Backend Software Engineer (PHP/Python)

Test Case - Centralized License Service

Context

group.one is a fast-growing group that acquires and integrates multiple brands into a unified ecosystem. We are bringing several WordPress-focused products closer together (WP Rocket, Imagify, RankMath, BackWPup, RocketCDN, WP.one, etc.). A key enabler is knowing, from a single source of truth, which products a given person can access. To that end, we need a centralized License Service.

This service becomes the authority for license lifecycle and entitlements across brands. Brand-specific systems keep managing users, payments/subscriptions, and billing, and they integrate with the License Service to provision, update, and query licenses. End-user products (plugins, apps, CLIs) call the License Service to activate/validate their usage.

Your priority is to demonstrate strong backend architecture and thoughtful interfaces, while also showing mastery in writing code. You should demonstrate your capacity to design a complex system, and that you are able to implement it following the relevant best practices for this use case.

The recommended time allocation for this test is 4 to 6 hours.

Expected outcome

You have to design and implement a multi-tenant License Service used by multiple brands that:

- Acts as the single source of truth for licenses and entitlements.
- Exposes clear, brand-integrable APIs to provision and manage licenses.
- Exposes product-facing APIs to activate/check licenses (and optionally manage seats).
- Is designed for scalability, multi-brand support, and future extensibility.
- Is observable and operable in a production context.

Important: The system requires multiple features. In your documentation, describe and design the full system, but it is acceptable to implement only a subset of the most important/core features in code. For example, managing multiple seats can be fully designed (data model, flows, API) without being fully implemented.

User Stories

Design for all stories; implement the core ones as indicated below. The specifications are not extensive, on purpose. If needed, make the opinionated choices that you would recommend in a real-life situation and discuss the trade-offs in your documentation.

No graphical user interface are required for this exercise. The system must be operated through an API.

US1: Brand can provision a license

- As a brand system, I can create license keys and licenses for a customer email and associate them together to grant access to one or more products (e.g., a single license key may unlock multiple licenses).
- Through the service, brands must be able to generate a license key, create licenses, and associate those licenses with the license key. Brands should then be able to get the license key so that it can be transmitted to end users.
- Each license must be related to a single product, have a status (valid, suspended, cancelled), and an expiration date.
- As an example, the designed system must allow for the following scenario:
    - A new user just bought a RankMath subscription on rankmath.com, so a RankMath license & license key n°1 must be created.
    - Later, this user purchases a Content AI subscription: Content AI is an addon in RankMath, unlocking additional features. The RankMath brand wants its users to have access to all their RankMath-related subscriptions through a single license key, so a Content AI license must be created and associated with the already existing license key n°1.
    - Finally, the user purchases a WP Rocket license on wp-rocket.me. This is a different brand and therefore, we want this license on a different, new license key n°2.

US2: Brand can change license lifecycle

- As a brand system, I can renew (extend), suspend/resume, or cancel a license.

US3: End-user product can activate a license

- As an end-user product, I can activate a license for a specific instance (e.g., site URL, host, machine ID), potentially consuming a seat if applicable.
- The service enforces seat limits if implemented; if not implemented, describe your intended approach.

US4: User can check license status

- As an end-user product or customer, I can check the status and entitlements of a license key to know if it's valid, what it provides access to, and if applicable, the number of remaining seats.

US5: End-user product or customer can deactivate a seat

- As an end-user product or customer, I can deactivate a specific activation (freeing a seat) for an instance.

US6: Brands can list licenses by customer email across all brands

- As a brand, I can list all licenses associated with a given email across the entire ecosystem.
- End users & external parties should not have access to this list.

Minimum Implementation (Core Slice)

We recommend implementing at least US1, US3, US4 and US6.

You may choose to also implement Story 2 and/or Story 5 if time permits.

Technical Expectations

It's okay not to meet every criterion as long as gaps are acknowledged and discussed in your documentation. Because this is a critical system, please justify your choices and trade-offs.

- Architecture and design
    - Clearly model multi-tenancy across brands and products.
    - Define integration points and how brand systems and end-user products should use them.

    - Provide a comprehensive design of the complete service, including the technical architecture and the data model.
- Observability, operability and error handling
  Design, describe, and if possible implement observability, operability, and error handling features that are relevant for such a system: The system should be observable, resilient, and easily monitored.
- Repository
    - Public GitHub repository with commit history.
- Code
    - PHP or Python. Django is recommended for Python; other frameworks (e.g., FastAPI, Flask, Laravel, Symfony) are acceptable.
    - Clear setup and run instructions. Docker is optional but welcome.
    - Built with modern OOP.

Documentation (Explanation.md)

Include an Explanation.md at the root of your repo that covers:

- Problem and requirements in your own words
- Architecture and design (see Technical expectations)
- Trade-offs and decisions
    - Alternatives considered and why you chose your approach.
    - Scaling plan and how this could evolve.
- How your solution satisfies each User Story
    - Call out which parts are implemented vs. designed-only.
    - If some requirements are not met, state it clearly and explain your decision.
- How to run locally
    - Step-by-step commands, environment variables, sample requests (cURL/Postman).
- Known limitations and next steps

🎖 Bonus Points (Optional)

- Your codebase passes a linter/code style inspection.
- Your codebase has meaningful unit and integration tests.
- Your GitHub repository has a CI automatically checking linter and tests on every PR and commits to the main branch.

- Your codebase follows our public engineering best-practices.

Questions?

Email us. It’s okay to ask. We’re happy to help you.