```
{
"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\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u25
CRITICAL: Fix for Playwright + PyInstaller + Windows\n# Must be at the very top, before any other imports\n#
\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u25
},
{
"type": "import",
"content": "import sys\n"
},
{
"type": "import",
"content": "import platform\n"
},
{
"type": "import",
"content": "import os\n"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "unknown",
"content": "if platform.system() == 'Windows' and getattr(sys, 'frozen', False):\n # Running as frozen EXE on Windows\n import
asyncio\n try:\n # Check if Playwright is likely to be available\n playwright_path =
os.path.expanduser(\"~\\\\AppData\\\\Local\\\\ms-playwright\")\n has_playwright = os.path.exists(playwright_path)\n\n # GPT5
Fix: Default to Selector loop if Playwright is missing to satisfy curl_cffi\n if
os.getenv(\"FORTUNA_USE_SELECTOR_EVENT_LOOP\") == \"1\" or not has_playwright:\n
asyncio.set_event_loop_policy(asyncio.WindowsSelectorEventLoopPolicy())\n else:\n
asyncio.set_event_loop_policy(asyncio.WindowsProactorEventLoopPolicy())\n except AttributeError:\n pass\n"
},
{
"type": "miscellaneous",
"content": "#
\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u2501\u25
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 weakref\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\": {\"simply_success_trust_min\": 0.25, \"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\n # Deprecation bridge for trustworthy_ratio_min (BUG-2)\n analysis_cfg = config.get(\"analysis\",
{})\n legacy_val = analysis_cfg.get(\"trustworthy_ratio_min\")\n if legacy_val is not None:\n
structlog.get_logger().warning(\"config key analysis.trustworthy_ratio_min is deprecated; use
analysis.simply_success_trust_min\")\n if \"simply_success_trust_min\" not in toml_data.get(\"analysis\", {}):\n
analysis_cfg[\"simply_success_trust_min\"] = legacy_val\n\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(\"simply_success_trust_min\", 0.25)\n print_func(f\" \ud83d\udee1\ufe0f
Odds Hygiene: >{int(trust_min*100)}% trust ratio required\")\n\n # Reports\n reports = []\n if
get_writable_path(\"summary_grid.txt\").exists(): reports.append(\"Summary\")\n if
get_writable_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 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 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 =
get_writable_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": "import",
"content": "from fortuna_utils import (\n EASTERN, DATE_FORMAT, DATE_FORMAT_OLD, MAX_VALID_ODDS, MIN_VALID_ODDS,\n
DEFAULT_ODDS_FALLBACK, COMMON_PLACEHOLDERS,\n VENUE_MAP, RACING_KEYWORDS, BET_TYPE_KEYWORDS, DISCIPLINE_KEYWORDS,\n
clean_text, node_text, get_canonical_venue, normalize_venue_name,\n parse_odds_to_decimal, SmartOddsExtractor,
is_placeholder_odds,\n is_valid_odds, scrape_available_bets, detect_discipline,\n now_eastern, to_eastern, ensure_eastern,
get_places_paid,\n parse_date_string, to_storage_format, from_storage_format\n)\n"
},
{
"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] = {\n # \"Equibase\", # Decommissioned 2026-02: persistent bot blocking, 0%
30-day success\n \"TwinSpires\", \"RacingPostB2B\", \"StandardbredCanada\", \"AtTheRaces\", \"NYRABets\",\n
\"Official_DelMar\", \"Official_GulfstreamPark\", \"Official_TampaBayDowns\",\n \"Official_OaklawnPark\",
\"Official_SantaAnita\", \"Official_MonmouthPark\",\n \"Official_TheMeadowlands\", \"Official_YonkersRaceway\",
\"Official_Woodbine\",\n \"Official_LaurelPark\", \"Official_Pimlico\", \"Official_FairGrounds\",\n \"Official_ParxRacing\",
\"Official_PennNational\", \"Official_CharlesTown\",\n \"Official_Mountaineer\", \"Official_TurfParadise\",
\"Official_EmeraldDowns\",\n \"Official_LoneStarPark\", \"Official_SamHouston\", \"Official_RemingtonPark\",\n
\"Official_SunlandPark\", \"Official_ZiaPark\", \"Official_FingerLakes\",\n \"Official_Thistledown\",
\"Official_MahoningValley\", \"Official_BelterraPark\",\n \"Official_SaratogaHarness\", \"Official_HoosierPark\",
\"Official_NorthfieldPark\",\n \"Official_SciotoDowns\", \"Official_FortErie\", \"Official_Hastings\"\n}\n"
},
{
"type": "unknown",
"content": "INT_DISCOVERY_ADAPTERS: Final[set] = {\n \"TAB\", \"BetfairDataScientist\", \"HKJC\", \"JRA\",
\"Official_JRAJapan\",\n \"Official_Ascot\", \"Official_Cheltenham\", \"Official_Flemington\"\n}\n"
},
{
"type": "unknown",
"content": "OFFICIAL_DISCOVERY_ADAPTERS: Final[set] = {\n \"Official_DelMar\", \"Official_GulfstreamPark\",
\"Official_TampaBayDowns\",\n \"Official_OaklawnPark\", \"Official_SantaAnita\", \"Official_MonmouthPark\",\n
\"Official_Woodbine\", \"Official_TheMeadowlands\", \"Official_YonkersRaceway\",\n \"Official_JRAJapan\",
\"Official_LaurelPark\", \"Official_Pimlico\",\n \"Official_FairGrounds\", \"Official_ParxRacing\",
\"Official_PennNational\",\n \"Official_CharlesTown\", \"Official_Mountaineer\", \"Official_TurfParadise\",\n
\"Official_EmeraldDowns\", \"Official_LoneStarPark\", \"Official_SamHouston\",\n \"Official_RemingtonPark\",
\"Official_SunlandPark\", \"Official_ZiaPark\",\n \"Official_FingerLakes\", \"Official_Thistledown\",
\"Official_MahoningValley\",\n \"Official_BelterraPark\", \"Official_SaratogaHarness\", \"Official_HoosierPark\",\n
\"Official_NorthfieldPark\", \"Official_SciotoDowns\", \"Official_FortErie\",\n \"Official_Hastings\", \"Official_Ascot\",
\"Official_Cheltenham\", \"Official_Flemington\"\n}\n"
},
{
"type": "unknown",
"content": "GLOBAL_DISCOVERY_ADAPTERS: Final[set] = {\n \"SkyRacingWorld\", \"AtTheRaces\", \"AtTheRacesGreyhound\",
\"RacingPost\",\n \"Oddschecker\", \"Timeform\", \"SportingLife\", \"SkySports\",\n \"RacingAndSports\", \"HKJC\", \"JRA\"\n}
| OFFICIAL_DISCOVERY_ADAPTERS\n"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "unknown",
"content": "USA_RESULTS_ADAPTERS: Final[set] = {\n # \"EquibaseResults\", # Decommissioned 2026-02: persistent bot blocking,
0% 30-day success\n \"SportingLifeResults\",\n \"StandardbredCanadaResults\",\n \"RacingPostUSAResults\",\n \"DRFResults\", #
Reactivated for testing (Uses HTTPX engine)\n \"NYRABetsResults\",\n}\n"
},
{
"type": "unknown",
"content": "INT_RESULTS_ADAPTERS: Final[set] = {\n \"RacingPostResults\", \"RacingPostTote\", \"AtTheRacesResults\",\n
\"AtTheRacesGreyhoundResults\", \"SportingLifeResults\", \"SkySportsResults\",\n \"RacingAndSportsResults\",
\"TimeformResults\"\n}\n"
},
{
"type": "miscellaneous",
"content": "\n# Quality-based Partitioning (JB/Council Strategy)\n"
},
{
"type": "unknown",
"content": "SOLID_DISCOVERY_ADAPTERS: Final[set] = {\"TwinSpires\", \"SkyRacingWorld\", \"RacingPost\"}\n"
},
{
"type": "unknown",
"content": "SOLID_RESULTS_ADAPTERS: Final[set] = {\n \"StandardbredCanadaResults\",\n \"RacingPostResults\",\n
\"SportingLifeResults\",\n \"AtTheRacesGreyhoundResults\",\n \"TimeformResults\",\n \"SkySportsResults\",\n
\"NYRABetsResults\",\n}\n"
```

```
},
{
"type": "miscellaneous",
"content": "\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/133.0.0.0 Safari/537.36\"\n)\n"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "unknown",
"content": "CHROME_SEC_CH_UA: Final[str] = (\n '\"Google Chrome\";v=\"133\", \"Chromium\";v=\"133\",
\"Not.A/Brand\";v=\"24\"'\n)\n"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "unknown",
"content": "MOBILE_USER_AGENT: Final[str] = (\n \"Mozilla/5.0 (iPhone; CPU iPhone OS 18_3 like Mac OS X) AppleWebKit/605.1.15
\"\n \"(KHTML, like Gecko) Version/18.3 Mobile/15E148 Safari/604.1\"\n)\n"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "unknown",
"content": "MOBILE_SEC_CH_UA: Final[str] = (\n '\"Safari\";v=\"18\", \"Mobile\";v=\"18.3\"'\n)\n"
