ChatGPT

Python Smoke Test Automation Guide

This document explains how to design, structure, and implement smoke tests for REST-based services. It uses the **Tenant Management Service** smoke test harness as a canonical example but deliberately avoids any domain-specific assumptions. The goal is to provide a reusable playbook for building smoke tests that exercise a service end-to-end using **real HTTP calls** and **real authentication**.

Smoke tests ensure that the most critical flows of a system work as expected. They complement unit and integration tests by running against a deployed service with production-like credentials and environment. A good smoke test verifies that **entities can be created, read, updated, deleted, listed, assigned**, etc., across all relevant personas. It validates both success cases and expected error paths, and it always leaves the environment in a clean state.

1. Smoke Test Philosophy and Goals

Smoke tests are lightweight end-to-end tests that run against a live instance of a service. Unlike unit tests, which exercise isolated functions, smoke tests verify that the entire stack is wired correctly: request routing, authentication/authorization, persistence, and business logic. Unlike integration tests, which may stub external dependencies, smoke tests exercise real dependencies (databases, authentication services, etc.). They are intentionally shallow—exercising only one or two “happy path” flows for each feature—but they **must**:

- Use **real authentication and authorization**. Tokens are obtained by calling the service’s auth endpoint, and claims are set to model different personas. There are no mocks; if auth fails, the smoke test fails.
- Use **real HTTP calls**. The smoke tests send requests through the HTTP layer so that routing, middleware, and serialization code are exercised.
- Include **setup, validation, and teardown** as first-class concerns. Every resource created during a test must be cleaned up, even if a failure occurs halfway through. Cleanup prevents pollution of shared environments and makes the tests repeatable.
- Remain **independent**. Each test or test suite must not depend on the state left by another. Shared resources are tracked via a state object and removed after each test.

Smoke tests differ from deeper functional tests because they intentionally avoid exploring every edge case. Instead, they verify that the most important API flows work and that the service enforces basic authorization rules.

2. Project Structure Overview

The provided smoke test harness is organized into clear layers to promote reuse and separation of concerns. Understanding this structure helps when adapting the pattern to other services:

Component	Purpose
`main.py`	Entrypoint. Parses command-line arguments, loads configuration, selects personas and test suites, and orchestrates test execution. It creates shared objects such as `State`, `ResultTracker`, `Authenticator`, and `HTTPClient` and invokes each test suite for each persona.
`config.py` / `persona_config.py`	Define the configuration schema. `Config` holds the base URL of the service and a list of personas, loaded from a JSON file. `PersonaConfig` models a user with `name`, `username`, `password`, and optional `claims`.
`auth.py`	Implements JWT handling. `Authenticator` obtains real tokens from the service by calling an auth endpoint. `TokenProvider` merges persona-specific claims with global claims and caches tokens, invalidating them when claims change.
`http_client.py`	Wraps the `requests` library. Injects the current JWT into the `Authorization` header and handles base URL prefixing. Exposes a simple `request(method, path, params, json)` API.
`state.py`	Tracks shared state across test suites. Records created resource IDs per persona and manages global claims (`tenant_id`, `user_id`, etc.). Provides methods to set the current persona and to add or remove resources.
`result.py`	Provides `TestResult` and `ResultTracker` to record test outcomes. Supports summary statistics and HTML/JSON report generation.
`util/…_helper.py`	Contains domain-specific helper classes that wrap REST calls. Helpers standardize request/response handling and ensure consistent return values.
`tests/…`	Contains test suites. Each suite is a function named `run_<feature>`, which accepts a `HTTPClient`, `State`, `ResultTracker`, persona name, and persona config. The suite performs setup, runs core scenarios, validates results, handles errors, and cleans up.

This separation allows you to swap out the service under test while keeping the test harness structure intact.

