HEIRLOOM COMPUTING

Heirloom License System

A plain-language guide to deploying the license server, connecting client applications, and taking a license offline with your laptop when you travel.

Setup Guide · For IT administrators and developers · 2026

Contents

1. What this system does

Heirloom's license system has two halves that work together:

Piece	What it does	Who touches it
The license server this package	A small web application ("thin server") that keeps track of who is allowed to use which product, and how much of it (how many cores, how many developer seats, etc). It's the single source of truth.	IT / server administrator
The client application	Any product built on Heirloom's licensing (your compiler, your runtime, your IDE plugin). It asks the license server "am I allowed to run?" and gets a yes/no plus how much capacity it can use.	Developers, end users

In business terms, this is what keeps entitlements honest without anyone having to police it manually: every check, allocation, and denial is recorded in a tamper-evident audit log, so usage always matches what was actually sold, and that record can be produced as evidence if a customer or auditor ever asks for it.

Normally the client application talks to the license server over the network every time it needs to check its license. But sometimes a developer needs to keep working without a network connection — on a flight, at a client site with no VPN access, and so on. For that, the system supports **"License Borrowing"**: the client asks the server for a temporary offline license that's good for a set number of days, and the server hands back a small self-contained file the laptop can use entirely on its own, no network required, until that time runs out.

> **Nothing to install on the server for this to work.** The offline license is built by the command-line tool on your own machine, not by the server — so there's nothing extra to configure or maintain on the server side to support offline use.

2. What's in this package

File	What it is
`license-thin-client.war`	The license server itself. Deploy this into any standard Java web server (Tomcat is what we test against). It hosts the license-checking API and a read-only web page where you can see current grants and usage.
`license-cli.jar`	A command-line tool for setting up the database, installing licenses, and requesting offline "borrowed" licenses. This is the one tool you'll run from a terminal for every step below.
your database's JDBC driver **not included**	Only needed if the command-line tool doesn't already have your database's driver on its classpath (see step 5) — download whichever driver matches your database. Not needed if someone else has already set up the database.
`examples/license-developer.properties` `examples/license-production.properties`	Two ready-to-edit configuration files a client application can use to find and configure its license automatically, with no code changes. See step 7.

3. Deploy the license server

1. Copy `license-thin-client.war` into your web server's deploy folder. For Apache Tomcat, that's the `webapps` folder:

```
copy license-thin-client.war  <TOMCAT_HOME>\webapps\license-thin-client.war
```

2. Start (or restart) the web server:

```
<TOMCAT_HOME>\bin\startup.bat        (Windows)
<TOMCAT_HOME>/bin/catalina.sh start   (Linux/macOS)
```

3. Confirm it started — open this address in a browser (or run the command below) and look for `{"status":"UP"}`:

```
http://<your-server>:8080/license-thin-client/actuator/health
```

You'll also have a simple web page to browse licenses and usage once data is loaded:

```
http://<your-server>:8080/license-thin-client/monitor
```

Deploying at the "root" of the server instead? Rename the file to `ROOT.war` before copying it in, and drop the `/license-thin-client` part from every address in this guide.

4. Point it at your database

The license server needs one database to store its data (any JDBC-compatible database works — PostgreSQL is what we test against). Tell it where that database lives by setting three values as environment variables *before* the server starts.

For Tomcat, put them in a file called `setenv.bat` (Windows) or `setenv.sh` (Linux/macOS) inside `<TOMCAT_HOME>\bin\`:

```
set "SPRING_DATASOURCE_URL=jdbc:<your-db>://<db-host>:<port>/<db-name>"
set "SPRING_DATASOURCE_USERNAME=<db-user>"
set "SPRING_DATASOURCE_PASSWORD=<db-password>"
```

Then restart the web server so it picks up the change.

> **Didn't set anything?** The server defaults to a local database called `postgres` with username/password `postgres` / `root` — handy for a quick trial, not something to leave in place for real use.

Connection pool size

The server manages its own connection pool (HikariCP) — nothing extra to install. Two pools exist: a small one behind the read-only monitor page, and a separate, larger one behind actual license checks from client applications (grant/allocate/release calls). Both have sensible defaults but can be resized the same way, in `setenv.bat` / `setenv.sh`:

```
set "SPRING_DATASOURCE_HIKARI_MAXIMUM_POOL_SIZE=5"      (monitor page,
default: 5)
set "SPRING_DATASOURCE_HIKARI_MINIMUM_IDLE=1"           (monitor page,
default: 1)
set "SPRING_DATASOURCE_HIKARI_CONNECTION_TIMEOUT=5000"  (monitor page,
milliseconds, default: 5000)
set "LICENSE_POOL_MAX_CONNECTIONS=20"                   (license checks,
default: 20)
```

Using your own connection pool instead (JNDI)

Already running a connection pool for this database in your web server (a Tomcat `<Resource>`)? Point the license server at that pool instead of having it manage its own — declare the pool in Tomcat as usual, then set:

```
set "SPRING_DATASOURCE_JNDI_NAME=jdbc/YourPoolName"
set "LICENSE_POOL_JNDI_NAME=jdbc/YourPoolName"
```

Setting either one makes the URL/username/password/pool-size settings above ignored for that side — the server looks up your existing pool by name instead. Leave both unset (the default) unless you specifically need this; it's simplest to let the server manage its own pool.

