CYBERSHIP

Carrier Integration Service

Backend Engineering — Take-Home Assessment

Overview

At Cybership, we integrate with shipping carriers to provide our customers with real-time rates, labels, tracking, and more. For this assessment, you will build a **shipping carrier integration service** in TypeScript that wraps the UPS Rating API to fetch shipping rates.

The service should be designed as a real, maintainable production module — one our team would extend over time to support additional carriers (FedEx, USPS, DHL) and additional operations (label purchase, tracking, address validation).

You are **not** expected to have a working UPS API key or to make live API calls. We want to see how you architect, type, and implement the integration even without the ability to hit the real endpoint. Your code should be structurally correct and demonstrate that you understand the API based on its documentation.

Time & Reference

Please spend approximately **2–4 hours**. We value quality and thoughtfulness over completeness. If you run out of time, leave comments explaining what you would do next. The UPS Rating API docs are at:

```
https://developer.ups.com/tag/Rating?loc=en_US
```

Requirements

1 Rate Shopping

Accept a rate request (origin, destination, package dimensions/weight, optional service level) and return normalized rate quote(s). The caller should never need to know anything about UPS's raw request or response format.

2 Authentication

Implement the UPS OAuth 2.0 client-credentials flow: token acquisition, caching/reuse of valid tokens, and transparent refresh on expiry.

3 Extensible Architecture

Adding a second carrier or a second UPS operation should not require rewriting existing code. We should see a clear pattern for how new carriers and operations plug in. Do **not** hardcode to only support the single rate endpoint.

4 Configuration

All secrets and environment-specific values must come from environment variables or a configuration layer — never hardcoded. Include a `.env.example`.

5 Types & Validation

Strong TypeScript types and runtime validation schemas for all domain models (requests, responses, addresses, packages, errors). Validate input before making any external call.

6 Error Handling

Handle realistic failure modes: network timeouts, HTTP error codes, malformed responses, rate limiting, auth failures. Errors returned to the caller should be meaningful and structured.

7 Integration Tests

This is a critical requirement. Write integration tests that exercise your service's logic end-to-end using **stubbed API responses** based on payloads from the UPS documentation. These tests should verify that:

- Request payloads are correctly built from your domain models.
- Successful responses are parsed and normalized into your internal types.
- Auth token lifecycle works (acquisition, reuse, refresh on expiry).
- Error responses (4xx, 5xx, malformed JSON, timeouts) produce the expected structured errors.

Stub the HTTP layer and feed it realistic payloads so we can confirm the processing logic — parsing, mapping, validation, and error handling — all works as expected without a live API.

Evaluation Criteria

Criteria	What We Look For
Architecture & Extensibility	Clean separation of concerns. Could we add FedEx without touching UPS code?
Types & Domain Modeling	Well-defined domain objects. Clear boundary between internal and external API shapes.
Auth Implementation	Token lifecycle management transparent to the caller.
Error Handling	Structured, actionable errors. No swallowed exceptions.
Integration Tests	Stubbed end-to-end tests proving request building, response parsing, and error paths.
Code Quality	Readability, naming, idiomatic TypeScript. Comments where intent isn't obvious.

Deliverables

1. A GitHub repository — create a repo under your own account and share the link.
2. A `README.md` explaining your design decisions, how to run the project, and what you would improve given more time.

3. A `.env.example` listing required environment variables.

Important Notes

Language: TypeScript is required for this assessment.

No API key required: You will not have UPS developer credentials. Stub or mock HTTP calls as needed.

No UI needed: This is a backend/service-layer exercise. A test suite or simple CLI demonstrating usage is sufficient.

Good luck — we're excited to see your approach. If you have any questions, reach out to Jack at `jack@cybership.io`.