SCS 3203 Middleware Architecture

JavaEE Assignment

Index No: - 18001149

W.P Pallewatta

By University of Colombo School of Computing, **(Submission Date: -** November 12th)

Table of Contents

Part 1

For the years JavaEE has evolved to a greater extent. One of the recent highlights is that the **Oracle Inc.** deciding to giveaway the rights for **JEE** to the **Eclipse Foundation** and getting the new name of **Jakarta EE**. Thus, here onwards the terms *JEE* and *JakartaEE* will be used interchangeably.

Question 1

JEE Server is considered to be a main component of JakartaEE specification. Name and briefly explain two main container types of a JEE Server and artifacts being managed in each container type (if required, you may use graphical representation)

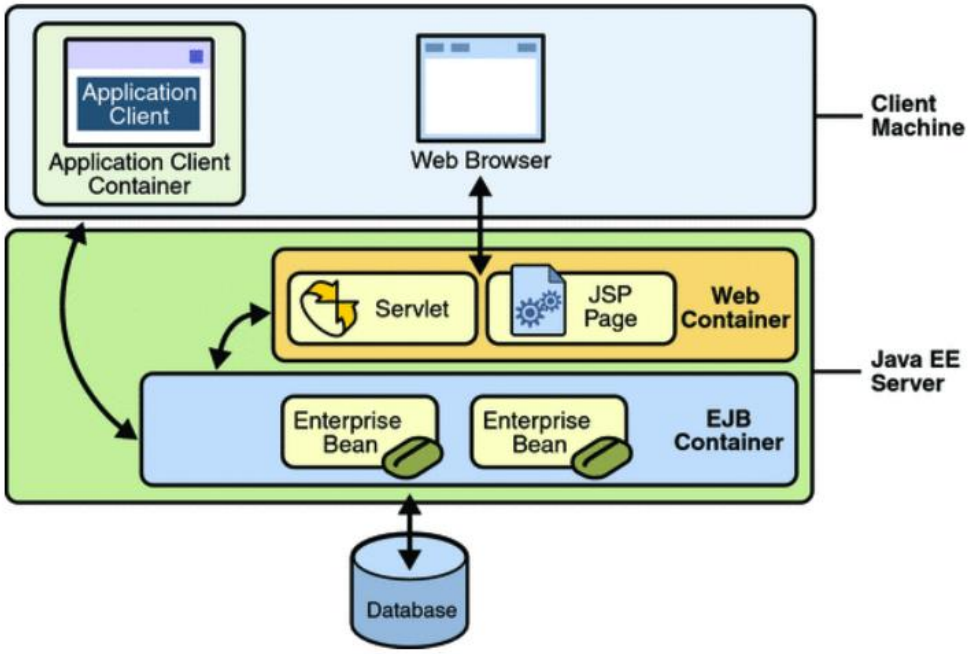


Figure 1- Java EE Server and Containers

Java EE server: The runtime portion of a Java EE product. A Java EE server provides EJB and web containers.

In the Java EE Server there are three main containers to carryout the fucntionalities in Java EE server which are ,

1) Enterprise JavaBeans (EJB) container:
2) Web container:
3) Application client container:

Let's provide basic introdcution of two main containers in **Java EE** which are vital in order to fucntion the server.

