Lesson 4 — Mock API Call Generation

Overview

This lesson teaches you how to transform natural language requests into structured API calls using LangChain tools. You'll learn to build tools that intelligently map user intent (e.g., "get reach for women 25-34") to specific data types, select the appropriate fixtures, construct valid request bodies, and generate executable curl commands.

This is a critical skill for agentic systems that must:

- Interpret vague user queries ("Show me performance by age group")
- Map concepts to technical terms ("by age group" → `impacts_by_sex_age.json`)
- Validate requests against available resources
- Generate reproducible API calls for debugging and documentation

Learning Outcomes

After completing this lesson, you will be able to:

1. Design tools that map natural language to structured data
2. Implement target audience detection from user queries
3. Build request body generators with validation
4. Create executable curl commands for API testing
5. Handle missing fixtures and data gracefully
6. Write tools that return rich, structured information to agents

Prerequisites

- Completion of Lessons 1-3
- Understanding of REST APIs and JSON
- Familiarity with curl command structure
- Knowledge of Pydantic schemas (Lesson 2)

The Problem: From Intent to API Call

User Request Lifecycle

```
User says: "Show me reach by gender and age"
    ↓
```

```
Agent needs to call: POST /api/campaigns/analyze
    ↓
With body: {
    "campaign_id": "Campaign 1",
    "data_type": "impacts_by_sex_age",
    "target_audience": "all"
}
    ↓
From fixture: "01_pre_postprocessing/input_from_api/impacts_by_sex_age.json"
```

The Gaps to Bridge

1. **Concept mapping**: "by gender and age" → `impacts_by_sex_age`
2. **Audience detection**: No mention of target → default to `"all"`
3. **File discovery**: Check if `impacts_by_sex_age.json` exists
4. **Request construction**: Build valid JSON body
5. **Command generation**: Create curl command for testing

Part 1: Understanding the DataBreeders API

Available Data Types (Campaign 1 Fixtures)

From `Starter Kit – Agentic Models/Campaign 1/01_pre_postprocessing/input_from_api/` :

Data Type	Fixture File	Description
`impacts_by_sex_age`	`impacts_by_sex_age.json`	Reach/frequency by demographics
`impacts_in_target`	`impacts_in_target.json`	Performance in target audience
`json_request`	`json_request.json`	Campaign metadata
`r1plus_in_target_buildup`	`r1plus_in_target_buildup.json`	Daily reach accumulation
`rf_in_target_overall`	`rf_in_target_overall.json`	Reach/frequency distribution
`target_universe`	`target_universe.json`	Target audience definition
`tv_spot_schedule`	`tv_spot_schedule.json`	Ad airings schedule
`universe_by_sex_age`	`universe_by_sex_age.json`	Population demographics

Request Body Structure

```
{
  "campaign_id": "Campaign 1",
  "data_type": "impacts_by_sex_age",
  "target_audience": "women_25_34"
}
```

Target Audience Options

- `"all"` - Total audience
- `"women_25_34"` - Women aged 25-34
- `"men_18_49"` - Men aged 18-49
- Custom segments (if defined in fixtures)

Part 2: Tool Design - DataRequestInput Schema

Pydantic Schema (from lesson_01_working_code.py, lines 414-424)

```python
from pydantic import BaseModel, Field

class DataRequestInput(BaseModel):
    """
    Input schema for requesting data from dataBreeders API.

    This schema represents a user's request to analyze campaign data.
    The tool will map this to the appropriate fixture file.
    """
    data_type: str = Field(
        description=(
            "Type of data requested. Options: "
            "'impacts_by_sex_age', 'impacts_in_target', 'json_request', "
            "'r1plus_in_target_buildup', 'rf_in_target_overall', "
            "'target_universe', 'tv_spot_schedule', 'universe_by_sex_age'"
        )
    )
    target_audience: str = Field(
        default="all",
        description=(
            "Target audience segment. Examples: 'all', 'women_25_34', 'men_18_49'. "
            "Default is 'all' if not specified."
        )
    )
```

Why This Schema?

