resp.text, f\"atr_grey_index_{date}\")\n parser = HTMLParser(resp.text)\n metadata =
self._extract_race_metadata(parser, date)\n if not metadata:\n links = []\n scripts =
self._parse_all_jsons_from_scripts(parser, 'script[type=\"application/ld+json\"]', context=\"ATR Greyhound Index\")\n for d in
scripts:\n items = d.get(\"@graph\", [d]) if isinstance(d, dict) else []\n for item in items:\n if item.get(\"@type\") ==
\"SportsEvent\":\n loc = item.get(\"location\")\n if isinstance(loc, list):\n for l in loc:\n if u := l.get(\"url\"):
links.append(u)\n elif isinstance(loc, dict):\n if u := loc.get(\"url\"): links.append(u)\n metadata = [{\"url\": l,
\"race_number\": 0} for l in set(links)]\n if not metadata:\n self.logger.warning(\"No metadata found\", context=\"ATR
Greyhound Index Parsing\", url=index_url)\n return None\n pages = await self._fetch_race_pages_concurrent(metadata,
self._get_headers(), semaphore_limit=5)\n return {\"pages\": pages, \"date\": date}\n\n def _extract_race_metadata(self,
parser: HTMLParser, date_str: str) -> List[Dict[str, Any]]:\n meta: List[Dict[str, Any]] = []\n pc =
parser.css_first(\"page-content\")\n if not pc: return []\n items_raw = pc.attributes.get(\":items\") or
pc.attributes.get(\":modules\")\n if not items_raw: return []\n\n try:\n target_date = datetime.strptime(date_str,
\"%Y-%m-%d\").date()\n except Exception:\n target_date = datetime.now(EASTERN).date()\n\n # Usually UK time\n site_tz =
ZoneInfo(\"Europe/London\")\n now_site = datetime.now(site_tz)\n\n try:\n modules = json.loads(html.unescape(items_raw))\n for
module in modules:\n for meeting in module.get(\"data\", {}).get(\"items\", []):\n # Broaden window to capture multiple races
(Memory Directive Fix)\n races = [r for r in meeting.get(\"items\", []) if r.get(\"type\") == \"racecard\"]\n\n for race in
races:\n r_time_str = race.get(\"time\") # Usually HH:MM\n if r_time_str:\n try:\n rt = datetime.strptime(r_time_str,
\"%H:%M\").replace(\n year=target_date.year, month=target_date.month, day=target_date.day, tzinfo=site_tz\n )\n diff = (rt -
now_site).total_seconds() / 60\n if not (-45 < diff <= 1080):\n continue\n\n r_num = race.get(\"raceNumber\") or
race.get(\"number\") or 1\n if u := race.get(\"cta\", {}).get(\"href\"):\n if \"/racecard/\" in u:\n meta.append({\"url\": u,
\"race_number\": r_num})\n except Exception: pass\n except Exception: pass\n return meta\n\n def _parse_races(self, raw_data:
Any) -> List[Race]:\n if not raw_data or not raw_data.get(\"pages\"): return []\n try: race_date =
datetime.strptime(raw_data.get(\"date\", \"\"), \"%Y-%m-%d\").date()\n except Exception: race_date =
datetime.now(EASTERN).date()\n races: List[Race] = []\n for item in raw_data[\"pages\"]:\n if not item or not
item.get(\"html\"): continue\n try:\n race = self._parse_single_race(item[\"html\"], item.get(\"url\", \"\"), race_date,
item.get(\"race_number\"))\n if race: races.append(race)\n except Exception: pass\n return races\n\n def
_parse_single_race(self, html_content: str, url_path: str, race_date: date, race_number: Optional[int]) -> Optional[Race]:\n
parser = HTMLParser(html_content)\n pc = parser.css_first(\"page-content\")\n if not pc: return None\n items_raw =
pc.attributes.get(\":items\") or pc.attributes.get(\":modules\")\n if not items_raw: return None\n try: modules =
json.loads(html.unescape(items_raw))\n except Exception: return None\n venue, race_time_str, distance, runners, odds_map =
\"\", \"\", \"\", [], {}\n\n # Try to extract venue from title as high-priority fallback\n title_node =
parser.css_first(\"title\")\n if title_node:\n title_text = title_node.text().strip()\n # Title: \"14:26 Oxford Greyhound