1) Enterprise JavaBeans (EJB) container

Introduction

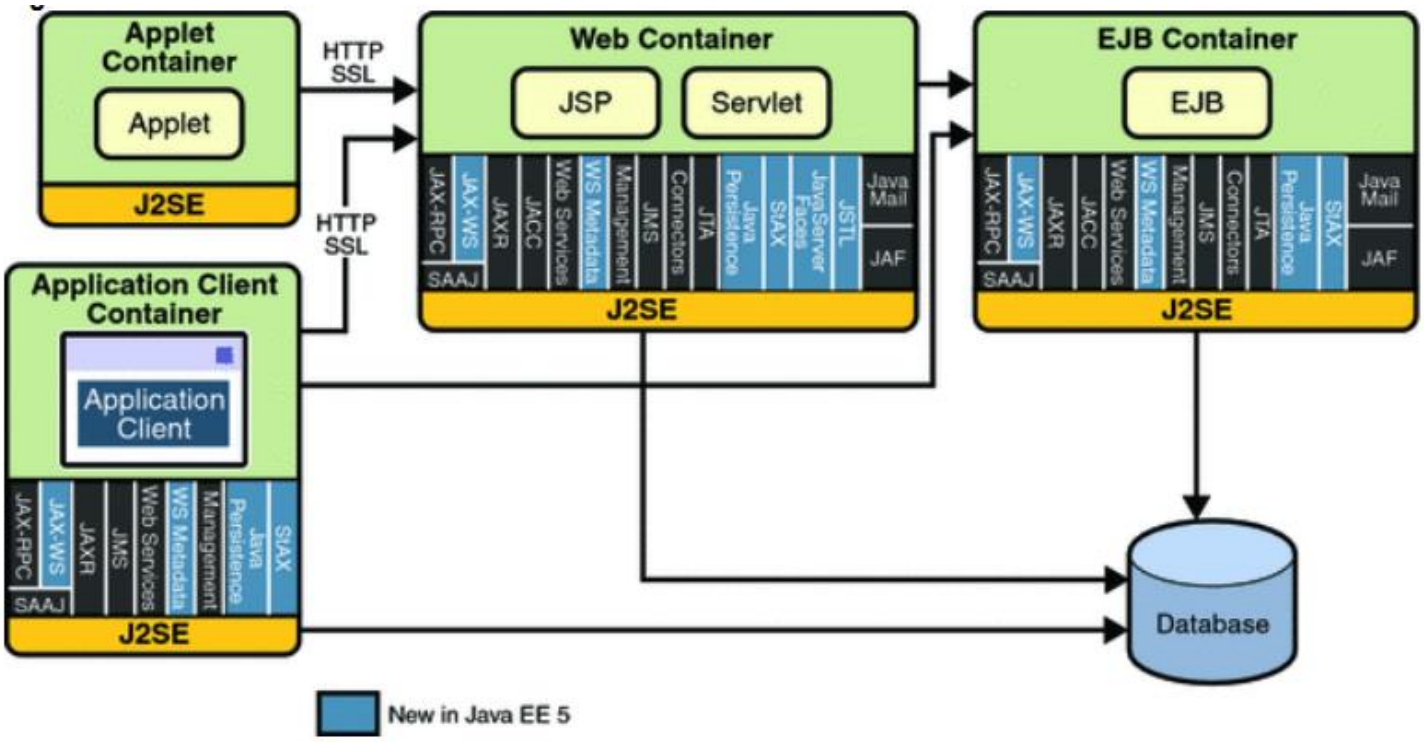
Manages enterprise bean execution for Java EE applications. The Java EE server hosts enterprise beans and associated containers.

Functionalities & Artifacts handling in EJB

An **Enterprise JavaBeans (EJB) container**, often known as an enterprise bean, is a set of fields and methods used to create business logic modules. An enterprise bean can be thought of as a building block for executing business logic on the Java EE server, and it can be used alone or in conjunction with other enterprise beans.

Enterprise beans are classified into two types: **session beans and message-driven beans.** A **session bean** symbolizes a brief interaction with a customer. When the client completes its execution, the session bean and its data are destroyed. A **message-driven bean** combines session bean and message listener characteristics, allowing a business component to accept messages asynchronously. These are typically Java Message Service (JMS) messages.

Entity beans have been replaced with Java persistence API entities in Java EE 5. An entity is a piece of persistent data that is kept in a single row of a database table. If the client or server quits, the persistence manager guarantees that the entity data is preserved.



Java EE Server with four main Containers with its own sets of Artifacts

2) Application Client Container

Introduction

The application client container serves as a bridge between Java EE application clients (special Java SE programs that employ Java EE server components) and the Java EE server. The application client container runs on the client computer and serves as a bridge between the client application and the Java EE server components that it employs.

Functionalities & Artifacts handling in ACC

Consists of a collection of Java classes, libraries, and other files that are necessary for and supplied with Java client programs that run in their own **Java Virtual Machine (JVM).** The ACC handles the execution of Java EE application **client components** (application clients), which are used to access a range of Java EE services (such as JMS resources, EJB components, web services, security, and so on) from a JVM that is not part of the Oracle Glassfish Server.