},
{
"type": "miscellaneous",
"content": "\n# Bet type keywords mapping (lowercase key -> display name)\n\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": "function",
"content": "def create_odds_data(source: str, win_odds: Optional[float]) -> Optional[OddsData]:\n \"\"\"Helper to create an
OddsData object for a given source and win odds.\"\"\"\n if win_odds is None:\n return None\n try:\n return
OddsData(source=source, win=Decimal(str(win_odds)))\n except Exception:\n return None\n",
"name": "create_odds_data"
},
{
"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 odds_source: Optional[str] = Field(None, description=\"How win_odds was
obtained: 'extracted', 'smart_extractor', 'default', or the source adapter name\")\n trainer: Optional[str] = None\n jockey:
Optional[str] = None\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
race_type: Optional[str] = None\n is_handicap: Optional[bool] = None\n\n @field_validator(\"venue\", mode=\"after\")\n
@classmethod\n def normalize_venue(cls, v: str) -> str:\n \"\"\"Ensure venue is normalized through VENUE_MAP.\"\"\"\n if not v
or v == \"Unknown\":\n return v\n normalized = normalize_venue_name(v)\n return normalized if normalized != \"Unknown\" else
v\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\n source: str\n discipline: str
= \"Thoroughbred\"\n surface: 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 _safe_int(text: str, default: int = 0) -> int:\n \"\"\"Extract leading digits from text, return *default* on
failure.\"\"\"\n if not text: return default\n cleaned = re.sub(r\"\\D\", \"\", str(text))\n try:\n return int(cleaned) if
cleaned else default\n except ValueError:\n return default\n",
"name": "_safe_int"
},
{
"type": "miscellaneous",
"content": "\n\n\n\n\n\n\n\n\n\n\n"
},
{
"type": "function",
"content": "def generate_race_id(\n prefix: str,\n venue: str,\n start_time: datetime,\n race_number: int,\n discipline:
Optional[str] = None,\n) -> str:\n venue_slug = get_canonical_venue(venue)\n\n # Defense: warn on suspiciously long venue
slugs (likely race title contamination)\n if len(venue_slug) > 25:\n _log = structlog.get_logger(\"generate_race_id\")\n
_log.warning(\n \"suspiciously_long_venue_slug\",\n raw_venue=venue,\n slug=venue_slug,\n prefix=prefix,\n )\n # Attempt
recovery: try first word only\n first_word = venue.split()[0] if venue else venue\n recovered =
get_canonical_venue(first_word)\n if recovered != \"unknown\":\n venue_slug = recovered\n\n date_str =
start_time.strftime(DATE_FORMAT)\n time_str = start_time.strftime(\"%H%M\")\n\n dl = (discipline or
\"Thoroughbred\").lower()\n if \"harness\" in dl:\n disc_suffix = \"_h\"\n elif \"greyhound\" in dl:\n disc_suffix = \"_g\"\n
elif \"quarter\" in dl:\n disc_suffix = \"_q\"\n else:\n 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@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 _locks:
weakref.WeakKeyDictionary[asyncio.AbstractEventLoop, asyncio.Lock] = field(default_factory=weakref.WeakKeyDictionary,
init=False)\n _lock_sentinel: ClassVar[threading.Lock] = threading.Lock()\n\n def __post_init__(self):\n self._tokens =
self.requests_per_second\n\n def _get_lock(self) -> asyncio.Lock:\n try:\n loop = asyncio.get_running_loop()\n except
RuntimeError:\n return asyncio.Lock()\n\n if loop not in self._locks:\n with self._lock_sentinel:\n if loop not in
self._locks:\n self._locks[loop] = asyncio.Lock()\n return self._locks[loop]\n\n async def acquire(self) -> None:\n lock =
self._get_lock()\n\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 GlobalResourceManager:\n \"\"\"Manages shared resources like HTTP clients and semaphores.\"\"\"\n _clients:
ClassVar[weakref.WeakKeyDictionary[asyncio.AbstractEventLoop, httpx.AsyncClient]] = weakref.WeakKeyDictionary()\n _semaphores:
ClassVar[weakref.WeakKeyDictionary[asyncio.AbstractEventLoop, asyncio.Semaphore]] = weakref.WeakKeyDictionary()\n _locks:
ClassVar[weakref.WeakKeyDictionary[asyncio.AbstractEventLoop, asyncio.Lock]] = weakref.WeakKeyDictionary()\n _host_limiters:
ClassVar[Dict[str, RateLimiter]] = {}\n _lock_initialized: ClassVar[threading.Lock] = threading.Lock()\n\n @classmethod\n
async def get_host_limiter(cls, host: str) -> RateLimiter:\n \"\"\"Returns a per-host rate limiter.\"\"\"\n if host not in
cls._host_limiters:\n with cls._lock_initialized:\n if host not in cls._host_limiters:\n # Default to 2 requests per second
per host to avoid 429s (Fix 13)\n limit = 2.0\n if \"racingpost\" in host: limit = 1.5 # Extra conservative for RP\n
cls._host_limiters[host] = RateLimiter(requests_per_second=limit)\n return cls._host_limiters[host]\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 for the current event loop.\n If timeout is provided and differs from current client, the client is
recreated.\n \"\"\"\n loop = asyncio.get_running_loop()\n lock = await cls._get_lock()\n async with lock:\n client =
cls._clients.get(loop)\n if client is not None:\n # Guard against None in timeout comparison\n current_timeout =
getattr(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 client.aclose()\n except Exception:\n pass\n client = None\n\n if client is None:\n
use_timeout = timeout or DEFAULT_REQUEST_TIMEOUT\n 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 cls._clients[loop] = client\n return client\n\n
@classmethod\n def get_global_semaphore(cls) -> asyncio.Semaphore:\n \"\"\"Returns a shared semaphore for the current event
loop.\"\"\"\n try:\n loop = asyncio.get_running_loop()\n except RuntimeError:\n # If called outside a loop, we create a
temporary semaphore\n return asyncio.Semaphore(DEFAULT_CONCURRENT_REQUESTS * 2)\n\n if loop not in cls._semaphores:\n with
cls._lock_initialized:\n if loop not in cls._semaphores:\n cls._semaphores[loop] =
asyncio.Semaphore(DEFAULT_CONCURRENT_REQUESTS * 2)\n return cls._semaphores[loop]\n\n @classmethod\n async def cleanup(cls):\n
\"\"\"Closes all clients for all event loops.\"\"\"\n clients_to_close = []\n with cls._lock_initialized:\n clients_to_close =
list(cls._clients.values())\n cls._clients.clear()\n cls._semaphores.clear()\n cls._locks.clear()\n\n for client in
```

```
clients_to_close:\n try:\n await client.aclose()\n except (AttributeError, RuntimeError):\n pass\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 = False\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._health_lock
= asyncio.Lock()\n self._request_count = 0\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 self._sessions: Dict[BrowserEngine, Any] = {}\n self._session_lock = asyncio.Lock()\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 _get_persistent_session(self, engine: BrowserEngine) ->
Any:\n \"\"\"Returns a persistent session for browser-based engines to avoid launch overhead (Fix 12).\"\"\"\n async with
self._session_lock:\n if engine not in self._sessions:\n if engine == BrowserEngine.CAMOUFOX:\n self._sessions[engine] =
AsyncStealthySession(headless=True)\n await self._sessions[engine].__aenter__()\n elif engine == BrowserEngine.PLAYWRIGHT:\n
self._sessions[engine] = AsyncDynamicSession(headless=True)\n await self._sessions[engine].__aenter__()\n return
self._sessions[engine]\n\n async def fetch(self, url: str, **kwargs: Any) -> Any:\n method = kwargs.pop(\"method\",
\"GET\").upper()\n kwargs.pop(\"url\", None)\n\n async with self._health_lock:\n self._request_count += 1\n if
self._request_count % 100 == 0:\n for engine in self._engine_health:\n self._engine_health[engine] = max(0.1,
self._engine_health[engine] * 0.995)\n\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 async with self._health_lock:\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 async with