3. Entry Point and Execution Flow

The smoke test runner is invoked via the `main.py` script. It uses `argparse` to accept a configuration file, optional persona filters, optional test suite filters, and optional report output paths. The heart of the runner is the `run_smoke_tests` function:

```
def run_smoke_tests(config_file, personas=None, tests=None, report_json=None,
report_html=None) -> None:
    config = Config.from_file(config_file)
    # override base_url from environment if provided
    ...
    auth = Authenticator(config.base_url)
    state = State()
    result = ResultTracker()
    # Build selected personas dict
    selected_personas = {p.name: p for p in config.personas}
    if personas:
        selected_personas = {name: selected_personas[name] for name in personas
if name in selected_personas}
    # Determine which test suites to run
    all_tests = {
        "tenants": run_tenant_lifecycle,
        "roles": run_role_management,
        "groups": run_group_management,
        # … other suites …
    }
    if tests:
        selected_tests = {name: all_tests[name] for name in tests if name in
all_tests}
    else:
        selected_tests = all_tests
    # Run tests for each persona
    for name, persona in selected_personas.items():
        initial_claims = dict(persona.claims or {})
        state.set_current_persona(persona)
        # TokenProvider merges persona claims with global claims and caches
tokens
        provider = TokenProvider(
            persona_claims=initial_claims,
            global_claims_source=lambda s=state: s.global_claims,
            token_factory=lambda claims, p=state.get_current_persona():
auth.build_token(p, claims),
        )
        client = HTTPClient(config.base_url, provider)
        for suite_name, suite_fn in selected_tests.items():
            try:
                suite_fn(client, state, result, name, persona)
            except Exception:
                logger.exception("Error running %s tests for persona %s",
suite_name, name)
    # Print summary and optionally write reports
    summary = result.summary()
    print(f"Total tests: {summary['total']}")
    print(f"Successes: {summary['successes']}")
    print(f"Failures: {summary['failures']}")
```

This flow demonstrates several important patterns:

- **Configuration loading**: the base URL and list of personas are read from JSON. An environment variable (`API_BASE_URL`) can override the base URL, enabling testing in different environments without modifying the config file.
- **Persona selection**: if the user specifies `--persona`, only those personas run; otherwise all personas in the config run.
- **Test suite selection**: suites are registered in a dictionary; the `--tests` argument filters which suites to execute.
- **Token provider**: for each persona, a `TokenProvider` is created. It uses the persona's initial claims plus any global claims from `State`. Claims are passed to a `token_factory` lambda that calls `Authenticator.build_token()`. The `TokenProvider` caches the token until claims change.
- **HTTP client**: the `HTTPClient` wraps a `requests.Session` and retrieves a fresh JWT before each request. It inserts the token into the `Authorization` header just before sending the HTTP call.
- **State and result**: one `State` and `ResultTracker` instance are created per run and passed to all suites. They accumulate shared data and results across personas and tests.

After running the suites, the runner prints a summary of total tests, successes, and failures and can optionally write JSON or HTML reports.

4. Core Utility Classes

4.1 Persona Configuration

The `persona_config.py` module defines a simple data class used to model test personas:

```
@dataclass
class PersonaConfig:
    name: str
    username: str
    password: str
    claims: Dict[str, Any] | None = None

    def to_dict(self) -> Dict[str, Any]:
        return asdict(self)
```

A persona encapsulates login credentials and optional JWT claims. Claims are arbitrary key/value pairs (e.g., `{"admin": true}`) that are merged with global claims by the `TokenProvider`. By defining multiple personas in the configuration file, smoke tests can simulate different roles or authorization contexts.

4.2 Authentication and JWT Handling

Authentication is handled by two classes in `auth.py`:

- `Authenticator` obtains a JWT from the service. Its `build_token()` method sends a POST request to the `/utils/jwt` endpoint with the persona's username, password, and claims. It raises exceptions when the auth call fails. Using real auth ensures that tokens are generated and validated by the service, not by a mock.

- `TokenProvider` manages claims and caches tokens. It accepts a `token_factory`, a dictionary of persona-specific claims, and an optional function that supplies global claims from the `State`. When `get_token()` is called, it merges global and persona claims in `_effective_claims()`, creates a fingerprint of these claims, and reuses the cached token if the fingerprint has not changed. Otherwise, it calls the provided `token_factory` to obtain a new token. The provider also exposes `set_persona_claim()` and `invalidate()` to modify claims and force regeneration. This design ensures that all HTTP requests use a valid token reflecting the current claims.

The `State` object holds a list of token providers so that when global claims change (e.g., when a tenant or user ID is set), all cached tokens are invalidated.

