Backend engineer pre-liminary evaluation

Contents

[Backend Coding Exercises 2](#_Toc195539167)

[1. Java Exercises 2](#_Toc195539168)

[1.1 Advanced Stream Operations & var: 2](#_Toc195539169)

[1.2 Records & Pattern Matching: 2](#_Toc195539170)

[1.3 Concurrency & CompletableFuture 2](#_Toc195539171)

[2. Quarkus Exercises 3](#_Toc195539172)

[2.1 REST Endpoint with Path/Query Params & CDI 3](#_Toc195539173)

[2.2 Persistence with Panache 3](#_Toc195539174)

[2.3 MicroProfile Health Check 3](#_Toc195539175)

[3 Python Exercises 4](#_Toc195539176)

[3.1 Simple API with Flask/FastAPI 4](#_Toc195539177)

[3.2 Data Processing with Dictionaries/Lists 4](#_Toc195539178)

[3.3 Interacting with an External API 4](#_Toc195539179)

[3.4 Database Interaction (Conceptual/SQLAlchemy) 4](#_Toc195539180)

[4 Helm Exercises 5](#_Toc195539181)

[4.1 Conditionals and Loops in Templates 5](#_Toc195539182)

[4.2 Named Templates (\_helpers.tpl) 5](#_Toc195539183)

[4.3 Chart Dependencies 5](#_Toc195539184)

[4.4 Helm Hooks 6](#_Toc195539185)

Work Experience responses:

1. Designed and implemented high throughput microservices using Java 17/21 and Quarkus, reducing latency by X% and improving system resilience.
2. Leveraged Quarkus features like native compilation and RESTEasy Reactive to optimize application startup time and memory footprint for cloud deployments.
3. Engineered and maintained complex Helm charts for deploying multi-service applications onto Kubernetes, standardizing deployment processes and reducing manual effort by Y%.
4. Integrated applications with various data stores (PostgreSQL, Kafka, Redis) ensuring data consistency and performance

# Backend Coding Exercises

## Java Exercises

### 1.1 Advanced Stream Operations & var:

* Task: Given a List<Product> (where Product has fields like name, category, price), use Java 11+ features (var for brevity if desired) and the Stream API to:
  + Group products by category.
  + For each category, calculate the average price of products in that category.
  + Return a Map<String, Double> where the key is the category and the value is the average price.
* Focus: groupingBy(), averagingDouble(), var.

### 1.2 Records & Pattern Matching:

* Task:
  + Define a record Payment(double amount, String currency) to represent a payment.
  + Create a sealed interface Currency with permits USD, EUR, GBP {}. Define record USD() implements Currency, record EUR() implements Currency, record GBP() implements Currency.
  + Write a method String processPayment(Payment payment, Currency targetCurrency) that uses pattern matching (either instanceof or switch) on targetCurrency to simulate a currency conversion message (e.g., "Converting [amount] [currency] to USD/EUR/GBP").
* Focus: Records, sealed types, pattern matching for instanceof or switch.

### 1.3 Concurrency & CompletableFuture

* Task: Simulate fetching data from two independent remote services asynchronously. Write a method that takes two user IDs, calls two hypothetical CompletableFuture<String> fetchUserData(long userId) methods concurrently, and returns a CompletableFuture<String> that completes with a combined string (e.g., "Data1: [result1], Data2: [result2]") once *both* underlying futures complete. Handle potential exceptions from either fetch operation gracefully.
* Focus: CompletableFuture.supplyAsync(), thenCombine(), exceptionally().

## Quarkus Exercises

### 2.1 REST Endpoint with Path/Query Params & CDI

* Task:
  + Create a simple CDI bean (e.g., GreetingService) with a method String greet(String name).
  + Inject GreetingService into your REST resource.
  + Modify the /hello endpoint (or create a new one like /greet/{name}) to accept a name as a path parameter and an optional query parameter suffix.
  + The endpoint should use the GreetingService to return a personalized greeting (e.g., "Hello [Name][Suffix]!").
* Focus: CDI (@ApplicationScoped, @Inject), @PathParam, @QueryParam.