self._health_lock:\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 # Copy headers before mutation to avoid leaking state across
requests\n headers = dict(kwargs.get(\"headers\", {}))\n # Merge - browserforge headers complement provided ones\n for k, v in
bf_headers.items():\n if k not in headers:\n headers[k] = v\n kwargs[\"headers\"] = 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\n # BUG-14: Impersonation fallback chain to handle unsupported versions\n
requested_impersonate = kwargs.get(\"impersonate\") or getattr(strategy, \"impersonate\", None) or \"chrome133\"\n
impersonate_chain = [requested_impersonate, \"chrome133\", \"chrome128\", \"chrome124\", \"chrome120\"]\n # Filter out
duplicates while preserving order\n impersonate_chain = list(dict.fromkeys(impersonate_chain))\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 last_err = None\n for imp_version in
impersonate_chain:\n try:\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=imp_version,\n **clean_kwargs\n )\n return UnifiedResponse(resp.text,
resp.status_code, resp.status_code, resp.url, resp.headers)\n except Exception as e:\n err_lower = str(e).lower()\n if
(\"impersonat\" in err_lower or \"supported\" in err_lower) and \"chrome\" in err_lower:\n self.logger.debug(\"curl_cffi
impersonation not supported, trying next\", version=imp_version)\n last_err = e\n continue\n raise\n\n raise last_err or
FetchError(f\"All curl_cffi impersonations failed for {url}\")\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 # BUG-1 Fix: Use persistent session to avoid launch
overhead\n s = await self._get_persistent_session(engine)\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 # BUG-1 Fix: Use persistent session to avoid launch overhead\n s = await
self._get_persistent_session(engine)\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 Persistent scrapling sessions are cleaned up here (Fix 12).\n \"\"\"\n async with
self._session_lock:\n for engine, session in self._sessions.items():\n try:\n await session.__aexit__(None, None, None)\n
except Exception as e:\n self.logger.warning(f\"failed_closing_persistent_session\", engine=engine.value, error=str(e))\n
self._sessions.clear()\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 _locks:
weakref.WeakKeyDictionary[asyncio.AbstractEventLoop, asyncio.Lock] = field(default_factory=weakref.WeakKeyDictionary,
init=False)\n _lock_sentinel: ClassVar[threading.Lock] = threading.Lock()\n\n def __post_init__(self):\n self._tokens =
self.requests_per_second\n\n def _get_lock(self) -> asyncio.Lock:\n try:\n loop = asyncio.get_running_loop()\n except
RuntimeError:\n return asyncio.Lock()\n\n if loop not in self._locks:\n with self._lock_sentinel:\n if loop not in
self._locks:\n self._locks[loop] = asyncio.Lock()\n return self._locks[loop]\n\n async def acquire(self) -> None:\n lock =
self._get_lock()\n\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, 0.001))\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 self.parse_warnings = 0\n self.parse_errors = 0\n\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\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
= None\n\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\n def
record_parse_warning(self) -> None:\n with self._lock:\n self.parse_warnings += 1\n\n def record_parse_error(self) -> None:\n
with self._lock:\n self.parse_errors += 1\n\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 \"parse_warnings\": self.parse_warnings,\n \"parse_errors\": self.parse_errors\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(node_text(script))\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(node_text(script)))\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 is_mobile = getattr(self, \"config\", {}).get(\"mobile\", False)\n ua =
MOBILE_USER_AGENT if is_mobile else CHROME_USER_AGENT\n sec_ua = MOBILE_SEC_CH_UA if is_mobile else CHROME_SEC_CH_UA\n mob =
\"?1\" if is_mobile else \"?0\"\n plat = '\"iOS\"' if is_mobile else '\"Windows\"'\n\n h = {\n **DEFAULT_BROWSER_HEADERS,\n
\"User-Agent\": ua,\n \"sec-ch-ua\": sec_ua,\n \"sec-ch-ua-mobile\": mob,\n \"sec-ch-ua-platform\": plat\n }\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 = get_writable_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 # Lowered threshold to 100 to avoid
unnecessary retries for small valid data files\n if resp and hasattr(resp, \"text\") and resp.text and len(resp.text) > 100:\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 # Default to False to ensure races with
partial odds data are analyzed\n PROVIDES_ODDS: ClassVar[bool] = False\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.headers: Dict[str, str] = {}\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 strategy.primary_engine in [BrowserEngine.PLAYWRIGHT, BrowserEngine.CAMOUFOX]:\n
if is_frozen():\n self.logger.info(\"Skipping browser-dependent adapter in monolith mode\")\n return []\n # FIX_06: Gracefully
skip if Playwright is required but missing (GHA check)\n try:\n import playwright\n except ImportError:\n
self.logger.warning(\"Playwright not installed, skipping browser-based adapter\", source=self.source_name)\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 # Propagate adapter capability flag to race
metadata\n for r in races:\n r.metadata[\"provides_odds\"] = self.PROVIDES_ODDS\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\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 else:\n # Clear invalid or missing odds to maintain
hygiene\n runner.win_odds = None\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 #
FIX_03: Duplicate race data detection (content fingerprinting)\n deduped_races = []\n fingerprints = {}\n for r in races:\n
active = [(run.name, str(run.win_odds)) for run in r.runners if not run.scratched]\n fp = (r.venue, frozenset(active))\n if fp
in fingerprints:\n fingerprints[fp] += 1\n if fingerprints[fp] >= 3:\n self.logger.warning(\"Duplicate race content detected
at venue, skipping\", venue=r.venue, race=r.race_number)\n continue\n else:\n fingerprints[fp] = 1\n
deduped_races.append(r)\n\n valid, warnings = DataValidationPipeline.validate_parsed_races(deduped_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\n # Apply host-based rate limiting to
prevent 429s (Fix 13)\n from urllib.parse import urlparse\n host = urlparse(full_url).netloc\n if host:\n limiter = await
GlobalResourceManager.get_host_limiter(host)\n await limiter.acquire()\n\n self.logger.debug(\"Requesting\", method=method,
url=full_url)\n\n # Merge adapter-level headers if defined\n if hasattr(self, 'headers') and self.headers:\n current_headers =
kwargs.get(\"headers\", {})\n # Passed headers take precedence over adapter defaults\n merged_headers = {**self.headers,
**current_headers}\n kwargs[\"headers\"] = merged_headers\n\n # Apply global concurrency limit\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 HKJCAdapter(JSONParsingMixin, BrowserHeadersMixin, DebugMixin, RacePageFetcherMixin, BaseAdapterV3):\n
\"\"\"\n Adapter for Hong Kong Jockey Club (HKJC).\n Extremely reliable data source for Hong Kong racing.\n \"\"\"\n
SOURCE_NAME: ClassVar[str] = \"HKJC\"\n BASE_URL: ClassVar[str] = \"https://racing.hkjc.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.HTTPX,\n enable_js=False,\n timeout=30\n )\n\n def _get_headers(self) -> Dict[str, str]:\n return
self._get_browser_headers(host=\"racing.hkjc.com\")\n\n async def _fetch_data(self, date: str) -> Optional[Dict[str, Any]]:\n
# date is YYMMDD, HKJC results/entries often use YYYY/MM/DD\n dt = parse_date_string(date)\n date_hk =
dt.strftime(\"%Y/%m/%d\")\n \n # Try RaceCard first (Discovery)\n url =
f\"/racing/information/English/racing/RaceCard.aspx?RaceDate={date_hk}\"\n resp = await self.make_request(\"GET\", url,