Racecard...\"\n tm = re.search(r'\\d{1,2}:\\d{2}\\s+(.+?)\\s+Greyhound', title_text)\n if tm:\n venue =
normalize_venue_name(tm.group(1))\n for module in modules:\n m_type, m_data = module.get(\"type\"), module.get(\"data\", {})\n
if m_type == \"RacecardHero\":\n venue = normalize_venue_name(m_data.get(\"track\", \"\"))\n race_time_str =
m_data.get(\"time\", \"\")\n distance = m_data.get(\"distance\", \"\")\n if not race_number: race_number =
m_data.get(\"raceNumber\") or m_data.get(\"number\")\n elif m_type == \"OddsGrid\":\n odds_grid = m_data.get(\"oddsGrid\",
```

```
{})\n\n # If venue still empty, try to get it from OddsGrid data\n if not venue:\n venue =
normalize_venue_name(odds_grid.get(\"track\", \"\"))\n if not race_time_str:\n race_time_str = odds_grid.get(\"time\", \"\")\n
if not distance:\n distance = odds_grid.get(\"distance\", \"\")\n\n partners = odds_grid.get(\"partners\", {})\n all_partners
= []\n if isinstance(partners, dict):\n for p_list in partners.values(): all_partners.extend(p_list)\n elif
isinstance(partners, list): all_partners = partners\n for partner in all_partners:\n for o in partner.get(\"odds\", []):\n
g_id = o.get(\"betParams\", {}).get(\"greyhoundId\")\n price = o.get(\"value\", {}).get(\"decimal\")\n if g_id and price:\n
p_val = parse_odds_to_decimal(price)\n if p_val and is_valid_odds(p_val): odds_map[str(g_id)] = p_val\n for t in
odds_grid.get(\"traps\", []):\n trap_num = t.get(\"trap\", 0)\n name = clean_text(t.get(\"name\", \"\")) or \"\"\n g_id_match
= re.search(r\"/greyhound/(\\d+)\", t.get(\"href\", \"\"))\n g_id = g_id_match.group(1) if g_id_match else None\n win_odds =
odds_map.get(str(g_id)) if g_id else None\n\n # Advanced heuristic fallback\n if win_odds is None:\n win_odds =
SmartOddsExtractor.extract_from_text(str(t))\n\n\n odds_data = {}\n if ov := create_odds_data(self.source_name, win_odds):
odds_data[self.source_name] = ov\n runners.append(Runner(number=trap_num or 0, name=name, odds=odds_data,
win_odds=win_odds))\n\n url_parts = url_path.split(\"/\")\n if not venue:\n # /racecard/GB/oxford/10-February-2026/1426\n m =
re.search(r'/(?:racecard|result)/[A-Z]{2,3}/([^/]+)', url_path)\n if m:\n venue = normalize_venue_name(m.group(1))\n if not
race_time_str and len(url_parts) >= 5:\n race_time_str = url_parts[-1]\n if not venue or not runners: return None\n try:\n if
\":\" not in race_time_str and len(race_time_str) == 4: race_time_str = f\"{race_time_str[:2]}:{race_time_str[2:]}\"\n
start_time = datetime.combine(race_date, datetime.strptime(race_time_str, \"%H:%M\").time())\n except Exception: return None\n
return Race(discipline=\"Greyhound\", id=generate_race_id(\"atrg\", venue, start_time, race_number or 0, \"Greyhound\"),
venue=venue, race_number=race_number or 0, start_time=start_time, runners=runners, distance=str(distance) if distance else
None, source=self.source_name, available_bets=scrape_available_bets(html_content))\n",
"name": "AtTheRacesGreyhoundAdapter"
},
{
"type": "miscellaneous",
"content": "\n# ----------------------------------------\n# BoyleSportsAdapter\n# ----------------------------------------\n"
},
{
"type": "class",
"content": "class BoyleSportsAdapter(BrowserHeadersMixin, DebugMixin, BaseAdapterV3):\n SOURCE_NAME: ClassVar[str] =
\"BoyleSports\"\n BASE_URL: ClassVar[str] = \"https://www.boylesports.com\"\n\n def __init__(self, config: Optional[Dict[str,
Any]] = None) -> None:\n super().__init__(source_name=self.SOURCE_NAME, base_url=self.BASE_URL, config=config)\n\n def
_configure_fetch_strategy(self) -> FetchStrategy:\n # Use CURL_CFFI with chrome120 for better reliability against bot