1. **data_type is required**: The agent must know what data to fetch
2. **target_audience has a default**: Many queries don't specify audience ("Show me reach" → assume all)
3. **Rich descriptions**: Help the LLM choose correct values
4. **Enumeration hints**: List valid options to reduce errors

Part 3: Implementation - generate_mock_api_call Tool

Complete Implementation (lines 427-523)

```python
from pathlib import Path
from typing import Any
import logging

logger = logging.getLogger(__name__)

def generate_mock_api_call(
    data_type: str,
    target_audience: str = "all",
    **_  # LangGraph compatibility (see Lesson 2)
) -> dict[str, Any]:
    """
    Generate a mock API call based on user request.

    This tool:
    1. Maps data_type to fixture file
    2. Checks if fixture exists
    3. Constructs request body
    4. Generates curl command for testing

    Args:
        data_type: Type of data requested (e.g., 'impacts_by_sex_age')
        target_audience: Audience segment (default: 'all')
        **_: Absorbs extra kwargs from LangGraph

    Returns:
        Dictionary with request details, curl command, and fixture info
    """
    logger.info(f"Generating mock API call for: {data_type}, audience: {target_audience}")

    # Step 1: Define fixture directory
    fixtures_dir = Path("Starter Kit - Agentic Models") / \
                   "Campaign 1" / \
                   "01_pre_postprocessing" / \
                   "input_from_api"

    # Step 2: Map data_type to fixture filename
    fixture_filename = f"{data_type}.json"
    fixture_path = fixtures_dir / fixture_filename

    # Step 3: Check if fixture exists
    fixture_exists = fixture_path.exists()
    if not fixture_exists:
        logger.warning(f"Fixture not found: {fixture_path}")
        return {
            "success": False,
            "error": f"Fixture file not found: {fixture_filename}",
            "data_type": data_type,
            "target_audience": target_audience,
            "available_fixtures": list_available_fixtures(fixtures_dir),
        }

    # Step 4: Construct request body
    request_body = {
        "campaign_id": "Campaign 1",
        "data_type": data_type,
        "target_audience": target_audience,
    }

    # Step 5: Generate curl command
    import json
    curl_command = (
        "curl -X POST https://api.databreeders.io/v1/campaigns/analyze \\\n"
        "  -H 'Content-Type: application/json' \\\n"
        "  -H 'Authorization: Bearer YOUR_API_KEY' \\\n"
        f"  -d '{json.dumps(request_body, indent=2)}'"
    )

    # Step 6: Return comprehensive result
    result = {
        "success": True,
        "request_body": request_body,
        "curl_command": curl_command,
        "fixture_path": str(fixture_path),
        "fixture_exists": True,
        "message": f"Mock API call prepared for {data_type} (audience: {target_audience})",
    }

    logger.info(f"Mock API call generated successfully: {fixture_filename}")
    return result

def list_available_fixtures(fixtures_dir: Path) -> list[str]:
    """List all available fixture files (helper function)."""
    if not fixtures_dir.exists():
        return []
    return [f.stem for f in fixtures_dir.glob("*.json")]
```

Step-by-Step Breakdown

Step 1: Define Fixture Directory

```
fixtures_dir = Path("Starter Kit - Agentic Models") / \
               "Campaign 1" / \
               "01_pre_postprocessing" / \
               "input_from_api"
```

Why Path instead of string concatenation?

- Platform-independent (Windows uses `\` , Unix uses `/`)
- Elegant composition with `/` operator
- Built-in methods: `.exists()` , `.glob()` , `.stem`

Step 2: Map data_type to Filename

```
fixture_filename = f"{data_type}.json"
fixture_path = fixtures_dir / fixture_filename
```

Convention: Fixture files follow pattern `{data_type}.json`

- Input: `"impacts_by_sex_age"`

- Output: "impacts_by_sex_age.json"

Step 3: Validate Fixture Exists

```
fixture_exists = fixture_path.exists()
if not fixture_exists:
    logger.warning(f"Fixture not found: {fixture_path}")
    return {
        "success": False,
        "error": f"Fixture file not found: {fixture_filename}",
        "available_fixtures": list_available_fixtures(fixtures_dir),
    }
```

Why return available fixtures on error?