### 2.2 Persistence with Panache

* Task:
  + Define a simple JPA entity (e.g., Person with name and birthDate fields).
  + Create a Panache repository for Person (PersonRepository implements PanacheRepository<Person>).
  + Add REST endpoints to:
    - Create a new Person (POST /persons).
    - List all Persons (GET /persons).
    - Find a Person by name (GET /persons/search/{name}).
  + Configure a suitable database (e.g., H2 for dev mode).
* Focus: JPA entities, Panache repository patterns (persist(), listAll(), find()), JAX-RS for CRUD, Dev Services/DB config.

### 2.3 MicroProfile Health Check

* Task: Implement a custom MicroProfile Health check (@Liveness or @Readiness) for your Quarkus application. For example, create a check that simulates verifying a connection to a downstream service or checks database connectivity status. Ensure the check appears on the /q/health endpoint.
* Focus: @Liveness/@Readiness, HealthCheck, HealthCheckResponse.

## 3 Python Exercises

### 3.1 Simple API with Flask/FastAPI

* Task: Create a very basic web API using either Flask or FastAPI that has one endpoint /items/{item\_id}. This endpoint should return a JSON response like {"item\_id": <item\_id>, "description": "Sample Item"}.
* Focus: Basic Flask/FastAPI setup, routing, returning JSON.

### 3.2 Data Processing with Dictionaries/Lists

* Task: You are given a list of dictionaries, where each dictionary represents a user ({'id': 1, 'name': 'Alice', 'role': 'admin'}). Write a Python function that takes this list and returns a dictionary where keys are roles and values are lists of names belonging to that role.
* Focus: Looping, dictionary manipulation (setdefault or defaultdict), list appending.

### 3.3 Interacting with an External API

* Task: Write a Python function that uses the requests library to fetch data from a public JSON API (e.g., https://jsonplaceholder.typicode.com/posts/1). The function should handle potential requests exceptions (like connection errors) and return the JSON response body as a Python dictionary if successful, or None if an error occurs.
* Focus: requests library (get), exception handling (try...except), JSON parsing (response.json()).

### 3.4 Database Interaction (Conceptual/SQLAlchemy)

* Task: (Conceptual or Code) Describe how you would use an ORM like SQLAlchemy in Python to:
  + Define a model representing a Product table (with id, name, price columns).
  + Write code to query the database for all products with a price greater than 100.0.
* Focus: ORM concepts, model definition (declarative base), basic querying (session.query, filter).

## 4 Helm Exercises

### 4.1 Conditionals and Loops in Templates

* Task:
  + Add a boolean value service.enabled (default: true) to values.yaml.
  + Wrap the entire service.yaml template content within an {{ if .Values.service.enabled }} block.
  + Add a list of extraEnvVars in values.yaml (e.g., [{ name: "VAR1", value: "VAL1" }]).
  + Modify deployment.yaml to loop through extraEnvVars using {{ range }} and add them as environment variables to the container spec.
* Focus: Helm control structures (if, range), accessing nested values.

### 4.2 Named Templates (\_helpers.tpl)

* Task:
  + Define a named template in \_helpers.tpl called my-app.fullname that generates a standard full name for resources (e.g., {{ .Release.Name }}-{{ .Chart.Name }}).
  + Define another named template my-app.labels that generates a standard set of labels (e.g., app: {{ template "my-app.fullname" . }}).
  + Use these named templates within deployment.yaml and service.yaml for resource names and labels.
* Focus: Defining (define) and using (template, include) named templates, DRY principles.

### 4.3 Chart Dependencies

* Task:
  + Assume your my-app chart requires a database like PostgreSQL. Add a dependency to the official Bitnami PostgreSQL chart in your Chart.yaml.
  + Configure the dependency in values.yaml to disable the PostgreSQL service exposure if your app connects internally (e.g., set postgresql.service.type=ClusterIP or similar, depending on the subchart's values).
  + Explain how you would pass configuration (like the database host generated by the subchart) from the PostgreSQL subchart to your my-app deployment template.
* Focus: dependencies in Chart.yaml, managing subchart values, accessing subchart outputs (global values or lookups - conceptual understanding is key).

### 4.4 Helm Hooks

* Task: Describe a scenario where you would use a Helm hook (e.g., pre-install, post-upgrade). Write a simple example manifest (e.g., a Job) annotated as a post-install hook that prints a message indicating the installation is complete.
* Focus: Understanding hook use cases, hook annotations (helm.sh/hook), hook weights, hook deletion policies.