5. Create the database tables

Before the server can store anything, its tables need to be created. The command-line tool does this in one step and is safe to run more than once (it won't touch data that's already there):

```
java -jar license-cli.jar init --url "jdbc:<your-db>://<db-host>:<port>/<db-name>" --user <db-user> --password <db-password>
```

What this actually creates: tables for grants (the licenses themselves), live pool allocations (who's currently using what), and the audit log. Running it again on a database that already has these tables is harmless — it only fills in what's missing.

> **"No suitable driver found"?** The command-line tool doesn't ship with every database's JDBC driver bundled in. Download the driver jar that matches your database, keep it next to `license-cli.jar`, and run commands this way instead of plain `-jar`:
>
> ```
> java -cp "license-cli.jar;<your-jdbc-driver>.jar" com.heirloom.license.cli.ImportTool init --url "jdbc:<your-db>://<db-host>:<port>/<db-name>" --user <db-user> --password <db-password>
> ```
>
> Every command in this guide works the same way — just swap in this longer form whenever the plain `-jar` form can't find a driver.

6. Install a license

A "grant" is the actual license — who it's for, what product, how many cores or seats, and when it expires. Your Heirloom account contact will give you either a single grant file, or a ready-made install script (a `.sql` file) that contains everything at once.

> **Needs your database's JDBC driver on the classpath**, same as step 5 — the command-line tool doesn't ship with one bundled in. Both commands below use the `-cp` form for that reason; drop `;<your-jdbc-driver>.jar` if the driver is already on your classpath some other way.

Option A — you were given a single grant file (`.jwt`)

```
java -cp "license-cli.jar;<your-jdbc-driver>.jar"
com.heirloom.license.cli.ImportTool import --url "jdbc:<your-db>://<db-host>:
<port>/<db-name>" --user <db-user> --password <db-password> --grant grant.jwt
```

Option B — you were given an install script (`.sql`)

```
java -cp "license-cli.jar;<your-jdbc-driver>.jar"
com.heirloom.license.cli.ImportTool install --source license-install.sql --url
"jdbc:<your-db>://<db-host>:<port>/<db-name>" --user <db-user> --password <db-
password>
```

Either way, once it's done, the license server will recognize that grant the next time a client application checks in.

Under the hood, a grant file is a cryptographically signed token (a JWT) — the import step verifies that signature before writing anything to the database, so a corrupted or tampered grant file is rejected rather than silently installed.

7. Configure a client application

Each application built on Heirloom licensing looks for a small text file named `license.properties` to know which server to talk to and how many resources to ask for. You don't need to change any code — just drop the right file in place. Every key below is read once, at startup, by the application's Heirloom licensing library — the file itself is never touched by the server.

1. Pick one of the two example files in `examples/`:

File	Use when…
`license-developer.properties`	Setting up a developer machine — every product asks for the same number of cores.
`license-production.properties`	Setting up a production deployment — different core counts per product, with one product metered by transaction volume instead of cores.

This is exactly what the `license.type` key inside the file controls, and it's independent of which example file you started from — you can set it to whichever of the three values actually matches your situation:

`license.type`	Meaning
`developer`	A developer's own machine — typically one grant per product, same core count for everything.
`production`	A real deployment — lets per-workload core counts and metering differ, and is what you'd point at a production grant.
`trial`	Matches a trial-edition grant specifically. Useful when a trial and a production grant for the same product both exist in the same database and need to be told apart.

2. Open it in a text editor and change the `license.server.api` line to point at your server from step 3, for example:

```
license.server.api=http://<your-server>:8080/license-thin-client
```

What "cores" means, and metering by transactions instead

`license.cores` is the amount of capacity the application asks the license server to reserve for it out of the grant's pool — not necessarily a literal CPU core count, just the unit that grant is metered in. `license.metering` says which unit applies: `cores` (the default) or `transactions`. Once metering resolves to `transactions` for a given workload, `license.cores` is not read for it at all — the two are mutually exclusive, matching how a grant itself is either core-metered or transaction-metered, never both.