- **Helps the agent** recover: "Did you mean 'impacts_in_target' instead of 'impact_in_target'?"
- **Improves debugging**: User can see what's available
- **Enables self-correction**: Agent can retry with valid option

Step 4: Construct Request Body

```
request_body = {
    "campaign_id": "Campaign 1",
    "data_type": data_type,
    "target_audience": target_audience,
}
```

Fixed fields:

- `campaign_id` : Hardcoded to "Campaign 1" (scope from AGENTS.md)

Dynamic fields:

- `data_type` : From tool input
- `target_audience` : From tool input (defaults to "all")

Step 5: Generate Curl Command

```
import json
curl_command = (
    "curl -X POST https://api.databreeders.io/v1/campaigns/analyze \\\n"
    "  -H 'Content-Type: application/json' \\\n"
    "  -H 'Authorization: Bearer YOUR_API_KEY' \\\n"
    f"  -d '{json.dumps(request_body, indent=2)}'"
)
```

Why generate curl?

- **Reproducibility**: Anyone can test the same request
- **Documentation**: Shows exactly what API call would be made

- **Debugging**: Copy-paste into terminal to verify behavior
- **Portability**: Works on any system with curl installed

Format details:

- Line continuations (`\`) for readability
- Indented for clarity
- Placeholder `YOUR_API_KEY` reminds user to substitute
- `json.dumps(indent=2)` for pretty-printed JSON

Step 6: Return Structured Result

```
result = {
    "success": True,
    "request_body": request_body,
    "curl_command": curl_command,
    "fixture_path": str(fixture_path),
    "fixture_exists": True,
    "message": f"Mock API call prepared for {data_type} (audience: {target_audience})",
}
```

Why so much information?

- `success` : Quick status check for agent
- `request_body` : Structured data for potential use
- `curl_command` : Human-readable execution format
- `fixture_path` : Traceability (which file would be used)
- `message` : Natural language summary for user-facing output

Part 4: Creating the Tool with LangChain

StructuredTool Integration

```
from langchain_core.tools import StructuredTool

# Create the tool
data_request_tool = StructuredTool(
    name="generate_mock_api_call",
    description=(
        "Generate a mock API call to fetch campaign data from dataBreeders. "
        "Use this when the user asks for campaign performance data, reach, frequency, or demographics
        "The tool will check if the requested data type is available in the fixtures "
        "and return a complete API request with curl command."
    ),
    func=generate_mock_api_call,
    args_schema=DataRequestInput,
)
```

Tool Description Best Practices

Good description (from above):

```
"Generate a mock API call to fetch campaign data from dataBreeders.
Use this when the user asks for campaign performance data, reach, frequency, or demographics."
```

Why it's good:

1. **What it does**: "Generate a mock API call"
2. **When to use**: "when the user asks for campaign performance data"
3. **What it returns**: "complete API request with curl command"
4. **Examples**: "reach, frequency, demographics"

Bad description:

```
"API tool"
```

Why it's bad: Too vague - agent won't know when to use it

Part 5: Natural Language → data_type Mapping

Common User Phrases

User Says	Correct data_type
"Show me reach by demographics"	`impacts_by_sex_age`
"How did we perform in our target?"	`impacts_in_target`
"What's the daily buildup?"	`r1plus_in_target_buildup`
"Show the reach and frequency distribution"	`rf_in_target_overall`
"What's the spot schedule?"	`tv_spot_schedule`
"What's the population by age and gender?"	`universe_by_sex_age`

How the LLM Learns the Mapping

1. **Field description** (from Pydantic schema):

```
data_type: str = Field(
    description="Type of data requested. Options: 'impacts_by_sex_age', ..."
)
```

2. **Tool description**:

```
description="...Use this when the user asks for campaign performance data, reach, frequency, or de
```

3. **Agent system prompt** (covered in Lesson 5):

```
When the user asks about demographics, use data_type='impacts_by_sex_age'.
When the user asks about target audience performance, use data_type='impacts_in_target'.
```

Example Interaction

User: "What's the reach for women 25-34?"