headers=self._get_headers())\n \n if not resp or not resp.text or \"Information will be released shortly\" in resp.text:\n #
Try Results page if RaceCard is not available (maybe it just finished)\n url =
f\"/racing/information/English/Racing/LocalResults.aspx?RaceDate={date_hk}\"\n resp = await self.make_request(\"GET\", url,
headers=self._get_headers())\n\n if not resp or not resp.text:\n return None\n\n self._save_debug_snapshot(resp.text,
f\"hkjc_index_{date}\")\n parser = HTMLParser(resp.text)\n \n # If still no info, try the general entries page\n if
\"Information will be released shortly\" in resp.text:\n entries_url = \"/racing/information/English/racing/Entries.aspx\"\n
resp = await self.make_request(\"GET\", entries_url, headers=self._get_headers())\n if not resp or not resp.text:\n return
None\n parser = HTMLParser(resp.text)\n\n # Find race links\n # HKJC uses specific icons or text for race numbers\n metadata =
[]\n # Case-insensitive attribute match for RaceNo (Fix 16)\n for a in parser.css(\"a\"):\n href = a.attributes.get(\"href\",
\"\")\n if \"RaceNo=\" in href or \"raceno=\" in href:\n metadata.append({\"url\": href})\n \n if not metadata:\n # Maybe it's
a single race page or all-races page\n if \"Race Card\" in resp.text:\n return {\"html\": resp.text, \"url\": url, \"date\":
date}\n return None\n\n # Fetch all races\n pages = await self._fetch_race_pages_concurrent(metadata, 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: return
[]\n races = []\n date_str = raw_data[\"date\"]\n try:\n race_date = parse_date_string(date_str).date()\n except Exception:\n
race_date = datetime.now(EASTERN).date()\n\n if \"pages\" in raw_data:\n for p in raw_data[\"pages\"]:\n if p and
p.get(\"html\"):\n race = self._parse_single_race(p[\"html\"], p.get(\"url\", \"\"), race_date)\n if race:
races.append(race)\n elif \"html\" in raw_data:\n race = self._parse_single_race(raw_data[\"html\"], raw_data.get(\"url\",
\"\"), race_date)\n if race: races.append(race)\n \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 # Venue is usually Sha Tin or Happy Valley\n
venue = \"Hong Kong\"\n if \"Sha Tin\" in html_content: venue = \"Sha Tin\"\n elif \"Happy Valley\" in html_content: venue =
\"Happy Valley\"\n \n # Race number\n race_num = 1\n num_match = re.search(r\"RaceNo=(\\d+)\", url)\n if num_match:\n race_num
= int(num_match.group(1))\n else:\n # Try to find in text \"Race 1\"\n txt_match = re.search(r\"Race\\s+(\\d+)\",
html_content, re.I)\n if txt_match: race_num = int(txt_match.group(1))\n\n # Runners\n runners = []\n # HKJC uses a table with
class 'performance'\n for row in parser.css(\"table.performance tr\"):\n cols = row.css(\"td\")\n if len(cols) < 5: continue\n
\n # Saddle cloth number\n try:\n num = int(clean_text(node_text(cols[0])))\n except Exception: continue\n \n # Horse Name\n
name_node = cols[2].css_first(\"a\")\n name = clean_text(node_text(name_node or cols[2]))\n if not name or name.upper() in
[\"HORSE\", \"NAME\"]: continue\n \n # Odds\n win_odds = None\n # HKJC odds are usually in a specific column or can be found
in text\n # For now, we'll use SmartOddsExtractor as HKJC layout is complex\n win_odds =
SmartOddsExtractor.extract_from_node(row)\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=num, odds=odds_data, win_odds=win_odds))\n \n if
not runners: return None\n \n # Start time - HKJC usually lists it\n start_time = datetime.combine(race_date,
datetime.min.time())\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(\"hkjc\", 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 discipline=\"Thoroughbred\"\n )\n",
"name": "HKJCAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialTrackAdapter(BaseAdapterV3):\n \"\"\"\n Adapter that verifies the availability of an official
racetrack website.\n Supports a '200 OK' health check as requested by JB.\n \"\"\"\n ADAPTER_TYPE = \"discovery\"\n
PROVIDES_ODDS = False\n\n def __init__(self, track_name: str, url: str, config: Optional[Dict[str, Any]] = None):\n
self.track_name = track_name\n self.official_url = url\n # Use a safe name for the source\n source =
f\"Official_{track_name.replace(' ', '').replace('/', '')}\"\n super().__init__(source_name=source, base_url=url,
config=config)\n\n def _configure_fetch_strategy(self) -> FetchStrategy:\n return
FetchStrategy(primary_engine=BrowserEngine.HTTPX, timeout=30)\n\n async def _fetch_data(self, date: str) -> Optional[str]:\n #
Perform a GET to check status\n try:\n resp = await self.make_request(\"GET\", \"\")\n if resp and get_resp_status(resp) ==
200:\n return \"ALIVE\"\n except Exception:\n pass\n return None\n\n def _parse_races(self, raw_data: Any) -> List[Race]:\n #
Return a single dummy race to indicate success in harvest logs\n if raw_data == \"ALIVE\":\n now = datetime.now(EASTERN)\n
return [Race(\n id=f\"ping_{get_canonical_venue(self.track_name)}_{now.strftime('%y%m%d')}\",\n venue=self.track_name,\n
race_number=1,\n start_time=now,\n runners=[Runner(name=\"Status OK\", number=1), Runner(name=\"Health Check\", number=2)],\n
source=self.source_name,\n discipline=\"StatusCheck\",\n metadata={\"status\": \"HTTP 200\", \"url\": self.official_url}\n
)]\n return []\n",
"name": "OfficialTrackAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialDelMarAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_DelMar\"\n def __init__(self,
config=None): super().__init__(\"Del Mar\", \"https://www.dmtc.com/racing/entries\", config=config)\n",
"name": "OfficialDelMarAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialGulfstreamAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_GulfstreamPark\"\n def
__init__(self, config=None): super().__init__(\"Gulfstream Park\", \"https://www.gulfstreampark.com/racing/entries\",
config=config)\n",
"name": "OfficialGulfstreamAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialTampaBayAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_TampaBayDowns\"\n def
__init__(self, config=None): super().__init__(\"Tampa Bay Downs\",
\"https://www.tampabaydowns.com/racing/entries-results/entries\", config=config)\n",
"name": "OfficialTampaBayAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialOaklawnAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_OaklawnPark\"\n def __init__(self,
config=None): super().__init__(\"Oaklawn Park\", \"https://www.oaklawn.com/racing/entries/\", config=config)\n",
"name": "OfficialOaklawnAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialSantaAnitaAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_SantaAnita\"\n def
__init__(self, config=None): super().__init__(\"Santa Anita\", \"https://www.santaanita.com/racing/entries\",
config=config)\n",
"name": "OfficialSantaAnitaAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialMonmouthAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_MonmouthPark\"\n def
__init__(self, config=None): super().__init__(\"Monmouth Park\", \"https://www.monmouthpark.com/racing-info/entries/\",
config=config)\n",
"name": "OfficialMonmouthAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialWoodbineAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_Woodbine\"\n def __init__(self,
config=None): super().__init__(\"Woodbine\", \"https://woodbine.com/racing/entries-results/\", config=config)\n",
"name": "OfficialWoodbineAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialMeadowlandsAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_TheMeadowlands\"\n def
__init__(self, config=None): super().__init__(\"The Meadowlands\", \"https://playmeadowlands.com/racing/racing-info/\",
config=config)\n",
"name": "OfficialMeadowlandsAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialYonkersAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_YonkersRaceway\"\n def
