Database migrations (SQL bundles)

How Le Sage 13 (and Inpro 9, Aïda 6) upgrade their databases: bundles, adding a column, adding a function, testing locally, releasing a bundle and building the installer.

- Adding a column (a table step)
- Adding or changing a function
- Testing locally without releasing
- Releasing a bundle and building the installer
- At launch: the connect rule and its messages

Adding a column (a table step)

This is what you used to do by adding a block to `upgradedb.sql`. Now it is a **step file** in `~/dev/improvize/lesage/sql/steps/`.

1. Pick the version

Look at `lesage/sql/version` and `lesage/released.json`:

- **The current version is in `released.json`** (e.g. `version` = 1 and `"1"` is released): bump `version` to `2`. Your new steps are `0002-*.sql`.
- **The current version is not released yet** (you already bumped it): keep it and add your step under that number.

Several files per version are fine. They run in byte-wise filename order, so use a suffix to order them: `0002-1-…`, `0002-2-…`. Lint refuses a step numbered above `version`.

> Never add a block to a step that belongs to a released version (e.g. `0001-1-upgradedb.sql`). The databases that already have version 1 would never run it, and lint fails because the released checksum no longer matches.

2. Write the step

```
-- steps/0002-1-oeu-isdeposited.sql
-- Why: (ticket / one sentence). Converges on every replay: addcolumn skips an existing
column.
DO $step$
BEGIN
  IF tableexists('oeu') THEN
    PERFORM addcolumn('oeu', 'IsDeposited', 'BOOLEAN NOT NULL DEFAULT FALSE',
      'The work has been deposited with the society');
  END IF;
END
$step$;
```

Same idea as in `upgradedb.sql`, with three differences:

- it is a `DO` block (or plain DDL), not the body of a function;
- it runs **as the owner role** (`SET LOCAL ROLE` is done by the engine), inside the engine's transaction;
- it may be run **again** at any time on a database that already has the column (reinstall), so it must be a no-op then.

Backfill only when the column is new

`addcolumn` returns `TRUE` when it created the column, so a backfill runs once and a replay skips it:

```
DO $step$
BEGIN
  IF addcolumn('oeu', 'IsDeposited', 'BOOLEAN NOT NULL DEFAULT FALSE') THEN
    UPDATE oeu SET IsDeposited = TRUE WHERE datedepotsacem IS NOT NULL;
  END IF;
END
$step$;
```

Other guarded changes

```
DO $step$
BEGIN
  -- new table
  CREATE TABLE IF NOT EXISTS public.depots (iddepot SERIAL PRIMARY KEY, idoeu INTEGER NOT NULL,
datedepot DATE);

  -- index
  IF NOT indexexists('ix_depots_idoeu') THEN
    CREATE INDEX ix_depots_idoeu ON public.depots (idoeu);
  END IF;

  -- foreign key (helper checks constraintexists itself)
  PERFORM addconstraintforeignkey('fk_depots_oeu', 'depots', 'idoeu', 'oeu', 'idoeu', 'ON DELETE
CASCADE');

  -- other constraint
  IF NOT constraintexists('depots', 'ck_depots_date') THEN
    ALTER TABLE public.depots ADD CONSTRAINT ck_depots_date CHECK (datedepot >= '1900-01-01');
  END IF;
```

```
-- remove a column / widen a VARCHAR
PERFORM dropcolumn('oeu', 'oldfield');
PERFORM resizefield('oeu', 'soustitre', 100);
END
$step$;
```

> `addconstraintforeignkey` cleans orphans before adding the key: it sets them to NULL when the column is nullable and **deletes the rows** otherwise. Check the data first.

Helpers available in steps (from `common`)

Helper	Returns
`addcolumn(table, column, type [, comment])`	`TRUE` if it created the column
`dropcolumn(table, column)`	boolean
`resizefield(table, column, newsize)`	boolean
`addconstraintforeignkey(name, table, column, othertable, othercolumn, options)`	void, no-op if the constraint exists
`tableexists`, `columnexists`, `indexexists`, `constraintexists`, `triggerexists`, `columnisnullable`, `columntype`, `hasdefault`	tests for guards
`dropconstraint`, `addcreatorcreated(table)`, `fixsequence(table)`	

Table names without a schema resolve to the bundle's schema (`search_path` file, `public` for Le Sage).