Agent reasoning:

1. User wants reach → use `generate_mock_api_call` tool
2. Specific audience mentioned → `target_audience="women_25_34"`
3. General reach data → `data_type="impacts_by_sex_age"` (contains all demographics)

Tool call:

```
generate_mock_api_call(
    data_type="impacts_by_sex_age",
    target_audience="women_25_34"
)
```

Part 6: Error Handling and Edge Cases

Case 1: Missing Fixture

```
# User asks for non-existent data
result = generate_mock_api_call(data_type="impacts_by_daypart")

# Result:
{
    "success": False,
    "error": "Fixture file not found: impacts_by_daypart.json",
    "data_type": "impacts_by_daypart",
    "target_audience": "all",
    "available_fixtures": [
        "impacts_by_sex_age",
        "impacts_in_target",
        "json_request",
        "r1plus_in_target_buildup",
        "rf_in_target_overall",
        "target_universe",
        "tv_spot_schedule",
        "universe_by_sex_age"
    ]
}
```

Agent can now:

- Explain to user: "That data type isn't available"
- Suggest alternatives: "Did you mean 'impacts_in_target'?"
- List what's available

Case 2: Typo in data_type

```
# User says "show impact by age" (singular "impact")
# Agent might extract: data_type="impact_by_sex_age"

result = generate_mock_api_call(data_type="impact_by_sex_age")
# → Fixture not found

# Agent sees available_fixtures includes "impacts_by_sex_age" (plural)
# Agent can correct: "I'll use 'impacts_by_sex_age' instead"
```

Case 3: Unknown target_audience

```
# User: "Show reach for teenagers"
# Agent extracts: target_audience="teenagers"

result = generate_mock_api_call(
    data_type="impacts_by_sex_age",
    target_audience="teenagers"
)

# Tool succeeds (no validation on target_audience in our current implementation)
# But API might reject invalid audience

# Improvement: Add audience validation
```

Improvement: Validate target_audience

```
VALID_AUDIENCES = ["all", "women_25_34", "men_18_49", "women_18_49", "men_25_54"]

def generate_mock_api_call(
    data_type: str,
    target_audience: str = "all",
    **_
) -> dict[str, Any]:
    # ... (fixture validation) ...

    # Validate target_audience
    if target_audience not in VALID_AUDIENCES:
        logger.warning(f"Unknown target audience: {target_audience}")
```

```python
        return {
            "success": False,
            "error": f"Unknown target audience: {target_audience}",
            "valid_audiences": VALID_AUDIENCES,
            "suggestion": "Use 'all' as default or specify a valid audience segment",
        }

    # ... (rest of implementation) ...
```

Part 7: Testing the Tool

Unit Tests

```python
import pytest
from pathlib import Path

def test_generate_mock_api_call_success():
    """Test successful API call generation."""
    result = generate_mock_api_call(
        data_type="impacts_by_sex_age",
        target_audience="women_25_34"
    )

    assert result["success"] is True
    assert result["request_body"]["campaign_id"] == "Campaign 1"
    assert result["request_body"]["data_type"] == "impacts_by_sex_age"
    assert result["request_body"]["target_audience"] == "women_25_34"
    assert "curl" in result["curl_command"]
    assert result["fixture_exists"] is True

def test_generate_mock_api_call_default_audience():
    """Test that target_audience defaults to 'all'."""
    result = generate_mock_api_call(data_type="impacts_in_target")

    assert result["request_body"]["target_audience"] == "all"

def test_generate_mock_api_call_missing_fixture():
    """Test behavior when fixture doesn't exist."""
    result = generate_mock_api_call(data_type="nonexistent_data")

    assert result["success"] is False
    assert "not found" in result["error"]
    assert "available_fixtures" in result
    assert len(result["available_fixtures"]) > 0

def test_generate_mock_api_call_curl_format():
    """Test curl command format."""
    result = generate_mock_api_call(data_type="impacts_by_sex_age")

    curl = result["curl_command"]
    assert "curl -X POST" in curl
    assert "https://api.databreeders.io" in curl
    assert "-H 'Content-Type: application/json'" in curl
    assert "-H 'Authorization: Bearer" in curl
    assert "campaign_id" in curl
```

```python
def test_generate_mock_api_call_kwargs_ignored():
    """Test that extra kwargs (from LangGraph) are ignored."""
    result = generate_mock_api_call(
        data_type="impacts_in_target",
        target_audience="all",
        extra_param="should_be_ignored",
        another_param=123
    )

    # Should succeed despite extra params
    assert result["success"] is True
```