__init__(self, config=None): super().__init__(\"Yonkers Raceway\", \"https://empirecitycasino.mgmresorts.com/en/racing.html\",
config=config)\n",
"name": "OfficialYonkersAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialJRAAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_JRAJapan\"\n def __init__(self,
config=None): super().__init__(\"JRA Japan\", \"https://japanracing.jp/\", config=config)\n",
"name": "OfficialJRAAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialLaurelParkAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_LaurelPark\"\n def
__init__(self, config=None): super().__init__(\"Laurel Park\", \"https://www.laurelpark.com/racing/entries\",
config=config)\n",
"name": "OfficialLaurelParkAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialPimlicoAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_Pimlico\"\n def __init__(self,
config=None): super().__init__(\"Pimlico\", \"https://www.pimlico.com/racing/entries\", config=config)\n",
"name": "OfficialPimlicoAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialFairGroundsAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_FairGrounds\"\n def
__init__(self, config=None): super().__init__(\"Fair Grounds\", \"https://www.fairgroundsracecourse.com/racing/entries\",
config=config)\n",
"name": "OfficialFairGroundsAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialParxRacingAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_ParxRacing\"\n def
__init__(self, config=None): super().__init__(\"Parx Racing\", \"https://www.parxracing.com/overnights.php\",
config=config)\n",
"name": "OfficialParxRacingAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialPennNationalAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_PennNational\"\n def
__init__(self, config=None): super().__init__(\"Penn National\", \"https://www.pennnational.com/racing/entries\",
config=config)\n",
"name": "OfficialPennNationalAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialCharlesTownAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_CharlesTown\"\n def
__init__(self, config=None): super().__init__(\"Charles Town\", \"https://www.hollywoodcasinocharlestown.com/racing/entries\",
config=config)\n",
"name": "OfficialCharlesTownAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialMountaineerAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_Mountaineer\"\n def
__init__(self, config=None): super().__init__(\"Mountaineer\", \"https://www.mountaineer-casino.com/racing/entries\",
config=config)\n",
"name": "OfficialMountaineerAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialTurfParadiseAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_TurfParadise\"\n def
__init__(self, config=None): super().__init__(\"Turf Paradise\", \"https://www.turfparadise.com/racing/entries/\",
config=config)\n",
"name": "OfficialTurfParadiseAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialEmeraldDownsAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_EmeraldDowns\"\n def
__init__(self, config=None): super().__init__(\"Emerald Downs\", \"https://emeralddowns.com/racing/entries/\",
config=config)\n",
"name": "OfficialEmeraldDownsAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialLoneStarParkAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_LoneStarPark\"\n def
__init__(self, config=None): super().__init__(\"Lone Star Park\", \"https://www.lonestarpark.com/racing/entries/\",
config=config)\n",
"name": "OfficialLoneStarParkAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialSamHoustonAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_SamHouston\"\n def
__init__(self, config=None): super().__init__(\"Sam Houston\", \"https://www.shrp.com/racing/entries\", config=config)\n",
"name": "OfficialSamHoustonAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialRemingtonParkAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_RemingtonPark\"\n def
__init__(self, config=None): super().__init__(\"Remington Park\", \"https://www.remingtonpark.com/racing/entries/\",
config=config)\n",
"name": "OfficialRemingtonParkAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialSunlandParkAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_SunlandPark\"\n def
__init__(self, config=None): super().__init__(\"Sunland Park\", \"https://www.sunlandpark.com/racing/entries/\",
config=config)\n",
"name": "OfficialSunlandParkAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialZiaParkAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_ZiaPark\"\n def __init__(self,
config=None): super().__init__(\"Zia Park\", \"https://www.ziapark.com/racing/entries/\", config=config)\n",
"name": "OfficialZiaParkAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialFingerLakesAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_FingerLakes\"\n def
__init__(self, config=None): super().__init__(\"Finger Lakes\", \"https://www.fingerlakesracing.com/racing/entries/\",
config=config)\n",
"name": "OfficialFingerLakesAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialThistledownAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_Thistledown\"\n def
__init__(self, config=None): super().__init__(\"Thistledown\", \"https://www.thistledown.com/racing/entries/\",
config=config)\n",
"name": "OfficialThistledownAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialMahoningValleyAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_MahoningValley\"\n def
__init__(self, config=None): super().__init__(\"Mahoning Valley\",
\"https://www.hollywood-mahoning-valley.com/racing/entries\", config=config)\n",
"name": "OfficialMahoningValleyAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialBelterraParkAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_BelterraPark\"\n def
__init__(self, config=None): super().__init__(\"Belterra Park\", \"https://www.belterrapark.com/racing/entries/\",
config=config)\n",
"name": "OfficialBelterraParkAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialSaratogaHarnessAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_SaratogaHarness\"\n def
__init__(self, config=None): super().__init__(\"Saratoga Harness\", \"https://saratogacasino.com/racing/entries/\",
config=config)\n",
"name": "OfficialSaratogaHarnessAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialHoosierParkAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_HoosierPark\"\n def
__init__(self, config=None): super().__init__(\"Hoosier Park\", \"https://www.hoosierpark.com/racing/entries/\",
config=config)\n",
"name": "OfficialHoosierParkAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialNorthfieldParkAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_NorthfieldPark\"\n def
__init__(self, config=None): super().__init__(\"Northfield Park\", \"https://www.mgmnorthfieldpark.com/racing/entries/\",
config=config)\n",
"name": "OfficialNorthfieldParkAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialSciotoDownsAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_SciotoDowns\"\n def
__init__(self, config=None): super().__init__(\"Scioto Downs\", \"https://www.eldoradoscioto.com/racing/entries/\",
config=config)\n",
"name": "OfficialSciotoDownsAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialFortErieAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_FortErie\"\n def __init__(self,
config=None): super().__init__(\"Fort Erie\", \"https://www.forterieracing.com/racing/entries\", config=config)\n",
"name": "OfficialFortErieAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialHastingsAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_Hastings\"\n def __init__(self,