What "idempotent" means here

- Guard every object on its existence (`IF NOT EXISTS`, `indexexists`, `constraintexists`...). Never rely on "this step already ran".
- A replay converges to the target but **does not fix an object with a wrong definition**: changing an existing column or constraint takes a new step that detects the old shape (`columntype`, `constraintexists`) and changes it.
- Table DDL (`CREATE/ALTER/DROP TABLE`, `CREATE INDEX`, `VALIDATE CONSTRAINT`) is allowed **only in `steps/`**.

Forbidden in every bundle file (lint refuses)

`BEGIN` / `COMMIT` / `ROLLBACK` / `SAVEPOINT`, `SET ROLE` / `RESET ROLE` / `setrole()`, plain session `SET` (`SET LOCAL` is fine), `CONCURRENTLY`, `VACUUM`, `COPY`, `CREATE/ALTER/DROP EXTENSION`, psql `\` commands, `udf_dropfunction`. Files are UTF-8 without BOM, LF line endings.

Then

Run the checks and try it on your test database: see *Testing locally without releasing*.

Adding or changing a function

Functions are no longer dropped and recreated at every launch. They live in `functions/*.sql` of a bundle and are (re)created only when the bundle changes, as `<owner>_ddl`.

1. Choose the bundle

- **`lesage/sql/functions/`** for anything Le Sage uses (calculations, ABR, décomptes, imports…). This is the normal case.
- **`common/sql/functions/`** only for a helper several products need. It ships to Inpro and Aïda too, and must stay backward compatible. A function name belongs to one bundle only (lint rule `owner-bundle`).

Files are grouped by topic (`calculatecopyright.sql`, `abr3.sql`, `decomptes-lesage.sql`…). Add to the right one, or create a new `.sql` file: every file in `functions/` is run.

2. Write it

```
CREATE OR REPLACE FUNCTION public.worktitle(AIdOeu INTEGER) RETURNS TEXT AS
$BODY$
SELECT COALESCE(titrelong, '') FROM oeu WHERE idoeu = AIdOeu;
$BODY$ LANGUAGE sql STABLE
SET search_path = public;

COMMENT ON FUNCTION public.worktitle(INTEGER) IS 'Long title of a work, '''' when none';
```

- Only `CREATE OR REPLACE FUNCTION` + `COMMENT ON FUNCTION`. No `udf_dropfunction` prelude any more.
- `SET search_path` on the function is recommended (not enforced).
- **No table DDL** inside a function body: use `CREATE TEMP TABLE`, or `TRUNCATE`/`DELETE` a work table that a step created (like `zDroits`, `zQuoteParts`).
- **No role switch** (`setrole`, `SET ROLE`): functions run as the plain login now.
- Never touch the FROZEN functions `oversimplify`, `shortcode`, `unaccent` (indexes depend on them).

3. Regenerate `functions.list`

`functions.list` holds the exact signature of every function the bundle manages (`public.worktitle(integer)`). The engine drops any overload not listed, so a new function must be in it. It is generated, never hand-edited:

```
cd ~/dev/improvize/common
PGHOST=/var/run/postgresql npm run check -- apply sql ../lesage/sql --write-lists
```

Commit the updated `lesage/sql/functions.list` with your function.

Changing an existing function

Change	What to do
Body only	Edit it. Nothing else (apart from the version bump rule below).
Argument types (new signature)	Edit the `CREATE`, run `--write-lists`. The old signature is no longer listed, so the engine drops it on the next upgrade.
Return type or argument names (same signature)	Put `DROP FUNCTION IF EXISTS public.f(argtypes);` just before the `CREATE`: Postgres refuses `CREATE OR REPLACE` for that.
Remove or rename	Delete the definition, add the old signature to `retired.list` (canonical form, e.g. `public.f(integer, text)`), run `--write-lists`.

> Drops never use `CASCADE`: if a view or another object depends on a function being dropped, the whole upgrade fails and rolls back. Drop or recreate the dependent object first (in `post/` or a step).

The version bump applies to functions too

The bundle's checksum covers every file, functions included. Changing only a function after a release still needs `version` N → N+1 (no step file needed). Clients with N then get an *upgrade* that reruns `functions/`.