Integration Test with Agent

```python
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain.agents import create_tool_calling_agent, AgentExecutor
from langchain_core.prompts import ChatPromptTemplate

def test_tool_with_agent():
    """Test the tool integrated with a real agent."""

    # Setup
    llm = ChatGoogleGenerativeAI(model="gemini-2.5-flash")
    tools = [data_request_tool]

    prompt = ChatPromptTemplate.from_messages([
        ("system", "You are a helpful assistant for analyzing TV campaign data."),
        ("human", "{input}"),
        ("placeholder", "{agent_scratchpad}"),
    ])

    agent = create_tool_calling_agent(llm=llm, tools=tools, prompt=prompt)
    agent_executor = AgentExecutor(agent=agent, tools=tools)

    # Test: User asks for data
    response = agent_executor.invoke({
        "input": "Show me reach by demographics"
    })

    # Verify agent used the tool
    assert "impacts_by_sex_age" in str(response)
    assert "curl" in str(response) or "API call" in str(response)
```

Part 8: Advanced Patterns

Pattern 1: Multiple Data Types in One Tool Call

```python
class BulkDataRequestInput(BaseModel):
    """Request multiple data types at once."""
    data_types: list[str] = Field(
        description="List of data types to fetch"
    )
    target_audience: str = Field(default="all")

def generate_bulk_mock_api_calls(
    data_types: list[str],
    target_audience: str = "all",
    **_
) -> dict[str, Any]:
    """Generate multiple API calls."""
    results = []

    for data_type in data_types:
        result = generate_mock_api_call(
            data_type=data_type,
            target_audience=target_audience
        )
        results.append(result)

    return {
        "success": all(r.get("success", False) for r in results),
        "results": results,
        "total_requests": len(results),
        "successful_requests": sum(1 for r in results if r.get("success")),
    }
```

Pattern 2: Dynamic Audience Mapping

```
# Map user phrases to technical audience segments
AUDIENCE_ALIASES = {
    "women": "women_25_34",
    "men": "men_18_49",
    "females": "women_25_34",
    "males": "men_18_49",
    "everyone": "all",
    "total": "all",
}

def normalize_target_audience(audience: str) -> str:
    """Map user-friendly terms to technical audience codes."""
    audience_lower = audience.lower().strip()
    return AUDIENCE_ALIASES.get(audience_lower, audience)

# Usage in tool:
def generate_mock_api_call(
    data_type: str,
    target_audience: str = "all",
    **_
) -> dict[str, Any]:
    # Normalize audience
    target_audience = normalize_target_audience(target_audience)

    # ... rest of implementation ...
```

Pattern 3: Fixture Content Preview

```python
import json

def generate_mock_api_call_with_preview(
    data_type: str,
    target_audience: str = "all",
    include_preview: bool = False,
    **_
) -> dict[str, Any]:
    """Generate API call with optional fixture data preview."""

    # ... (standard implementation) ...

    if include_preview and fixture_exists:
        # Load first N lines of fixture
        with open(fixture_path) as f:
            data = json.load(f)

        # Add preview to result
        result["fixture_preview"] = {
            "keys": list(data.keys()) if isinstance(data, dict) else None,
            "size_bytes": fixture_path.stat().st_size,
            "sample": str(data)[:500] + "..." if len(str(data)) > 500 else str(data),
        }

    return result
```

Part 9: Hands-On Exercise

Exercise: Add Data Type Aliasing

Objective: Allow the agent to use alternative names for data types.