detection\n return FetchStrategy(primary_engine=BrowserEngine.CURL_CFFI, enable_js=True, stealth_mode=\"camouflage\",
timeout=45)\n\n async def make_request(self, method: str, url: str, **kwargs: Any) -> Any:\n
kwargs.setdefault(\"impersonate\", \"chrome120\")\n return await super().make_request(method, url, **kwargs)\n\n def
_get_headers(self) -> Dict[str, str]:\n return self._get_browser_headers(host=\"www.boylesports.com\",
referer=\"https://www.google.com/\")\n\n async def _fetch_data(self, date: str) -> Optional[Dict[str, Any]]:\n url =
\"/sports/horse-racing\"\n resp = await self.make_request(\"GET\", url, headers=self._get_headers())\n if not resp or not
resp.text:\n if resp: self.logger.warning(\"Unexpected status\", status=resp.status, url=url)\n return None\n
self._save_debug_snapshot(resp.text, f\"boylesports_index_{date}\")\n return {\"pages\": [{\"url\": url, \"html\":
resp.text}], \"date\": date}\n\n def _parse_races(self, raw_data: Any) -> List[Race]:\n if not raw_data or not
raw_data.get(\"pages\"): return []\n try: race_date = datetime.strptime(raw_data[\"date\"], \"%Y-%m-%d\").date()\n except
Exception: race_date = datetime.now(EASTERN).date()\n item = raw_data[\"pages\"][0]\n parser = HTMLParser(item.get(\"html\",
\"\"))\n races: List[Race] = []\n meeting_groups = parser.css('.meeting-group') or parser.css('.race-meeting') or
parser.css('div[class*=\"meeting\"]')\n for meeting in meeting_groups:\n tnn = meeting.css_first('.meeting-name') or
meeting.css_first('h2') or meeting.css_first('.title')\n if not tnn: continue\n trw = clean_text(tnn.text())\n track_name =
normalize_venue_name(trw)\n if not track_name: continue\n m_harness = any(kw in trw.lower() for kw in ['harness', 'trot',
'pace', 'standardbred'])\n is_grey = any(kw in trw.lower() for kw in ['greyhound', 'dog'])\n race_nodes =
meeting.css('.race-time-row') or meeting.css('.race-details') or meeting.css('a[href*=\"/race/\"]')\n for i, rn in
enumerate(race_nodes):\n txt = clean_text(rn.text())\n r_harness = m_harness or any(kw in txt.lower() for kw in ['trot',
'pace', 'attele', 'mounted'])\n tm = re.search(r'(\\d{1,2}:\\d{2})', txt)\n if not tm: continue\n fm =
re.search(r'\\((\\d+)\\s+runners\\)', txt, re.I)\n fs = int(fm.group(1)) if fm else 0\n dm =
re.search(r'(\\d+(?:\\.\\d+)?\\s*[kmf]|1\\s*mile)', txt, re.I)\n dist = dm.group(1) if dm else None\n try: st =
datetime.combine(race_date, datetime.strptime(tm.group(1), \"%H:%M\").time())\n except Exception: continue\n runners =
[Runner(number=j+1, name=f\"Runner {j+1}\", scratched=False, odds={}) for j in range(fs)]\n disc = \"Harness\" if r_harness
else \"Greyhound\" if is_grey else \"Thoroughbred\"\n ab = []\n if 'superfecta' in txt.lower(): ab.append('Superfecta')\n elif
r_harness or ' (us)' in trw.lower():\n if fs >= 6: ab.append('Superfecta')\n
races.append(Race(id=f\"boyle_{track_name.lower().replace(' ', '')}_{st:%Y%m%d_%H%M}\", venue=track_name, race_number=i + 1,
start_time=st, runners=runners, distance=dist, source=self.source_name, discipline=disc, available_bets=ab))\n return
races\n",
"name": "BoyleSportsAdapter"
},
{
"type": "miscellaneous",
"content": "\n\n# ----------------------------------------\n# SportingLifeAdapter\n#
----------------------------------------\n"
},
{
"type": "class",
"content": "class SportingLifeAdapter(JSONParsingMixin, BrowserHeadersMixin, DebugMixin, RacePageFetcherMixin,
BaseAdapterV3):\n SOURCE_NAME: ClassVar[str] = \"SportingLife\"\n BASE_URL: ClassVar[str] =
\"https://www.sportinglife.com\"\n\n def __init__(self, config: Optional[Dict[str, Any]] = None) -> None:\n
super().__init__(source_name=self.SOURCE_NAME, base_url=self.BASE_URL, config=config)\n\n def _configure_fetch_strategy(self)