Neighbours of `functions/`

You want	Put it in
A view, a trigger, a `GRANT`, audit enablement	`post/*.sql` (runs every upgrade, as the owner; views `CREATE OR REPLACE`, triggers guarded with `triggerexists`). Le Sage's `post/` must end with `SELECT audit.sync()` (`90-audit.sql`).
Reference rows (territories, SPRD…)	Edit `lesage/data-src/<name>.csv` (+ `.json` for table/key/types), then `npm run check -- lint sql ../lesage/sql --write-data` regenerates `data/<name>.sql`.
A table, column, index, constraint	A step: see *Adding a column*.

Testing locally without releasing

You never need an installer or a release to try a change. There are two loops: the check script on the Linux host (fast, throwaway databases), and `Sage.exe` in the VM pointed straight at your working copies.

Loop 1: the check script (Linux host)

Runs on the local PostgreSQL as a superuser, creates throwaway `check_<pid>_<n>` databases from `lesage/makedb/empty.sql` plus a synthetic seed, and drops them at the end. No client data involved.

```
cd ~/dev/improvize/common
export PGHOST=/var/run/postgresql   # over the socket: the TCP default asks for a password

npm run check -- lint sql ../lesage/sql     # no database: shape, forbidden SQL, released.json
rules (seconds)
npm run check -- apply sql ../lesage/sql    # the apply cases below, common + lesage
npm run check -- all                        # lint + apply on common, lesage, inpro, aida and
LS+IN
```

Apply case	Proves
`bootstrap`	The whole bundle applies on an empty legacy database; a second run is `upToDate`; `functions.list` matches what was created; owners are right.
`replay`	A reinstall (every step again, on seeded tables) leaves the exact same schema: your steps are idempotent.
`reload`	The dev reload (functions + post) works and changes no schema.
`upgrade`	The last **released** version, then yours as a normal upgrade, gives the same schema as a fresh bootstrap: your new steps work on a database that already had version N.
`foreign`	Step 0001 refuses a database that isn't a Le Sage one (SQLSTATE `IZ001`).

Useful options: `--keep` keeps the throwaway databases so you can open them in psql; `--write-lists` rewrites `functions.list` after adding or removing a function.

Loop 2: Sage.exe in the VM on `sage13test`

Two command-line parameters make the exe read your working copies:

- `sqlroot=Z:\dev\improvize\` — the exe reads `Z:\dev\improvize\common\sql\` and `Z:\dev\improvize\lesage\sql\` (the host's git checkouts through the VirtualBox share) instead of its own folder.
- `dev=1` — shows the reload buttons (main window and décomptes screen).

IDE run parameters (`Sage.params` already has the `SQLROOT` macro):

```
srv=192.168.56.1 user=jc pwd=0 owner=izowner db=sage13test sqlroot=Z:\dev\improvize\ dev=1
```

`sage13test` is the test copy of `sage12demo` owned by `izowner` (NOLOGIN), with `izowner_ddl` created by `ddl-roles.sql` and `jc` as a plain member login. That is the real hosted setup.

> Don't use the "*Local : sage12demo (bundles, dev)*" entry for this: its owner `cat` is a superuser, and the connect rule refuses to upgrade for a superuser owner (`ownersuperuser`). And never point a test exe at a client database.

What happens when you relaunch

You did	Next launch does
Bumped `version` to 2 and added `0002-*.sql`	*upgrade* 1 → 2: runs the 0002 steps, then data, functions, post.
Edited anything again, version still 2	Checksum differs → *reinstall*: replays **every** step, then the rest. That's also a free idempotency test.
Nothing	*upToDate*, nothing written.

So the loop is simply: **edit the SQL on the host → relaunch Sage.exe**. No copy, no installer.

Iterating on a function without relaunching

With `dev=1`, the reload button calls `ReloadBundles`: it reruns `functions/` and `post/` of both bundles (no steps, no cleanup, no version check) and marks the ledger `dirty`, so the next launch does a full reinstall. Good for tuning `calculatecopyright` or a décompte function in a running session.

