The AI-Native Smells Detector, Lesson 4: Viscosity

Hands-On: Making the Right Way the Easy Way

Overview

This hands-on shows viscosity in its most common form: a 'correct' API that is so clunky to use that engineers route around it with inline shortcuts, and the shortcuts then hide a typo bug that surfaces only at runtime. We will:

- Read a permissions registry whose 'correct' path takes three required arguments and is so awkward that the engineers added inline if-checks.
- Watch a typo - 'billling' with three Ls - sit silently in the codebase until a real billing user is denied access.
- Refactor to a one-line @requires decorator, a frozenset of valid roles, and a sensible default audit handler.
- Notice that the SAME engineers, given the new API, naturally write the correct code on the first try - because the correct code is now shorter.

Before You Start

- Have Python 3.10+ installed. Verify with python3 --version.
- Open two files from the source directory: permissions_before.py and permissions_after.py.
- Have a terminal open in the same directory, ready to run each file.
- Have the lecture slides visible on a second screen - especially slide 14, the AI-native angle. The before-and-after contrast in this hands-on is the cleanest example of the slope effect in the whole course.

REMEMBER

Viscosity is not a bug in any one function. It is the SLOPE between two ways of doing the same thing. In the 'before' file the slope points toward the shortcut; in the 'after' file the slope points toward the right path. The same engineers, given the same problem, take whichever path is downhill. Your job as an API designer is to choose which direction is downhill.

During Hands-On - Three Steps at a Glance

- **STEP 1** - Read the 'before' file. Find the clunky registry call. Find where engineers gave up and inlined if-checks. Find the typo. ~5 min
- **STEP 2** - Run the 'before' file. Watch the typo surface at runtime as a real user gets the wrong answer. ~3 min
- **STEP 3** - Open the 'after' file. Notice the decorator, the frozenset, and the default. Run it. See the full audit trail. ~7 min

Step 1: Read the Clunky Registry. Spot the Inline Shortcuts.

WHAT TO LEARN

Open permissions_before.py. The team built a PermissionRegistry to centralize permission checks - well-intentioned. But to call it you need three required arguments: an action key (a free-form string), a list of allowed roles, and an audit handler. Three arguments on every call. So the engineers gave up. Look at the UserService methods. Some use the registry. Some use inline if-checks. Some have a typo - 'billling' with three Ls - that the linter cannot catch because the action keys are free-form strings.

Open permissions_before.py and find the registry definition. It looks like this:

```
# Excerpt from permissions_before.py - the clunky 'correct' API
class PermissionRegistry:
    def __init__(self):
        self._rules = {}
        self._audit = []

    def check(self, action: str, user_role: str,
              allowed_roles: list, audit_handler) -> bool:
        # Three required arguments. Free-form action string.
        # No validation. Easy to misuse.
        ok = user_role in allowed_roles
        audit_handler(action, user_role, ok)
        return ok
```

Now scroll to the UserService methods. Notice the pattern - and the inconsistency.

```
# Excerpt from permissions_before.py - mixed call patterns
class UserService:
    def delete_user(self, actor_role, user_id):
        # The 'correct' way - via the registry.
        if not self._perms.check(
                "delete_user", actor_role,
                ["admin"], self._audit):
            raise PermissionError("Not allowed")
        print(f"Deleted user {user_id}")
```

```
    def view_billing(self, actor_role, user_id):
        # Engineer gave up - inline check.
        # Also notice: the typo "billling" hidden in another call site.
        if actor_role not in ("admin", "billing"):
            raise PermissionError("Not allowed")
        print("Viewed billing")

    def export_data(self, actor_role):
        # The 'correct' way again - but with the TYPO 'billling'.
        # No validation catches it.
        if not self._perms.check(
                "export_data", actor_role,
                ["admin", "billling"], self._audit):
            raise PermissionError("Not allowed")
        print("Exported data")
```

WHAT YOU SHOULD GET

An understanding that the 'correct' path is six characters of typing longer than the shortcut, on every call - and engineers, being rational, took the shortcut on roughly half the methods. The codebase is now inconsistent. Some checks audit, some don't. Some typos crash, some silently deny legitimate users. Ask the room: 'Whose fault is this - the engineers or the API?' The answer is the API. The engineers were doing rational math on a sloped surface.

Step 2: Watch the Typo Surface in Production

WHAT TO LEARN

Run permissions_before.py. The script simulates a normal day: alice (admin) deletes a user, bob (admin) resets a password, alice exports data, a real billing user named carol tries to view billing - and is denied. Carol's role is 'billing', spelled with the standard two Ls. The registry rule has 'billling' with three Ls. The typo was hidden in a free-form string. Carol is a real user with a real role, and the system tells her she cannot do her job.

