

Source: authored in `b3u_use_cases` as this use case's build guide.

Running p2t locally

You downloaded `p2t-pipeline.zip`. This is what to do with it.

By the end you will have regenerated a TypeScript web service from Python source on your own machine, started it, and scored it against a test suite written by someone else — 293 assertions covering authentication, articles, comments, profiles and following.

Nothing here needs `sudo`, Docker, or a compiler toolchain. The PostgreSQL used by the tests is a user-space binary distribution unpacked into the project directory, and the grammar compilation happens on `b3u.dev`.

1. What you need

- **Node.js and npm** — for the TypeScript runtime and the test runner.
- **Python 3** — the pipeline stages are Python.
- **A b3u.dev account with an active subscription** — step 3 compiles ten grammars on the service.
- About **500 MB** of disk and a working internet connection.

2. Unpack

```
mkdir -p ~/ccsUseCases/p2t
cd ~/ccsUseCases/p2t
unzip ~/Downloads/p2t-pipeline.zip
```

You now have the pipeline: `decomposer`, `sgdl`, `emitter`, `assembler`, `bodies`, `runtime`, plus tests and scripts.

3. One-time setup

Four commands. They are idempotent — re-running them detects existing state and skips the heavy parts.

⚠ **Do not use sudo for any of them.** `install_postgres.sh` unpacks a PostgreSQL distribution into the project directory and initialises a database cluster owned by *you*; PostgreSQL's `initdb` refuses to run as root and will stop with `initdb: error: cannot be run as root`. If you have already tried it with `sudo`, remove the half-built directory first — `rm -rf tests/postgres` — because the downloaded files will be owned by root and the retry will fail on permissions.

```
scripts/install_postgres.sh    # user-space PostgreSQL 15, no sudo (~200 MB)
scripts/install_upstream.sh    # clones the upstream Python service at a pinned commit
(cd tests && npm install)      # newman, the Postman test runner
(cd runtime && npm install)     # tsx + express, to run the emitted TypeScript
```

`install_upstream.sh` clones the reference implementation this project translates — `nsidnev/fastapi-realworld-example-app` — and pins the commit. That repository is also where the test suite comes from, which is the point: **we did not write the tests we are graded by**.

4. Compile the grammars on b3u.dev

The first pipeline stage reads the Python source through ten compiled grammar frontends, one per kind of thing the service contains — data model entities, route contracts, handlers, repository queries, auth policy, validation rules, error handling, configuration, bootstrap.

One command compiles all ten on the service and installs them:

```
export B3U_BASE=https://b3u.dev
export B3U_EMAIL=you@example.com
export B3U_PASSWORD=...
scripts/build_frontends.sh
```

Each `.sgr` grammar is uploaded, compiled, and the resulting frontend downloaded and installed as `sgdl/<family>/bin/ccsf<N>`. Ten upload-compile-download cycles; expect a couple of minutes.

The service path needs an account with an active subscription — the login above is the same one the site uses. **Two ways forward if you do not have one, or do not want to use the service at all:**

- **--quick** (section 6) needs no frontends at all and still runs the full 293-assertion suite. Most readers should start there.

- **scripts/build_frontends.sh --local** builds the ten frontends with a local cppcc instead of the service, if you have one licensed. It needs CPPCCHOME set and nothing else; every file it uses already shipped in this archive, since each family carries its own build/makeall.

If the login reports that a Cloudflare Access wall answered instead of the application, B3U_BASE is pointed at a private host rather than at the public service; it should be `https://b3u.dev`.

The grammars themselves are in the zip, readable, and are the artifact you are compiling — `sgdl/F7_AuthPolicy/build/F7_AuthPolicy.sgr` and its nine siblings.

5. Convert and test

```
scripts/convert_and_test.sh
```

```
== 1. regenerate the TypeScript from the Python source ==
PASS  decomposer – output reproduced byte-identically
PASS  emitter    – output reproduced byte-identically
PASS  assembler  – output reproduced byte-identically
PASS  bodies     – output reproduced byte-identically
== 2. serve the emitted TypeScript and run the upstream suite ==
PASS  external suite: 293 / 293 assertions passing

P2T CONVERT + TEST: ALL PASS
```

Two halves, and they check different things.

Stage 1 is reproducibility. Each stage rewrites its output and diffs it against what shipped in the zip. A stage passes only if it reproduces itself *byte-for-byte* — so a passing run means the pipeline is deterministic, not merely that it produced something.

Stage 2 is correctness against an outside standard. The emitted TypeScript is served and hit with the upstream Postman collection. The assertions encode the public RealWorld/Conduit API specification.

6. If you want to skip ahead

```
scripts/convert_and_test.sh --quick
```

Skips the regeneration and goes straight to serving and testing the TypeScript that shipped in the zip. This needs **no** grammar frontends, so you can run it before step 4 — steps 2, 3 and `--quick` alone will show you the 293 assertions passing.

That is the fastest path to seeing the result. Step 4 is what lets you verify the result was *produced* rather than shipped — and the script says which of the two you just did, so a `--quick` run cannot be mistaken for a full one.

7. When something fails

grammar frontends not built — step 4 has not run, or ran partially. The message lists which families are missing. `scripts/build_frontends.sh F7_AuthPolicy` rebuilds one.

setup incomplete — one of step 3's four commands has not run. The message names what is missing.

missing source/upstream — `install_upstream.sh` has not run. The decomposer reads the Python service from there.

A stage reports DIVERGED — it produced output differing from the shipped golden. That is a genuine finding rather than a setup problem, and the run prints the tail of the diff.

8. What this shows, and what it does not

It shows that a non-trivial translation — a real service with authentication, relational data and pagination — can be produced by a pipeline and then held to a standard its authors did not set.

It does not show that the generated TypeScript is code you would enjoy maintaining, and this is not a general Python-to-TypeScript translator: the pipeline was built for this service's shape. What generalises is the method and the checking, not the emitter.

The full account is the paper *Cross-Language Translation* in the documentation section.

Terms

Development and evaluation use under your subscription (Terms §5.3). Production use of generated components requires a separate agreement — contact support@b3u.dev (§5.4).