Useful SQL on the test database

```
-- where is it?
SELECT bundle, version, dirty, appliedat, appliedby FROM improvize.bundles;
SELECT * FROM improvize.bundlehistory ORDER BY id DESC LIMIT 10;

-- force a full replay at the next launch
UPDATE improvize.bundles SET dirty = true WHERE bundle = 'lesage';

-- you tested an unreleased version 2 and want to go back to an exe that ships 1
-- (otherwise: clientBehind). Next launch with 2 will upgrade 1 → 2 again.
UPDATE improvize.bundles SET version = 1, dirty = true WHERE bundle = 'lesage';
```

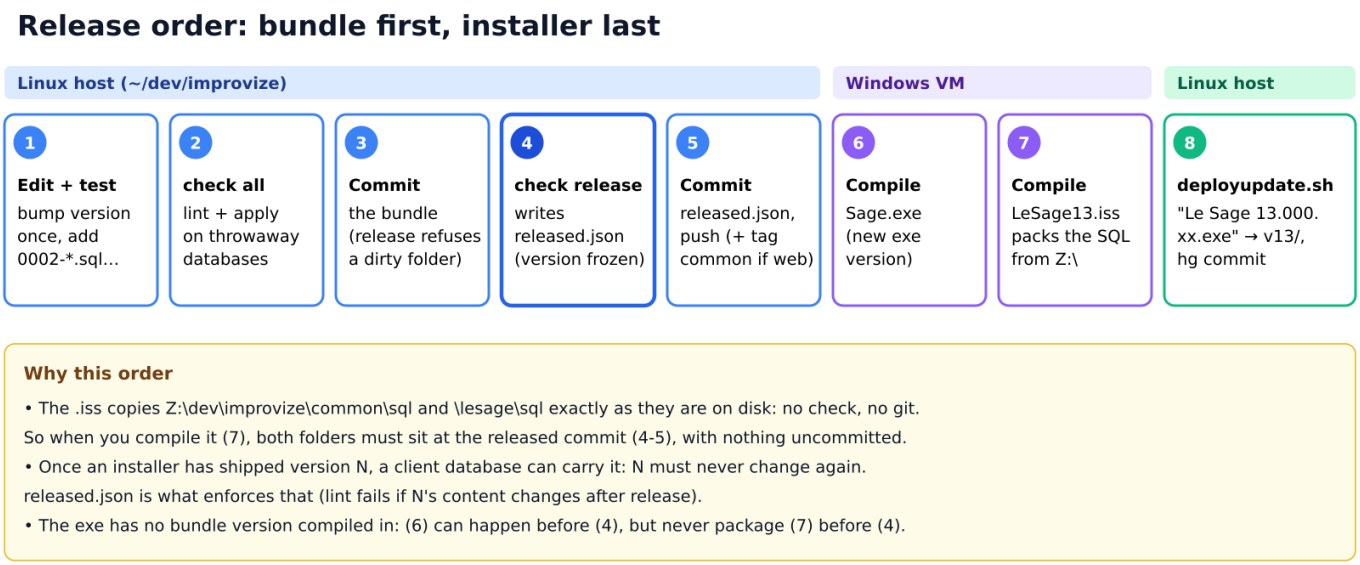
(As `cat`, superuser, on your local server. The ledger belongs to `izowner_ddl`.)

Notes

- `~/dev/improvize/bundles-dev/` is an old manual copy from 4 Nov 2026, made when `lesage/sql` wasn't on `main` yet. Not needed any more: point `sqlroot` at `Z:\dev\improvize\` . Since 18 Nov 2026 it is the GitLab repo `improvize/bundles-dev`, without `common/` (that bundle is in `db-bundles`); nothing reads it.
- Le Sage web doesn't apply bundles yet (the web runner is still to come). Its `npm run build` runs `check lint` on both bundles in `prebuild`, so a bundle that fails lint breaks the web build too.
- A test database that received your unreleased version keeps working after the release: same checksum → *upToDate*; different checksum → *reinstall*.

Releasing a bundle and building the installer

"Releasing" a bundle version means freezing it: recording its checksum and git commit in `released.json`, because from then on a real database may carry it. Delphi installers ship **only released versions**.



Checklist

On the Linux host

1. **Version bumped**: `lesage/sql/version` is above the last entry of `lesage/released.json` (same for `common` if you changed it). New steps are named `NNNN-*.sql` with that number.
2. **Everything passes**:

   ```
   cd ~/dev/improvize/common
   export PGHOST=/var/run/postgresql
   npm run check -- all
   ```

3. **Commit the bundle** (and `functions.list`, `data/`) and push. `check release` refuses a bundle folder with uncommitted changes, and records the current `HEAD`.

```
cd ~/dev/improvize/lesage
git add sql data-src && git commit -m "feat(sql): …" && git push
```

4. **Release**: runs lint and apply again, then writes the new entry in each `released.json` whose version is new (an already released version is just reported).