Run it from the terminal:

```
python3 permissions_before.py
```

WHAT YOU SHOULD GET

Output that shows alice and bob's operations succeeding, the export going through, and then a line that reads: 'BUG surfaced: billing user denied billing access (Not allowed)'. Take a beat. Ask the room: 'How long would this bug live in a real codebase?' Realistically, weeks or months. It only surfaces when a billing user happens to try the exported-data feature. There is no test that catches it because the test was written by the same engineer who wrote the typo. There is no compile check because the action key is a string. This is

what viscosity costs you - the right path was so clunky that no one used it consistently, and the inconsistency hid a bug.

Step 3: Flatten the Slope. Make Right = Easy.

WHAT TO LEARN

Open permissions_after.py. Three small changes flip the slope. First, VALID_ROLES is a frozenset declared at module level - any typo in a role name crashes on import, not in production. Second, @requires(*roles) is a decorator: one line above the method, declarative, with the audit handler defaulted to a sensible behavior. Third, the raw registry is gone - there is one obvious way to do this now. Watch how short the UserService methods become.

Open permissions_after.py. The new declarative API:

```
# Excerpt from permissions_after.py - the declarative API
VALID_ROLES = frozenset({"admin", "billing", "support"})
_audit_log: list = []

def _default_audit(actor, action, ok):
    _audit_log.append((actor, action, ok))

def requires(*roles, audit=_default_audit):
    # Validate role names AT IMPORT TIME. Typos crash here.
    unknown = set(roles) - VALID_ROLES
    if unknown:
        raise ValueError(f"Unknown role(s): {unknown}")

    def decorator(method):
        def wrapped(self, actor, *args, **kwargs):
            ok = actor in roles
            audit(actor, method.__name__, ok)
            if not ok:
                raise PermissionError("Not allowed")
            return method(self, actor, *args, **kwargs)
        return wrapped
    return decorator
```

Now the UserService. The right way IS the only way - and it is one line:

```
# Excerpt from permissions_after.py - call sites are tiny
class UserService:
    @requires("admin")
    def delete_user(self, actor, user_id):
        print(f"Deleted user {user_id}")
```

```
    @requires("admin", "billing")   # typo here would crash on import
    def view_billing(self, actor, user_id):
        print("Viewed billing")

    @requires("admin", "billing")
    def export_data(self, actor):
        print("Exported data")
```

Run it:

```
python3 permissions_after.py
```

WHAT YOU SHOULD GET

Five operations succeed. The audit trail prints automatically. Carol the billing user can now view billing - because there is no longer a free-form string to mistype. If you change @requires("admin", "billing") to @requires("admin", "billling"), the module fails to import with 'Unknown role(s): {"billling"}' - the typo dies at startup. The right way is now shorter than any conceivable wrong way. Engineers will naturally write the right code. So will AI assistants.

Optional live demo: in front of the class, change one of the @requires decorators to use 'billling' with three Ls. Re-run. Watch Python refuse to even import the module. Then change it back. The typo cannot hide. That is what 'validate at the edge, loudly' looks like in practice.

Wrap-Up

REMEMBER

We did not write more code in the 'after' file - we wrote less. We did not lecture the engineers about consistency - we removed the incentive to be inconsistent. Viscosity yields to short, declarative, validated APIs more reliably than to any other intervention in this course. When the right path is the shortest path, the team takes it. When it is not, they don't. That is the entire mechanic.

Key takeaways from this hands-on:

- The team is rational. If two methods have different shapes for the same concern, the API designed the inconsistency, not the engineers.
- Free-form strings as keys (action names, role names, event types) are viscosity generators. They invite typos that hide until production.
- Sensible defaults remove a required argument from the call site. Every required argument you can default away makes the right path shorter.

- Decorators are the highest-leverage fix for viscosity in OO codebases. They turn a six-line concern into a one-line declaration above the method.
- AI coding assistants will use whichever API the codebase favors. Flatten the slope and the assistant naturally generates the correct code.

Source File Reference

The two files for this hands-on are permissions_before.py and permissions_after.py. Each is under a hundred lines of standard Python with no external dependencies. Run them in order. In the next and final lesson we tackle Needless Complexity - the smell of code written for problems you do not have yet - and we will see how AI assistants generate this smell more aggressively than any other, and what to do about it.