-> FetchStrategy:\n return FetchStrategy(primary_engine=BrowserEngine.HTTPX, enable_js=False, stealth_mode=\"camouflage\",
timeout=30)\n\n def _get_headers(self) -> Dict[str, str]:\n return self._get_browser_headers(host=\"www.sportinglife.com\",
referer=\"https://www.sportinglife.com/racing/racecards\")\n\n async def _fetch_data(self, date: str) -> Optional[Dict[str,
Any]]:\n index_url = f\"/racing/racecards/{date}/\" if date else \"/racing/racecards/\"\n resp = await
self.make_request(\"GET\", index_url, headers=self._get_headers(), follow_redirects=True)\n if not resp or not resp.text:\n if
resp: self.logger.warning(\"Unexpected status\", status=resp.status, url=index_url)\n raise AdapterHttpError(self.source_name,
getattr(resp, 'status', 500), index_url)\n self._save_debug_snapshot(resp.text, f\"sportinglife_index_{date}\")\n parser =
HTMLParser(resp.text)\n metadata = self._extract_race_metadata(parser, date)\n if not metadata:\n self.logger.warning(\"No
metadata found\", context=\"SportingLife Index Parsing\", url=index_url)\n return None\n pages = await
self._fetch_race_pages_concurrent(metadata, self._get_headers(), semaphore_limit=8)\n return {\"pages\": pages, \"date\":
date}\n\n def _extract_race_metadata(self, parser: HTMLParser, date_str: str) -> List[Dict[str, Any]]:\n meta: List[Dict[str,
Any]] = []\n data = self._parse_json_from_script(parser, \"script#__NEXT_DATA__\", context=\"SportingLife Index\")\n\n try:\n
```

```
target_date = datetime.strptime(date_str, \"%Y-%m-%d\").date()\n except Exception:\n target_date =
datetime.now(EASTERN).date()\n\n site_tz = ZoneInfo(\"Europe/London\")\n now_site = datetime.now(site_tz)\n\n if data:\n for
meeting in data.get(\"props\", {}).get(\"pageProps\", {}).get(\"meetings\", []):\n # Broaden window to capture multiple races
(Memory Directive Fix)\n races = meeting.get(\"races\", [])\n for i, race in enumerate(races):\n r_time_str =
race.get(\"time\") # Usually HH:MM\n if r_time_str:\n try:\n rt = datetime.strptime(r_time_str, \"%H:%M\").replace(\n
year=target_date.year, month=target_date.month, day=target_date.day, tzinfo=site_tz\n )\n diff = (rt -
now_site).total_seconds() / 60\n if not (-45 < diff <= 1080):\n continue\n\n if url := race.get(\"racecard_url\"):\n
meta.append({\"url\": url, \"race_number\": i + 1})\n except Exception: pass\n if not meta:\n meetings =
parser.css('section[class^=\"MeetingSummary\"]') or parser.css(\".meeting-summary\")\n for meeting in meetings:\n # In HTML
fallback, just take the first upcoming link we find\n for link in meeting.css('a[href*=\"/racecard/\"]'):\n if url :=
link.attributes.get(\"href\"):\n # Try to see if time is in link text\n txt = node_text(link)\n if
re.match(r\"\\d{1,2}:\\d{2}\", txt):\n try:\n rt = datetime.strptime(txt, \"%H:%M\").replace(\n year=target_date.year,
month=target_date.month, day=target_date.day, tzinfo=site_tz\n )\n # Skip if in past (Today only)\n if target_date ==
now_site.date() and rt < now_site - timedelta(minutes=5):\n continue\n except Exception: pass\n\n meta.append({\"url\": url,
\"race_number\": 1})\n break\n return meta\n\n def _parse_races(self, raw_data: Any) -> List[Race]:\n if not raw_data or not
raw_data.get(\"pages\"): return []\n try: race_date = datetime.strptime(raw_data[\"date\"], \"%Y-%m-%d\").date()\n except
Exception: return []\n races: List[Race] = []\n for item in raw_data[\"pages\"]:\n html_content = item.get(\"html\")\n if not
html_content: continue\n try:\n parser = HTMLParser(html_content)\n race = self._parse_from_next_data(parser, race_date,
item.get(\"race_number\"), html_content)\n if not race: race = self._parse_from_html(parser, race_date,