config=None): super().__init__(\"Hastings Racecourse\", \"https://www.hastingsracecourse.com/racing/entries\",
config=config)\n",
"name": "OfficialHastingsAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialAscotAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_Ascot\"\n def __init__(self,
config=None): super().__init__(\"Ascot\", \"https://www.ascot.com/\", config=config)\n",
"name": "OfficialAscotAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialCheltenhamAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_Cheltenham\"\n def
__init__(self, config=None): super().__init__(\"Cheltenham\", \"https://www.cheltenham.co.uk/\", config=config)\n",
"name": "OfficialCheltenhamAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class OfficialFlemingtonAdapter(OfficialTrackAdapter):\n SOURCE_NAME = \"Official_Flemington\"\n def
__init__(self, config=None): super().__init__(\"Flemington\", \"https://www.vrc.com.au/\", config=config)\n",
"name": "OfficialFlemingtonAdapter"
},
{
"type": "miscellaneous",
"content": "\n"
},
{
"type": "class",
"content": "class JRAAdapter(BrowserHeadersMixin, DebugMixin, RacePageFetcherMixin, BaseAdapterV3):\n \"\"\"\n Adapter for
Japan Racing Association (JRA).\n Provides high-quality data for Japanese racing.\n \"\"\"\n SOURCE_NAME: ClassVar[str] =
\"JRA\"\n BASE_URL: ClassVar[str] = \"https://japanracing.jp\"\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.HTTPX,\n
enable_js=False,\n timeout=30\n )\n\n def _get_headers(self) -> Dict[str, str]:\n return
self._get_browser_headers(host=\"japanracing.jp\")\n\n async def _fetch_data(self, date: str) -> Optional[Dict[str, Any]]:\n #
JRA uses /racing/calendar/{YYYY}/{MM}/{DD}.html or similar\n dt = parse_date_string(date)\n url =
f\"/racing/calendar/{dt.year}/{dt.month}/{dt.day}.html\"\n \n # Actually JRA has a simpler entries page\n #
https://japanracing.jp/en/racing/go_racing/jra_racecourses/\n # For now we'll check the calendar\n resp = await
self.make_request(\"GET\", url, headers=self._get_headers())\n if not resp or not resp.text:\n # Fallback to current entries\n
resp = await self.make_request(\"GET\", \"/en/racing/go_racing/\", headers=self._get_headers())\n if not resp or not
resp.text: return None\n\n self._save_debug_snapshot(resp.text, f\"jra_index_{date}\")\n parser = HTMLParser(resp.text)\n \n
metadata = []\n # JRA layout is very structured. Look for race links.\n for a in
parser.css(\"a[href*='/racing/calendar/']\"):\n href = a.attributes.get(\"href\")\n if href and \"index.html\" not in href:\n
metadata.append({\"url\": href})\n \n if not metadata:\n return None\n\n pages = await
self._fetch_race_pages_concurrent(metadata[:20], 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 races = []\n
date_str = raw_data[\"date\"]\n try:\n race_date = parse_date_string(date_str).date()\n except Exception:\n race_date =
datetime.now(EASTERN).date()\n\n for p in raw_data[\"pages\"]:\n if p and p.get(\"html\"):\n race =
self._parse_single_race(p[\"html\"], p.get(\"url\", \"\"), race_date)\n if race: races.append(race)\n \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 from header or URL\n venue = \"Japan\"\n header = parser.css_first(\"h1\") or
parser.css_first(\"h2\")\n if header:\n venue = normalize_venue_name(node_text(header))\n \n # Race number\n race_num = 1\n
num_match = re.search(r\"race(\\d+)\", url)\n if num_match: race_num = int(num_match.group(1))\n\n # Runners\n runners = []\n
for row in parser.css(\"table.race_table tr\"):\n cols = row.css(\"td\")\n if len(cols) < 5: continue\n \n try:\n num =
int(clean_text(node_text(cols[0])))\n name = clean_text(node_text(cols[2]))\n if not name or name.upper() in [\"HORSE\",
\"NAME\"]: continue\n \n win_odds = SmartOddsExtractor.extract_from_node(row)\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=num, odds=odds_data, win_odds=win_odds))\n except Exception: continue\n \n if not runners: return None\n \n # Start
time\n start_time = datetime.combine(race_date, datetime.min.time())\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(\"jra\", 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
discipline=\"Thoroughbred\"\n )\n",
"name": "JRAAdapter"
},
{
"type": "miscellaneous",
"content": "\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 dt = parse_date_string(date)\n date_iso =
dt.strftime(\"%Y-%m-%d\")\n url = f\"/racing-index?date={date_iso}\"\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 = node_text(venue_cell)\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+)\",
node_text(link))\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 self.metrics.record_parse_warning()\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 = parse_date_string(raw_data[\"date\"]).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(node_text(name_node))\n\n num_node =
row.css_first(\".runner-number\")\n number = int(\"\".join(filter(str.isdigit, node_text(num_node)))) if num_node else 0\n\n
odds_node = row.css_first(\".odds-win\")\n win_odds = parse_odds_to_decimal(clean_text(node_text(odds_node))) if odds_node
else None\n odds_source = \"extracted\" if win_odds is not None else None\n\n # Advanced heuristic fallback\n if win_odds is
None:\n win_odds = SmartOddsExtractor.extract_from_node(row)\n odds_source = \"smart_extractor\" if win_odds is not None 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, odds_source=odds_source))\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 PROVIDES_ODDS: ClassVar[bool] = False\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\", \"chrome133\")\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 dt =
parse_date_string(date)\n date_iso = dt.strftime(\"%Y-%m-%d\")\n index_url = f\"/form-guide/thoroughbred/{date_iso}\"\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(DATE_FORMAT)\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 # Broaden selectors for race links (Fix 15)\n for link in
parser.css(\"a.fg-race-link, a[href*='/form-guide/'][href*='/R']\"):\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 self.metrics.record_parse_warning()\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 = parse_date_string(raw_data[\"date\"]).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:\n
self.metrics.record_parse_error()\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(node_text(header))\n\n # Strategy 0:
Extract track name from URL if possible (most reliable)\n # URL usually /form-guide/australia/wyong/2026-02-17/R1\n venue =
None\n url_parts = url.lower().split(\"/\")\n if \"form-guide\" in url_parts:\n idx = url_parts.index(\"form-guide\")\n # Skip
discipline if present (thoroughbred, harness, greyhound)\n if len(url_parts) > idx + 1 and url_parts[idx+1] in
[\"thoroughbred\", \"harness\", \"greyhound\"]:\n idx += 1\n if len(url_parts) > idx + 2:\n # idx+1 is country, idx+2 is
track\n venue = normalize_venue_name(url_parts[idx+2])\n\n match = re.search(r\"(\\d{1,2}:\\d{2})\\s+(.+)\", header_text)\n if
match:\n time_str = match.group(1)\n if not venue:\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(node_text(name_node))\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, node_text(num_node)))\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 win_odds =
parse_odds_to_decimal(clean_text(node_text(odds_node))) if odds_node else None\n odds_source = \"extracted\" if win_odds is
not None else None\n\n if win_odds is None:\n win_odds = SmartOddsExtractor.extract_from_node(row)\n odds_source =
\"smart_extractor\" if win_odds is not None else None\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, odds=od, win_odds=win_odds,
odds_source=odds_source))\n except Exception: continue\n\n if not runners: return None\n\n disc =