4.3 HTTP Client

The `HTTPClient` encapsulates HTTP communication and token injection. It takes a base URL and a `TokenProvider` and uses a persistent `requests.Session`. Before each call, it retrieves the current token and updates the `Authorization` header. It exposes a simple `request()` method that accepts an HTTP method, a path (concatenated to the base URL), and optional `params` or `json` body. Because the token is refreshed on each call, tests can change claims mid-test by calling `state.set_tenant_id()` or `state.set_user_id()`.

4.4 State Management

Maintaining context across multiple test steps and suites is critical. The `State` class stores shared data and global claims:

- It tracks created resource IDs per persona (tenants, roles, groups, users, memberships, assignments, integrations, configs) so that they can be cleaned up later.
- It exposes `set_current_persona()` to record which persona is currently running and invalidates all tokens when switching.
- It exposes `set_tenant_id()` and `set_user_id()` to update global claims and invalidate tokens, ensuring subsequent requests include the new `tenant_id` or `user_id`.
- It provides `add_*` methods (e.g., `add_tenant()`, `add_group_assignment()`) to record new resources. These records are used during teardown.

By centralizing state, tests avoid global variables and can run concurrently for multiple personas.

4.5 Result and Error Tracking

Every operation performed by a smoke test is recorded using the `ResultTracker`. A `TestResult` consists of a descriptive name, the endpoint path, HTTP method, status code, a boolean indicating success, an optional message, and a timestamp. The `ResultTracker.record()` method appends a new result to the list. At the end of the run, the runner calls `ResultTracker.summary()` to compute total tests, successes, and failures. `ResultTracker` can also emit JSON or HTML reports for consumption by CI systems.

Structured result tracking is invaluable for smoke tests because it provides immediate feedback about which step failed and why. Messages often include parsed JSON or the raw response text.

5. Helper Classes and Domain Abstraction

In many APIs, CRUD operations for a resource follow predictable patterns: create, read, update, delete, assign, list. Rather than duplicating HTTP logic in every test, the sample project uses **helper classes** to encapsulate these patterns. The `TenantServiceHelper` in `util/tenant_service_helper.py` is a prime example.

5.1 Normalizing Responses

At the heart of the helper is a private method `_result()` that normalizes an HTTP response into a tuple `(success, id, data, response)`:

```python
def _result(resp, *, ok_statuses, id_field=None) -> tuple:
    success = resp.status_code in ok_statuses
    data = None
    try:
        parsed = resp.json()
        if isinstance(parsed, dict):
            data = parsed
    except Exception:
        data = None
    obj_id = None
    if data is not None and id_field:
        val = data.get(id_field)
        if isinstance(val, str):
            obj_id = val
    return success, obj_id, data, resp
```

This function hides the details of parsing JSON, handling endpoints that return 204, and extracting an identifier. Every helper method calls `_result()` with the expected success status codes and the field to treat as the identifier.

5.2 Helper Methods

Each helper method then wraps a specific endpoint. For example, to create a resource, a helper might call `client.request("POST", "/admin/entities", json=payload)` and then call `_result()` with `ok_statuses=(201,)` and `id_field="entity_id"`. Deleting a resource returns success status 204 and simply returns the provided ID. Helpers can also provide convenience methods like `create_smoke_test_entity()` that generate a unique payload. If a request requires an auto-generated key (e.g., a slug derived from a name), the helper can generate it using a utility such as `_to_snake_lower()`.

This pattern has several benefits:

- **Consistency**: every helper method returns the same tuple shape, making test code uniform.
- **Error isolation**: test code can check the `success` flag and decide whether to proceed or clean up.
- **Extensibility**: new endpoints can be added to the helper without changing existing tests.

When creating a new smoke test for another service, one should implement a helper class for that service's domain objects following the same pattern: wrap each REST endpoint, specify expected success codes, extract IDs, and optionally auto-generate keys.

6. Anatomy of a Smoke Test

A smoke test suite is a function imported from the `tests` package. It follows a common structure: **setup**, **exercise**, **validation**, **teardown**. The `run_group_management` function in the sample project demonstrates this pattern clearly.