   ```
   cd ~/dev/improvize/common
   npm run check -- release sql ../lesage/sql
   ```

   Last line expected: `lesage: released version N sha256:… at <commit> — commit released.json`.
5. **Commit** `released.json` in each repo it touched, and push:

   ```
   cd ~/dev/improvize/lesage
   git add released.json && git commit -m "chore(release): lesage bundle version 2" &&
   git push
   ```

   The lesage-web commit hook runs prettier on `released.json` and reflows it: harmless, the lint compares the values only (`node scripts/lint-bundles.mjs` still says `ok`). If `common` was released too: same in `~/dev/improvize/common`, and tag it (`v1.x.y`) when the web apps must pick it up (they pin `improvize-db-bundles#semver:^1.0.0`, then `npm update improvize-db-bundles`).

In the Windows VM

6. **Compile** `Sage.exe` with its new exe version (`versionprefix.txt` → `genversion.cmd`, which writes `version.RES` and the `version.ini` the `.iss` reads).
7. **Compile** `LeSage13.iss`. It copies `Z:\dev\improvize\common\sql\*` → `{app}\common\sql` and `Z:\dev\improvize\lesage\sql\*` → `{app}\lesage\sql`, after wiping both folders (`[InstallDelete]`) so no stale file survives. Output: `K:\Dev\Delphi\Sage13\Le Sage 13.000.xx.exe`.

Back on the host

8. **Publish**: `./deployupdate.sh "Le Sage 13.000.xx.exe"` in `Sage13/` (Mercurial commit + push, upload to `…/lesage/program/v13/`, `makelatest.pl`, announcement mail).

Why the bundle comes first

- The `.iss` does no check at all: it packs whatever is in the two folders at compile time. If you compile it while you have uncommitted or unreleased SQL there, that SQL goes to clients and becomes a version nobody can reproduce.

- Before step 7: `git -C ~/dev/improvize/lesage status sql` and `git -C ~/dev/improvize/common status sql` must be clean.
- The exe has no bundle version compiled in. Compiling `Sage.exe` before the release is fine; **packaging** before it is not.

Common situations

Situation	Do
Exe-only fix, no SQL change	No release. Bundles are already at a released commit; just steps 6–8.
Bug in a released version N	Never edit N. Bump to N+1 and fix it with a new step or a corrected function.
`check release` says "uncommitted changes in the bundle folder"	Commit (or stash) first: it records the commit, so the folder must match it.
Lint: `released` rule fails	You changed a released version's content without bumping `version`.
Changed only `common`	Release `common`, then rebuild the installer of **every** product (Le Sage, Inpro, Aïda): each ships its own copy.

> Never put a 13 exe in `program/v12/`: `makelatest.pl` would offer it to every 12.x, ready database or not. 13 goes in `v13/` only.

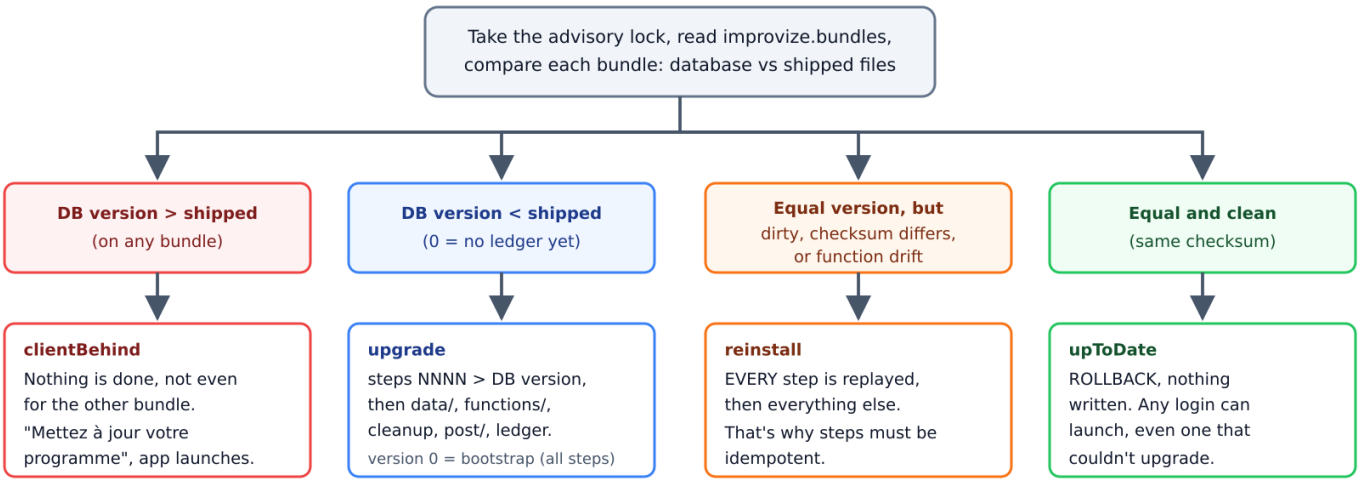
At launch: the connect rule and its messages