detect_discipline(html_content)\n\n # S5 \u2014 extract race type (independent review item)\n race_type = None\n is_handicap =
None\n header_node = parser.css_first(\".sdc-site-racing-header__name\") or parser.css_first(\"h1\") or
parser.css_first(\"h2\")\n if header_node:\n header_text = node_text(header_node)\n rt_match =
re.search(r'(Maiden\\s+\\w+|Claiming|Allowance|Graded\\s+Stakes|Stakes)', header_text, re.I)\n if rt_match: race_type =
rt_match.group(1)\n if \"HANDICAP\" in header_text.upper():\n is_handicap = True\n\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 race_type=race_type,\n is_handicap=is_handicap,\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\", \"chrome133\")\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 dt
= parse_date_string(date)\n date_iso = dt.strftime(\"%Y-%m-%d\")\n index_url = f\"/racecards/{date_iso}\"\n intl_url =
f\"/racecards/international/{date_iso}\"\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 self.metrics.record_parse_warning()\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 = parse_date_string(date_str).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:\n continue\n time_match = re.search(r\"/(\\d{4})$\", url)\n if not time_match:\n
if not re.search(r\"/\\d{1,2}$\", url):\n continue\n\n parts = url.rstrip(\"/\").split(\"/\")\n if len(parts) >= 3:\n # Handle
absolute (parts[4]) or relative (parts[2]) URLs\n raw_slug = parts[4] if url.startswith(\"http\") and len(parts) >= 5 else
parts[2]\n\n # Normalize venue from URL slug using word-boundary matching\n slug_words = raw_slug.replace('-', '
').upper().split()\n track_name = None\n for end in range(len(slug_words), 0, -1):\n candidate = \"
\".join(slug_words[:end])\n if candidate in VENUE_MAP:\n track_name = VENUE_MAP[candidate]\n break\n if not track_name:\n
track_name = normalize_venue_name(raw_slug)\n\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_tz = ZoneInfo(\"Europe/London\")\n now_site =
datetime.now(site_tz)\n\n # After building track_map, assign sequential race numbers per track (Fix 2)\n for track, race_infos
in track_map.items():\n # Sort by time to assign correct sequential race numbers\n race_infos_sorted = sorted(\n race_infos,\n
key=lambda r: r[\"time_str\"] or \"0000\",\n )\n for race_idx, r in enumerate(race_infos_sorted, start=1):\n if
r[\"time_str\"]:\n try:\n rt = datetime.strptime(r[\"time_str\"], \"%H%M\").replace(\n year=target_date.year,\n
month=target_date.month,\n day=target_date.day,\n tzinfo=site_tz,\n )\n diff = (rt - now_site).total_seconds() / 60\n if not
(-45 < diff <= 1080):\n continue\n meta.append({\n \"url\": r[\"url\"],\n \"race_number\": race_idx,\n \"venue_raw\": track,\n
})\n except Exception:\n 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 =
parse_date_string(raw_data[\"date\"]).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:\n self.metrics.record_parse_error()\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\n # Strategy 0: Extract track name from URL
(most reliable for UK tracks)\n # ATR URLs: /racecard/[race-title-slug]/date/time\n # e.g.,
/racecard/ludlow-suzuki-king-quad/2026-02-18/1705\n # We need \"Ludlow\" from \"ludlow-suzuki-king-quad\"\n url_parts =
url_path.lower().split(\"/\")\n for marker in [\"racecard\", \"racecards\"]:\n if marker in url_parts:\n idx =
url_parts.index(marker)\n for candidate in url_parts[idx+1:]:\n if (candidate\n and candidate not in [\"international\",
\"uk-ire\", \"usa\"]\n and not re.match(r\"\\d{4}-\\d{2}-\\d{2}\", candidate)\n and not re.match(r\"^\\d{4}$\",
candidate)):\n\n # Word-boundary venue matching against VENUE_MAP\n slug_words = candidate.replace('-', ' ').upper().split()\n
for end in range(len(slug_words), 0, -1):\n test = \" \".join(slug_words[:end])\n if test in VENUE_MAP:\n track_name =
VENUE_MAP[test]\n break\n else:\n # No known venue found \u2014 use first word as fallback\n # (venue names are 1-3 words;
race titles are 4+)\n if len(slug_words) >= 4:\n track_name = normalize_venue_name(slug_words[0])\n else:\n track_name =
normalize_venue_name(candidate)\n break\n if track_name:\n break\n\n header = parser.css_first(\".race-header__details\") or
parser.css_first(\".racecard-header\")\n if header:\n header_text = clean_text(node_text(header)) 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 if not track_name:\n # More
aggressive stripping of race titles from venue\n # We use the VENUE_MAP to try and find a known track name in the header.\n
upper_header = header_text.upper()\n found_track = None\n for known_track in sorted(VENUE_MAP.keys(), key=len,
reverse=True):\n if known_track in upper_header:\n found_track = VENUE_MAP[known_track]\n break\n\n if found_track:\n
track_name = found_track\n else:\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(node_text(track_node)))\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(node_text(time_node)).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\n # Extract
correct race number from header or URL\n race_number = race_number_fallback or 1\n rn_match = re.search(r\"Race\\s+(\\d+)\",
header_text, re.I)\n if rn_match:\n race_number = int(rn_match.group(1))\n else:\n # Fallback to URL if it ends in a small
number\n url_rn_match = re.search(r\"/(\\d{1,2})$\", url_path.rstrip(\"/\"))\n if url_rn_match:\n race_number =
int(url_rn_match.group(1))\n\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\n # S5 \u2014 extract race type (independent review item)\n race_type =
None\n is_handicap = None\n rt_match = re.search(r'(Maiden\\s+\\w+|Claiming|Allowance|Graded\\s+Stakes|Stakes)', header_text,
re.I)\n if rt_match: race_type = rt_match.group(1)\n if \"HANDICAP\" in header_text.upper():\n is_handicap = True\n\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, race_type=race_type, is_handicap=is_handicap,
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(node_text(name_node))\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(node_text(num_node))\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 odds_source = 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 not None:\n odds_source =
\"extracted\"\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(node_text(odds_node)))\n if win_odds is not None:\n odds_source = \"extracted\"\n\n #
Advanced heuristic fallback\n if win_odds is None:\n win_odds = SmartOddsExtractor.extract_from_node(row)\n if win_odds is not
None:\n odds_source = \"smart_extractor\"\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,
odds_source=odds_source)\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 dt = parse_date_string(date)\n date_iso =
dt.strftime(\"%Y-%m-%d\")\n index_url = f\"/racecards/{date_iso}\" 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 self.metrics.record_parse_warning()\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 = parse_date_string(date_str).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\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 = parse_date_string(raw_data.get(\"date\", \"\")).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 = node_text(title_node).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 odds_source = \"extracted\" if win_odds is not None else None\n\n # Advanced
heuristic fallback\n if win_odds is None:\n win_odds = SmartOddsExtractor.extract_from_text(str(t))\n if win_odds is not
None:\n odds_source = \"smart_extractor\"\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,
odds_source=odds_source))\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\n# ----------------------------------------\n# SportingLifeAdapter\n#
----------------------------------------\n"
},
{
"type": "class",
"content": "class SportingLifeAdapter(JSONParsingMixin, BrowserHeadersMixin, DebugMixin, RacePageFetcherMixin,