Per-workload overrides

A single properties file can serve every product on a machine, because each workload type can override the generic `license.cores` / `license.metering` with its own `license.<TYPE>.cores` / `license.<TYPE>.metering`. A workload with no override of its own just falls back to the generic value:

Workload	What it is	Typically metered by
`HSORT`	High-speed sort	Cores
`BATCH`	Batch processing	Cores
`RUNTIME`	General runtime execution	Cores
`ETP` (Elastic Transaction Platform)	Transaction-driven runtime workload	Transactions

```
license.cores=8
license.metering=cores

license.ETP.metering=transactions   # ETP is metered by transaction volume,
not cores
license.HSORT.cores=24              # HSORT asks for 24 cores instead of the
generic 8
license.BATCH.cores=16              # BATCH asks for 16 cores instead of the
generic 8
                                     # RUNTIME has no override here, so it
uses the generic 8
```

> **ETP isn't a separate license type on the server.** It's a client-side convenience for a transaction-metered `RUNTIME` product — the grant itself is stored and enforced as `RUNTIME`. You still write `license.ETP.*` in the properties file; the application's licensing library maps it to the right place internally.

Working from an offline (H2) license instead of the server

If there's exactly one `.mv.db` file sitting in the application's working directory or home directory, it's picked up automatically — no property needed, whatever it's named. If more than one is present, name the one you want used exactly `license-checkout.mv.db` (step 8's "License Borrowing" output already uses this name) so the application knows which one you mean — with more than one present and none matching that name, nothing is picked up automatically and this needs to be set explicitly. To point at an offline database somewhere else, set `license.h2.db` to its path, without the `.mv.db` suffix:

```
license.h2.db=C:\licenses\license-checkout
```

You can set both `license.server.api` and an offline database (found automatically or via `license.h2.db`) in the same file. The application tries the server first and falls back to the offline file only if the server can't be reached — handy for a laptop that's sometimes connected and sometimes not.

3 Rename the file to exactly `license.properties` and place it in one of these locations. The application checks them in this exact order and stops at the first one it finds:

Order	Location	Notes
1	Wherever the `-Dheirloom.licenseproperties=<path>` JVM system property points, if set	An explicit override — points at the file directly, any name, any folder. If set but the file isn't there, this is a hard error rather than falling through to the next location.
2	The application's current working directory — i.e. `license.properties` sitting right next to (or in the same folder you launch from as) the application's jar/executable	The most common choice — this is the folder you're `cd`'ed into (or that a service/shortcut is configured to start in) when the application runs.
3	The current user's home directory (`%USERPROFILE%` on Windows, `$HOME` on Linux/macOS)	Good for a setting that should apply no matter which folder the application is launched from, for one particular user account.

Nothing found in any of these? That's not an error — it just means every setting needs to come from somewhere else (environment, JNDI, or explicit configuration in code). Only an explicit `-Dheirloom.licenseproperties` path that doesn't exist actually fails.

8. Working offline ("License Borrowing")

When a developer knows they'll be without network access for a while, they can "check out" a temporary offline license before disconnecting. This is a single command:

```
java -jar license-cli.jar checkout --days 5
```

If a `license.properties` file is already in place (step 7), the command automatically knows which server to ask and how many cores to request — `--days` is the only thing you always have to specify yourself, since it's a deliberate choice each time.

This single command:

1. Asks the license server for a 5-day (in this example) offline license
2. Builds a small local license database from the server's response — no extra steps
3. Prints out exactly what was included, so you can confirm it looks right

> **Nothing to configure on the server for this.** The offline database is built entirely on the developer's own machine by the command-line tool — the server just answers the request, it doesn't need any special database driver or extra setup to support this.

Once it's done, you'll see a file called `license-checkout.mv.db` in the current folder. That's the offline license — a small embedded (H2) database, not a connection to anything. The client application reads it directly, so it works with no server reachable at all, right up until the days you asked for run out.

9. Command cheat-sheet

Command	What it does
`license-cli.jar init --url <jdbc> ...`	Create the database tables (step 5)
`license-cli.jar import --url <jdbc> ... --grant <file>`	Install a single license file (step 6)
`license-cli.jar install --url <jdbc> ... --source <file>`	Install from a ready-made `.sql` / `.mv.db` package (step 6)
`license-cli.jar checkout --days <n>`	Request an offline license good for `n` days (step 8)
`license-cli.jar verify --url <jdbc> ...`	Double-check an installed license is valid
`license-cli.jar audit-verify --url <jdbc> ...`	Confirm the audit log hasn't been tampered with
`license-cli.jar info`	Show this machine's hardware details (useful for support requests)
`license-cli.jar help`	Show the full list of commands and options

10. Troubleshooting

The health check doesn't return "UP"

The web server probably hasn't finished starting, or failed to start. Check the web server's own log folder for errors — most often this means the database connection details from step 4 are wrong.

"No suitable driver found" when running a command

Use the longer command form shown in step 5, with your database's JDBC driver jar included on the classpath.

A client application says it can't find a license

Check three things, in order: (1) is `license.properties` actually named exactly that and sitting in the right folder, (2) does `license.server.api` in that file point at a server that's actually running, (3) has a matching license actually been installed on that server (step 6) for the product/edition the application is asking for.

Checking out an offline license fails

This usually means either the license server couldn't be reached (check `license.server.api`), or there's no developer-edition license available on the server to check out from. An administrator can confirm which licenses are installed via the monitor page from step 3.