What `Sage.exe` does with the bundles at each launch (`TDM.ApplyBundles` → `RunConnectRule` in `catBundleUpgrade.pas`), and how to read what went wrong.

Startup sequence

1. Read the parameters (`srv= owner= user= pwd= db=`, plus `sqlroot=` and `dev=1`), connect as the login.
2. Licence check.
3. **Connect rule**, one transaction: wait for the advisory lock (another PC may be upgrading: splash with *Annuler*, gives up after 5 minutes), read the ledger, decide, apply.
4. Write the legacy stamp `config.lastversiondbupdate` (only after *upToDate/upgraded*, never lowered).
5. Open the datasets, as the plain login.

The connect rule: what Sage.exe decides at launch



When there is work, the login must reach <owner>_ddl, and <owner> must not be a superuser, else **refused**.
Any SQL error → **failed** (file + line shown, all rolled back). Both offer « Lancer quand même » / « Quitter ».

Roles

- `<owner>` (e.g. `izowner`): owns the tables. Steps, `data/` and `post/` run as this role.

- `<owner>_ddl`: owns the managed functions and the ledger, so an old exe running as `<owner>` can no longer drop or overwrite them. `functions/` runs as this role.
- Created per server by `common/roles/ddl-roles.sql` (operator, see `common/roles/RUNBOOK.md`); rerun it when a new login is added, or that login gets *refused* as soon as there is work to do.

Messages and what they mean

Result	Cause	Fix
clientBehind "Quelqu'un de votre équipe utilise une version plus récente..."	The database has a higher bundle version than this exe ships.	Update the exe. On a test DB: lower `version` in `improvize.bundles`.
refused `noddlrole` / `notddlmember`	`<owner>_ddl` is missing, or the login isn't a member.	Run `ddl-roles.sql` on the server.
refused `ownersuperuser`	The owner role is a superuser (e.g. `owner=cat` locally).	Use a non-superuser owner role.
failed "L'installation de Le Sage est incomplète : réinstallez le programme."	A bundle folder is missing under `sqlroot` (default: the exe's folder).	Reinstall, or fix `sqlroot=`.
failed "cette base n'est pas une base Le Sage"	Step 0001's precondition raised `IZ001`: empty or non-Le Sage database.	Wrong `db=`; empty databases come from `lesage/makedb/` only.
failed `foreignowner` (repair query shown, with *Copier*)	A managed function is owned by someone other than `<owner>` / `<owner>_ddl`.	Run the shown `ALTER FUNCTION … OWNER TO` as superuser.
failed with a file and line	An SQL error in that bundle file. Everything was rolled back.	Fix the file (next version if it was released).
failed lock wait	Another PC held the upgrade lock for more than 5 minutes.	Retry; check `pg_stat_activity` (`application_name` = `lesage-delphi 13.000.xx (login@PC)`).

On *refused* and *failed* the user can choose *Lancer quand même* (runs with whatever the database has, some functions may fail) or *Quitter* (default).

Inspecting a database

```
SELECT bundle, version, checksum, dirty, appliedat, appliedby FROM improvize.bundles;
SELECT bundle, fromversion, toversion, kind, appliedat, appliedby
  FROM improvize.bundlehistory ORDER BY id DESC LIMIT 20;   -- kind: upgrade | reinstall |
reload
```

Compare a checksum with the files: `npm run check -- manifest ../lesage/sql` (in `common/`) prints the per-file manifest and the bundle checksum.