Requirements:

1. Create a mapping of user-friendly aliases to technical data_type names:
   - `"demographics"` → `"impacts_by_sex_age"`
   - `"target_performance"` → `"impacts_in_target"`
   - `"daily_reach"` → `"r1plus_in_target_buildup"`
   - `"schedule"` → `"tv_spot_schedule"`
2. Add a `normalize_data_type()` function
3. Update `generate_mock_api_call()` to use it
4. Add alias information to error messages

Starter Code:

```python
# TODO: Create alias mapping
DATA_TYPE_ALIASES = {
    # Add mappings here
}

def normalize_data_type(data_type: str) -> str:
    """Map user-friendly data type names to technical fixture names."""
    # TODO: Implement normalization logic
    pass

def generate_mock_api_call_with_aliases(
    data_type: str,
    target_audience: str = "all",
    **_
) -> dict[str, Any]:
    # TODO: Normalize data_type before processing
    normalized_data_type = normalize_data_type(data_type)

    # TODO: Update fixture lookup to use normalized_data_type

    # ... rest of implementation ...
    pass
```

Solution:

▶ Click to reveal solution

Knowledge Check

Question 1: Request Body Structure

What are the required fields in a dataBreeders API request body?

A) Only `data_type`
B) `data_type` and `target_audience`
C) `campaign_id` , `data_type` , and `target_audience`
D) `campaign_id` and `data_type` (target_audience is optional)

▶ Answer

Question 2: Fixture Lookup

Given `data_type="impacts_by_sex_age"` , what fixture file will the tool look for?

A) `impacts_by_sex_age.txt`
B) `impacts_by_sex_age.json`
C) `Campaign 1/impacts_by_sex_age.json`
D) `input_from_api/impacts_by_sex_age.json`

▶ Answer

Question 3: Error Handling

What should the tool return when a fixture doesn't exist?

A) Raise FileNotFoundError
B) Return None
C) Return `{"success": False, "error": "...", "available_fixtures": [...]}`
D) Return empty curl command

▶ Answer

Question 4: Curl Command Format

Why does the curl command include line continuations (`\`)?

A) Required by curl syntax
B) For readability when printed
C) To escape special characters
D) For JSON formatting

▶ Answer

Question 5: LangGraph Compatibility

Why does the function signature include `**_` ?

A) To accept variable number of positional arguments
B) To absorb extra keyword arguments from LangGraph
C) To make target_audience optional
D) For Python 3.10+ compatibility

▶ Answer

Summary

In this lesson, you learned:

1. **API call generation** from natural language:
    - Map user intent to technical data types
    - Detect target audiences from queries
    - Construct valid request bodies
2. **Fixture management**:
    - Validate fixture existence before calling API
    - Provide helpful errors with available alternatives
    - Use Path objects for platform-independent file handling
3. **Tool design patterns**:
    - Rich return values (success, error, metadata)
    - Executable outputs (curl commands)
    - LangGraph compatibility (`**_` kwargs)
4. **Error handling**:

    - Graceful failures with recovery hints
    - List available options on errors
    - Normalize user input (aliases, case, whitespace)

Next Steps

In **Lesson 5**, you'll learn:

- Creating full LangChain agents with LLM + tools
- Integrating Gemini models
- Writing effective system prompts
- Handling agent responses and message formatting
- Setting up LangSmith tracing

References

- LangChain StructuredTool
- Pydantic Field Descriptions
- Python pathlib
- curl Documentation
- REST API Best Practices

Code Reference

Complete working implementation: lesson_01_working_code.py (lines 427-523)