item.get(\"race_number\"), html_content)\n if race: races.append(race)\n except Exception: pass\n return races\n\n def
_parse_from_next_data(self, parser: HTMLParser, race_date: date, race_number_fallback: Optional[int], html_content: str) ->
Optional[Race]:\n data = self._parse_json_from_script(parser, \"script#__NEXT_DATA__\", context=\"SportingLife Race\")\n if
not data: return None\n race_info = data.get(\"props\", {}).get(\"pageProps\", {}).get(\"race\")\n if not race_info: return
None\n summary = race_info.get(\"race_summary\") or {}\n track_name = normalize_venue_name(race_info.get(\"meeting_name\") or
summary.get(\"course_name\") or \"Unknown\")\n rt = race_info.get(\"time\") or summary.get(\"time\") or
race_info.get(\"off_time\") or race_info.get(\"start_time\")\n if not rt:\n def f(o):\n if isinstance(o, str) and
re.match(r\"^\\d{1,2}:\\d{2}$\", o): return o\n if isinstance(o, dict):\n for v in o.values():\n if t := f(v): return t\n if
isinstance(o, list):\n for v in o:\n if t := f(v): return t\n return None\n rt = f(race_info)\n if not rt: return None\n try:
start_time = datetime.combine(race_date, datetime.strptime(rt, \"%H:%M\").time())\n except Exception: return None\n runners =
[]\n for rd in (race_info.get(\"runners\") or race_info.get(\"rides\") or []):\n name = clean_text(rd.get(\"horse_name\") or
rd.get(\"horse\", {}).get(\"name\", \"\"))\n if not name: continue\n num = rd.get(\"saddle_cloth_number\") or
rd.get(\"cloth_number\") or 0\n wo = parse_odds_to_decimal(rd.get(\"betting\", {}).get(\"current_odds\") or
rd.get(\"betting\", {}).get(\"current_price\") or rd.get(\"forecast_price\") or rd.get(\"forecast_odds\") or
rd.get(\"betting_forecast_price\") or rd.get(\"odds\") or rd.get(\"bookmakerOdds\") or \"\")\n odds_data = {}\n if ov :=
create_odds_data(self.source_name, wo): odds_data[self.source_name] = ov\n runners.append(Runner(number=num, name=name,
scratched=rd.get(\"is_non_runner\") or rd.get(\"ride_status\") == \"NON_RUNNER\", odds=odds_data, win_odds=wo))\n if not
runners: return None\n return Race(id=generate_race_id(\"sl\", track_name or \"Unknown\", start_time,
race_info.get(\"race_number\") or race_number_fallback or 1), venue=track_name or \"Unknown\",
race_number=race_info.get(\"race_number\") or race_number_fallback or 1, start_time=start_time, runners=runners,
distance=summary.get(\"distance\") or race_info.get(\"distance\"), source=self.source_name, discipline=\"Thoroughbred\",
available_bets=scrape_available_bets(html_content))\n\n def _parse_from_html(self, parser: HTMLParser, race_date: date,
race_number_fallback: Optional[int], html_content: str) -> Optional[Race]:\n h1 =
parser.css_first('h1[class*=\"RacingRacecardHeader__Title\"]')\n if not h1: return None\n ht = clean_text(h1.text())\n if not
ht: return None\n parts = ht.split()\n if not parts: return None\n try: start_time = datetime.combine(race_date,
datetime.strptime(parts[0], \"%H:%M\").time())\n except Exception: return None\n track_name = normalize_venue_name(\"
\".join(parts[1:]))\n runners = []\n for row in parser.css('div[class*=\"RunnerCard\"]'):\n try:\n nn =
row.css_first('a[href*=\"/racing/profiles/horse/\"]')\n if not nn: continue\n name =
clean_text(nn.text()).splitlines()[0].strip()\n num_node = row.css_first('span[class*=\"SaddleCloth__Number\"]')\n number =
int(\"\".join(filter(str.isdigit, clean_text(num_node.text())))) if num_node else 0\n on =
row.css_first('span[class*=\"Odds__Price\"]')\n wo = parse_odds_to_decimal(clean_text(on.text()) if on else \"\")\n\n #
Advanced heuristic fallback\n if wo is None:\n wo = SmartOddsExtractor.extract_from_node(row)\n\n od = {}\n if ov :=