6.1 Test Setup

Setup prepares all prerequisites. For example, the group management test first creates a global user, then creates a tenant, and then assigns the user to the tenant. Each step records its result and, if it fails, aborts the test and cleans up:

```
# 1) Create user
ok, user_id, u_data, resp = Helper.create_smoke_test_user(client)
result.record("create_smoke_user", "/users", "POST", resp.status_code, ok and
bool(user_id), None if ok and user_id else f"Failed: {resp.status_code}
{resp.text}")
if not ok or not user_id:
    return
state.add_user(persona_name, user_id)
state.set_user_id(user_id)

# 2) Create tenant
ok, tenant_id, t_data, resp = Helper.create_smoke_test_tenant(client)
result.record("create_smoke_tenant", "/admin/tenants", "POST", resp.status_code,
ok and bool(tenant_id), ...)
```

```
if not ok or not tenant_id:
    # cleanup user
    Helper.delete_user(client, user_id)
    return
state.add_tenant(persona_name, tenant_id)
state.set_tenant_id(tenant_id)

# 3) Assign user to tenant
ok, _, m_data, resp = Helper.assign_user_to_tenant(client, tenant_id, user_id,
membership_payload)
result.record("assign_user_to_tenant", f"/tenants/{tenant_id}/users/{user_id}",
"PUT", resp.status_code, ok, ...)
if not ok:
    # cleanup tenant and user
    Helper.delete_tenant(client, tenant_id)
    Helper.delete_user(client, user_id)
    return
state.add_membership(persona_name, tenant_id, user_id)
```

Several patterns emerge:

- Each call uses the helper to hide the HTTP details. It records the result immediately.
- The test checks the success flag and required IDs before proceeding. If anything fails, it cleans up created resources and returns early.
- Calls to `state.set_user_id()` and `state.set_tenant_id()` update the global claims. This triggers invalidation of cached tokens so that subsequent requests include the proper `user_id` and `tenant_id` claims.

6.2 Core Test Execution

After prerequisites are in place, the test exercises the primary operations. Continuing the group example, it creates two groups, assigns them to the user, verifies membership, removes them, and verifies removal. Creation uses the helper's convenience method and ensures that the API returns a snake-case `group_key`:

```
for _ in range(2):
    ok, group_id, g_data, resp = Helper.create_smoke_test_group(client,
tenant_id)
    # Normalize/validate group_key returned by API
    if ok and isinstance(g_data, dict):
        gname = g_data.get("group_name") or g_data.get("name")
        if isinstance(gname, str) and gname.strip():
            expected_key = _to_snake_lower(gname)
            existing_key = g_data.get("group_key")
            if not isinstance(existing_key, str) or not existing_key.strip():
                g_data["group_key"] = expected_key
            else:
                g_data["group_key"] = _to_snake_lower(existing_key)
    result.record("create_group", f"/tenants/{tenant_id}/groups/", "POST",
resp.status_code, ok and bool(group_id), _msg(g_data) if ok else ...)
    if not ok or not group_id:
        break
    created_group_ids.append(group_id)
    state.add_group(persona_name, tenant_id, group_id)
```

Assigning groups to the user, verifying assignments, and removing them follow the same pattern: call helper functions, record results, check success, and update state. For read operations (e.g., listing user groups), the test parses the returned JSON and asserts that the expected items appear.

6.3 Error Handling

Smoke tests must be resilient. At every step, if an operation fails (e.g., the API returns a 500), the test records the failure and performs best-effort cleanup. For example, if assigning a user to a tenant fails, the test deletes the tenant and user that were just created. This guarantees that no stray resources remain even when failures occur.

Error messages include HTTP status codes and response text. These messages help diagnose why a test failed. The `ResultTracker` aggregates these messages and prints them at the end of the run.

6.4 Teardown and Cleanup

After the core assertions pass, the test cleans up by deleting resources in reverse order of creation. Deleting groups, removing group assignments, removing tenant memberships, deleting tenants, and deleting users should always be attempted, regardless of earlier failures. Use helper methods for deletion and record the results. When designing new tests, always include a teardown phase to ensure repeatability.

7. Creating a New Smoke Test Using This Pattern

To add smoke tests for a new domain or service, follow these steps:

1. **Understand the API**: Identify the endpoints needed for the test (create, read, update, delete, list, and any assignment or search endpoints). Note required claims and relationships between resources (e.g., a resource may belong to a tenant).