BaseAdapterV3):\n SOURCE_NAME: ClassVar[str] = \"SportingLife\"\n PROVIDES_ODDS: ClassVar[bool] = False\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 dt = parse_date_string(date)\n date_iso = dt.strftime(\"%Y-%m-%d\")\n index_url = f\"/racing/racecards/{date_iso}/\"
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 self.metrics.record_parse_warning()\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 = parse_date_string(date_str).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\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 =
parse_date_string(raw_data[\"date\"]).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:\n race = self._parse_from_html(parser, race_date, item.get(\"race_number\"), html_content, item.get(\"url\",
\"\"))\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\n # Skip completed races (Insight 4)\n stage = (summary.get(\"race_stage\") or
\"\").upper()\n if stage in [\"WEIGHEDIN\", \"RESULT\", \"OFF\", \"FINISHED\", \"ABANDONED\"]:\n self.logger.debug(\"Skipping
completed race\", stage=stage, venue=summary.get(\"course_name\"))\n return None\n\n # Strategy 0: Extract track name from URL
if possible (most reliable)\n # /racing/racecards/2026-02-18/punchestown/1340/\n track_name = None\n current_url =
data.get(\"query\", {}).get(\"url\", \"\")\n url_parts = current_url.lower().split(\"/\")\n if len(url_parts) >= 5:\n # 0: '',
1: 'racing', 2: 'racecards', 3: 'date', 4: 'venue'\n track_name = normalize_venue_name(url_parts[4])\n\n if not track_name:\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_source = \"extracted\" if wo is not None else None\n\n # Advanced heuristic
fallback\n if wo is None:\n wo = SmartOddsExtractor.extract_from_text(str(rd))\n odds_source = \"smart_extractor\" if wo is
not None else None\n\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, odds_source=odds_source))\n if not runners: return None\n # S5 \u2014 extract race type
(independent review item)\n race_type = summary.get(\"race_title\") or summary.get(\"race_name\") or \"\"\n rt_match =
re.search(r'(Maiden\\s+\\w+|Claiming|Allowance|Graded\\s+Stakes|Stakes)', race_type, re.I)\n if rt_match: race_type =
rt_match.group(1)\n else: race_type = None\n\n is_handicap = summary.get(\"has_handicap\")\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\"),
race_type=race_type, is_handicap=is_handicap, 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, url: str = \"\") -> Optional[Race]:\n h1 =
parser.css_first('h1[class*=\"RacingRacecardHeader__Title\"]')\n if not h1: return None\n ht = clean_text(node_text(h1))\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\n # Strategy 0: Extract track name from URL
if possible (most reliable)\n track_name = None\n url_parts = url.lower().split(\"/\")\n if len(url_parts) >= 5:\n # 0: '', 1:
'racing', 2: 'racecards', 3: 'date', 4: 'venue'\n track_name = normalize_venue_name(url_parts[4])\n\n if not track_name:\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(node_text(nn)).splitlines()[0].strip()\n num_node =
row.css_first('span[class*=\"SaddleCloth__Number\"]')\n number = int(\"\".join(filter(str.isdigit,
clean_text(node_text(num_node))))) if num_node else 0\n on = row.css_first('span[class*=\"Odds__Price\"]')\n wo =
parse_odds_to_decimal(clean_text(node_text(on)) if on else \"\")\n odds_source = \"extracted\" if wo is not None else None\n\n
# Advanced heuristic fallback\n if wo is None:\n wo = SmartOddsExtractor.extract_from_node(row)\n odds_source =
\"smart_extractor\" if wo is not None else None\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, odds_source=odds_source))\n
except Exception: continue\n if not runners: return None\n\n # S5 \u2014 extract race type (independent review item)\n
race_type = None\n ht_node = parser.css_first('h1[class*=\"RacingRacecardHeader__Title\"]')\n if ht_node:\n rt_match =
re.search(r'(Maiden\\s+\\w+|Claiming|Allowance|Graded\\s+Stakes|Stakes)', node_text(ht_node), re.I)\n if rt_match: race_type =
rt_match.group(1)\n\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(node_text(dn)) if dn else None, race_type=race_type, 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 =
parse_date_string(date)\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 = parse_date_string(date).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:\n continue\n vr = clean_text(node_text(hn)) or \"\"\n if \"ABD:\" in vr:\n continue\n\n # Normalize meeting name to strip
session qualifiers (Fix 6)\n vr_words = vr.upper().split()\n for end in range(len(vr_words), 0, -1):\n test = \"
\".join(vr_words[:end])\n if test in VENUE_MAP:\n vr = VENUE_MAP[test]\n break\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(node_text(tn)),
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
self.metrics.record_parse_warning()\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 =
parse_date_string(raw_data.get(\"date\", \"\")).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(node_text(h)) 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(node_text(tn)) or \"\", clean_text(node_text(cn)) or \"\"\n else: continue\n else: rts, tnr =
m.group(1), m.group(2)\n\n # Strategy 0: Extract track name from URL with word-boundary matching (Fix 6)\n track_name = None\n
url_parts = item.get(\"url\", \"\").lower().split(\"/\")\n if \"racecards\" in url_parts:\n idx =
url_parts.index(\"racecards\")\n if len(url_parts) > idx + 1:\n slug = url_parts[idx + 1]\n slug_words = slug.replace('-', '
').upper().split()\n for end in range(len(slug_words), 0, -1):\n test = \" \".join(slug_words[:end])\n if test in VENUE_MAP:\n
track_name = VENUE_MAP[test]\n break\n if not track_name:\n track_name = normalize_venue_name(slug)\n\n if not track_name:\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(node_text(d)) or \"\"\n if \"Distance:\" in dt: dist =
dt.replace(\"Distance:\", \"\").strip(); break\n\n # BUG-16: Improved discipline detection for SkySports\n disc =
detect_discipline(html_content)\n harness_venues = {'le croise laroche', 'vincennes', 'enghien', 'laval', 'cabourg', 'caen',
'graignes', 'mohawk', 'meadowlands', 'woodbine mohawk'}\n if get_canonical_venue(track_name).lower() in harness_venues:\n disc
= \"Harness\"\n elif any(k in html_content.lower() for k in ['trot', 'harness', 'pacer']):\n disc = \"Harness\"\n else:\n disc
= \"Thoroughbred\"\n\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(node_text(nn))\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(node_text(nnode))\n if nt:\n try: number = int(nt)\n except Exception: pass\n onode = (\n
node.css_first(\".sdc-site-racing-card__betting-odds\")\n or node.css_first(\".sdc-site-racing-card__odds\")\n or
node.css_first(\".odds\")\n or node.css_first(\"[class*='odds']\")\n or node.css_first(\"[class*='price']\")\n )\n wo =
parse_odds_to_decimal(clean_text(node_text(onode)) if onode else \"\")\n odds_source = \"extracted\" if wo is not None else
None\n\n # Advanced heuristic fallback\n if wo is None:\n wo = SmartOddsExtractor.extract_from_node(node)\n odds_source =
\"smart_extractor\" if wo is not None else None\n\n ntxt = clean_text(node_text(node)) 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, odds_source=odds_source))\n if not
runners: continue\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\n # S5 \u2014 extract race type
(independent review item)\n race_type = None\n if h:\n rt_match =
re.search(r'(Maiden\\s+\\w+|Claiming|Allowance|Graded\\s+Stakes|Stakes)', node_text(h), re.I)\n if rt_match: race_type =
rt_match.group(1)\n\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, race_type=race_type, source=self.source_name, available_bets=ab))\n return races\n",
"name": "SkySportsAdapter"
}
]
}
```