create_odds_data(self.source_name, wo): od[self.source_name] = ov\n runners.append(Runner(number=number, name=name, odds=od,
win_odds=wo))\n except Exception: continue\n if not runners: return None\n dn =
parser.css_first('span[class*=\"RacecardHeader__Distance\"]') or parser.css_first(\".race-distance\")\n return
Race(id=generate_race_id(\"sl\", track_name or \"Unknown\", start_time, race_number_fallback or 1), venue=track_name or
\"Unknown\", race_number=race_number_fallback or 1, start_time=start_time, runners=runners, distance=clean_text(dn.text()) if
dn else None, source=self.source_name, available_bets=scrape_available_bets(html_content))\n",
"name": "SportingLifeAdapter"
},
{
"type": "miscellaneous",
"content": "\n# ----------------------------------------\n# SkySportsAdapter\n# ----------------------------------------\n"
},
{
"type": "class",
"content": "class SkySportsAdapter(JSONParsingMixin, BrowserHeadersMixin, DebugMixin, RacePageFetcherMixin, BaseAdapterV3):\n
SOURCE_NAME: ClassVar[str] = \"SkySports\"\n BASE_URL: ClassVar[str] = \"https://www.skysports.com\"\n\n def __init__(self,
config: Optional[Dict[str, Any]] = None) -> None:\n super().__init__(source_name=self.SOURCE_NAME, base_url=self.BASE_URL,
config=config)\n\n def _configure_fetch_strategy(self) -> FetchStrategy:\n return
FetchStrategy(primary_engine=BrowserEngine.HTTPX, enable_js=False, stealth_mode=\"fast\", timeout=30)\n\n def
_get_headers(self) -> Dict[str, str]:\n return self._get_browser_headers(host=\"www.skysports.com\",
referer=\"https://www.skysports.com/racing\")\n\n async def _fetch_data(self, date: str) -> Optional[Dict[str, Any]]:\n dt =
datetime.strptime(date, \"%Y-%m-%d\")\n index_url = f\"/racing/racecards/{dt.strftime('%d-%m-%Y')}\"\n resp = await
self.make_request(\"GET\", index_url, headers=self._get_headers())\n if not resp or not resp.text:\n if resp:
self.logger.warning(\"Unexpected status\", status=resp.status, url=index_url)\n raise AdapterHttpError(self.source_name,
getattr(resp, 'status', 500), index_url)\n self._save_debug_snapshot(resp.text, f\"skysports_index_{date}\")\n parser =
HTMLParser(resp.text)\n metadata = []\n\n try:\n target_date = datetime.strptime(date, \"%Y-%m-%d\").date()\n except
Exception:\n target_date = datetime.now(EASTERN).date()\n\n site_tz = ZoneInfo(\"Europe/London\")\n now_site =
datetime.now(site_tz)\n\n meetings = parser.css(\".sdc-site-concertina-block\") or parser.css(\".page-details__section\") or
parser.css(\".racing-meetings__meeting\")\n for meeting in meetings:\n hn =
meeting.css_first(\".sdc-site-concertina-block__title\") or meeting.css_first(\".racing-meetings__meeting-title\")\n if not
hn: continue\n vr = clean_text(hn.text()) or \"\"\n if \"ABD:\" in vr: continue\n\n # Updated Sky Sports event discovery
logic\n events = meeting.css(\".sdc-site-racing-meetings__event\") or meeting.css(\".racing-meetings__event\")\n if events:\n
for i, event in enumerate(events):\n tn = event.css_first(\".sdc-site-racing-meetings__event-time\") or
event.css_first(\".racing-meetings__event-time\")\n ln = event.css_first(\".sdc-site-racing-meetings__event-link\") or
```

```
event.css_first(\".racing-meetings__event-link\")\n if tn and ln:\n txt, h = clean_text(tn.text()),
ln.attributes.get(\"href\")\n if h and re.match(r\"\\d{1,2}:\\d{2}\", txt):\n try:\n rt = datetime.strptime(txt,
\"%H:%M\").replace(\n year=target_date.year, month=target_date.month, day=target_date.day, tzinfo=site_tz\n )\n diff = (rt -
now_site).total_seconds() / 60\n if not (-45 < diff <= 1080):\n continue\n metadata.append({\"url\": h, \"venue_raw\": vr,