2. **Create a Helper**: Write a helper class in the `util` package for your domain. Each helper method should:

3. Accept an `HTTPClient` and any parameters (IDs, payloads).
4. Call `client.request()` with the appropriate HTTP method and path.
5. Specify a tuple of expected success status codes (`(200,)`, `(201,)`, `(204,)`, etc.).
6. Call a private `_result()` function to normalize the response to `(success, id, data, response)`—similar to the pattern in `TenantServiceHelper`.

7. Derive any required keys (e.g., slug or key) from input fields if needed.

For example:

```python
class MyDomainHelper:
    @staticmethod
    def create_resource(client: HTTPClient, payload: dict) -> HelperResult:
        resp = client.request("POST", "/admin/resources", json=payload)
        return _result(resp, ok_statuses=(201,), id_field="resource_id")
```

1. **Define Test Prerequisites**: In your test function (e.g., `run_resource_management`), define which resources must exist before exercising your API. Use your helper or other helpers to create them. Record each operation in the `ResultTracker`. Use `state.add_*()` to track IDs and call `state.set_tenant_id()` and `state.set_user_id()` when claims must change.

2. **Implement CRUD Verification**: After setup, call the helper's create, update, get, list, and delete functions in sequence. For each call, record the result and verify that the response data meets expectations. Use `client.request()` directly for endpoints that are not covered by helpers (e.g., list endpoints returning arrays). Parse the JSON and assert that created resources appear. If any step fails, abort and clean up.

3. **Track State and Results**: Use the `state` object to track all created IDs and assignments. This allows you to delete them later. Call `result.record()` for every operation, with a descriptive test name, endpoint path, method, status code, success flag, and an optional message (e.g., the response body or an error message). Maintaining fine-grained results helps triage failures quickly.

4. **Teardown**: Always implement a teardown phase. Delete resources in reverse order of creation to respect dependencies. If deletion fails, record the error but continue attempting to delete remaining resources. If your API supports idempotent deletes (returning 204 even if the resource is absent), rely on that behaviour to simplify cleanup.

5. **Register the Test Suite**: Import your new test function in `main.py` and add an entry to the `all_tests` dictionary. This allows the suite to be selected via the `--tests` argument.

6. **Add New State Tracking**: If your domain introduces new resource types, add corresponding `add_*` methods to the `State` class to record them. For example, add `add_policy()` or `add_integration()` as needed.

Following this process ensures that your smoke tests adhere to the same philosophy as the sample harness: using real tokens, real HTTP calls, structured result tracking, and robust cleanup.

8. Constraints and Style Rules

When writing smoke tests, adhere to these guidelines:

- **Domain agnosticism**: Write helpers and tests in a way that they can be repurposed for other services. Avoid embedding domain names in helper signatures; pass the endpoint path or resource name as a parameter.
- **No mocks**: Smoke tests must hit the actual service and authentication endpoints. Do not stub out HTTP calls or JWT creation.
- **Explicit error handling**: Always check the `success` flag from helper calls. If it is false, record the error and clean up immediately.
- **Comprehensive teardown**: Even when tests fail mid-way, ensure that all created resources are deleted. This protects the test environment and prevents flakiness in subsequent runs.
- **Readable test names**: Use descriptive names in `result.record()` to make the output easy to understand. Combine them with endpoint and method for clarity.
- **Avoid cross-test interference**: Use the `state` object to isolate resources per persona. Do not store global variables that persist across tests.
- **Use pagination and filtering**: When verifying list endpoints, supply pagination parameters (e.g., `limit` and `offset`) if the API defines them. Verify that created objects appear in the list.
- **Avoid summarizing files without explaining their purpose**: When documenting your test harness, focus on why each file exists and how it contributes to the overall testing strategy, not just what code it contains.

9. Conclusion

A well-designed smoke test suite provides confidence that a deployed service's critical flows work across different personas and claims. By using real authentication, real HTTP requests, helper abstractions, shared state management, and structured result tracking, you can build smoke tests that are robust, repeatable, and easy to extend. The patterns illustrated here—setup, exercise, validate, teardown—are applicable to any REST-based service. When you adopt this template for your own projects, remember to keep the tests simple, cleanup thorough, and reporting clear.