\"race_number\": i + 1})\n except Exception: pass\n else:\n # Fallback to older anchor-based discovery\n for i, link in
enumerate(meeting.css('a[href*=\"/racecards/\"]')):\n if h := link.attributes.get(\"href\"):\n txt = node_text(link)\n if
re.match(r\"\\d{1,2}:\\d{2}\", txt):\n try:\n rt = datetime.strptime(txt, \"%H:%M\").replace(\n year=target_date.year,
month=target_date.month, day=target_date.day, tzinfo=site_tz\n )\n diff = (rt - now_site).total_seconds() / 60\n if not (-45 <
diff <= 1080):\n continue\n metadata.append({\"url\": h, \"venue_raw\": vr, \"race_number\": i + 1})\n except Exception:
pass\n\n if not metadata:\n self.logger.warning(\"No metadata found\", context=\"SkySports Index Parsing\", url=index_url)\n
return None\n pages = await self._fetch_race_pages_concurrent(metadata, self._get_headers(), semaphore_limit=10)\n return
{\"pages\": pages, \"date\": date}\n\n def _parse_races(self, raw_data: Any) -> List[Race]:\n if not raw_data or not
raw_data.get(\"pages\"): return []\n try: race_date = datetime.strptime(raw_data.get(\"date\", \"\"), \"%Y-%m-%d\").date()\n
except Exception: race_date = datetime.now(EASTERN).date()\n races: List[Race] = []\n for item in raw_data[\"pages\"]:\n
html_content = item.get(\"html\")\n if not html_content: continue\n parser = HTMLParser(html_content)\n h =
parser.css_first(\".sdc-site-racing-header__name\")\n if not h: continue\n ht = clean_text(h.text()) or \"\"\n m =
re.match(r\"(\\d{1,2}:\\d{2})\\s+(.+)\", ht)\n if not m:\n tn, cn = parser.css_first(\".sdc-site-racing-header__time\"),
parser.css_first(\".sdc-site-racing-header__course\")\n if tn and cn: rts, tnr = clean_text(tn.text()) or \"\",
clean_text(cn.text()) or \"\"\n else: continue\n else: rts, tnr = m.group(1), m.group(2)\n track_name =
normalize_venue_name(tnr)\n if not track_name: continue\n try: start_time = datetime.combine(race_date, datetime.strptime(rts,
\"%H:%M\").time())\n except Exception: continue\n dist = None\n for d in
parser.css(\".sdc-site-racing-header__detail-item\"):\n dt = clean_text(d.text()) or \"\"\n if \"Distance:\" in dt: dist =
dt.replace(\"Distance:\", \"\").strip(); break\n runners = []\n for i, node in
enumerate(parser.css(\".sdc-site-racing-card__item\")):\n nn = node.css_first(\".sdc-site-racing-card__name a\")\n if not nn:
continue\n name = clean_text(nn.text())\n if not name: continue\n nnode = node.css_first(\".sdc-site-racing-card__number
strong\")\n number = i + 1\n if nnode:\n nt = clean_text(nnode.text())\n if nt:\n try: number = int(nt)\n except Exception:
pass\n onode = node.css_first(\".sdc-site-racing-card__betting-odds\")\n wo = parse_odds_to_decimal(clean_text(onode.text())
if onode else \"\")\n\n # Advanced heuristic fallback\n if wo is None:\n wo = SmartOddsExtractor.extract_from_node(node)\n\n
ntxt = clean_text(node.text()) or \"\"\n scratched = \"NR\" in ntxt or \"Non-runner\" in ntxt\n od = {}\n if ov :=
create_odds_data(self.source_name, wo): od[self.source_name] = ov\n runners.append(Runner(number=number, name=name,
scratched=scratched, odds=od, win_odds=wo))\n if not runners: continue\n disc = detect_discipline(html_content)\n ab =
scrape_available_bets(html_content)\n if not ab and (disc == \"Harness\" or \"(us)\" in tnr.lower()) and len([r for r in
runners if not r.scratched]) >= 6: ab.append(\"Superfecta\")\n races.append(Race(id=generate_race_id(\"sky\", track_name,
start_time, item.get(\"race_number\", 0), disc), venue=track_name, race_number=item.get(\"race_number\", 0),
start_time=start_time, runners=runners, distance=dist, discipline=disc, source=self.source_name, available_bets=ab))\n return
races\n",
"name": "SkySportsAdapter"
}
]
}
```