The ACC connects with the **GlassFish** Server using RMI-IIOP protocol and handles RMI-IIOP communication details via the client ORB that is included with it. The ACC is a lightweight Java EE container when compared to other Java EE containers.

When in Application Client Debugging, these steps taken to initialize the procedure.

➔ When the `appclient` script performs the java command to start the Application Client Container (ACC), which then starts the client, it includes the value of the VMARGS environment variable on the command line. You can set this variable to whatever value you choose. As an example:

```
VMARGS=-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=8118
```

Question 2

JEE Servers differ from the Java Web Servers **(i.e. Servlet Containers).** Name at least two for each " JEE Server" and "Java Web Servers" products available in the market

JEE Server is a server application that implements Java EE platform APIs and offers standard Java EE services. Java EE servers are also known as application servers because they enable you to deliver application data to clients in the same way that web servers serve web pages to web browsers.

Java Web Servers is a computer software (written in Java) that automatically distributes web pages as they are requested. The web server's primary goal is to store, process, and distribute web pages to users. The Hypertext Transfer Protocol is used for this intercommunication (HTTP).

Let 's talk about some of the main JEE servers and JWS available and popular in market.

JEE Server Products Examples

1) **Open Liberty**

- Build cloud-native apps and microservices while running only what you need. Open LibertyTM is the most flexible server runtime available to JavaTM developers in this solar system.

2) **Payara Server Community**

Payara Server was created in 2014 as a fork of GlassFish Server Open-Source Edition as a drop-in replacement. It was published in October 2014 in reaction to Oracle's news that commercial support for GlassFish will be discontinued. [3] Payara Services Ltd now provides commercial support and enterprise services to Payara Server users.Payara Server is based on the upstream **Glassfish** source tree, with Payara-specific additions and fixes.

3) **Apache TomEE**

Apache TomEE is available in four flavors: Web Profile, MicroProfile, Plus, and Plume. Servlets, JSP, JSF, JTA, JPA, CDI, Bean Validation, and EJB Lite are all provided by Apache TomEE Web Profile. MicroProfile support has been added to Apache TomEE MicroProfile. JMS, JAX-WS, and other features are added to Apache TomEE Plus and Plume.

Java Web Servers (Examples)

1) **Apache Tomcat**

- Apache Tomcat is a well-known web server in the Java environment. Sun Microsystems created Apache Tomcat before transferring it to the Apache Software Foundation in 1999. By the way, Tomcat typically runs on port 8080 and supports PHP, ASP.net, Perl, Python, and other programming languages.

2) **Jetty**

- The Eclipse Foundation is responsible for the development of the Jetty web server.

Because it is so small, it can be readily integrated into devices, frameworks, and application servers. Jetty is used in a variety of products, including **Apache ActiveMQ, Eclipse, Google App Engine, Apache Hadoop**, and Atlassian Jira.

3) **Oracle iPlant Web Server**

Oracle iPlanet Web Server (OiWS) is a web server that is intended for medium and large corporate applications. Oracle iPlanet Web Server is a successor **of Sun Java System Web Server**, Sun ONE Web Server, iPlanet Web Server, and Netscape Enterprise Server.

Question 3

Name and **briefly explain** what different Enterprise Java Bean types are available and how they **differ** in JakartaEE 8 or higher specification.

Enterprise Java Bean Types

(1) Session Bean

A Session Bean is an Enterprise Bean that is created for each client session and ends when the client closes. The Session Bean's lifespan is limited to the time between the start and termination of the user's use of the system. Stateless Session Beans, Stateful Session Beans, and Singleton Session Beans are the three types of Session Beans. (Bean is a Java Class)

A) Stateless Session Bean
B) Stateful Session Bean
C) Singleton Session Bean

(2) Entity Bean

An Entity Bean expresses the entity and must be saved (persisted) in the database as a requirement. As a result, even after the client exits, the Entity Bean's state remains in the database. This Enterprise Bean's lifespan is longer than that of a Session Bean. The EJB standards specify the following two management models:

A) BMP (Bean Managed Persistence)

This is a model for managing Enterprise Bean business methods' data persistence. The Enterprise Bean developer must implement processes such as connecting to the database and assembling and executing SQL statements.

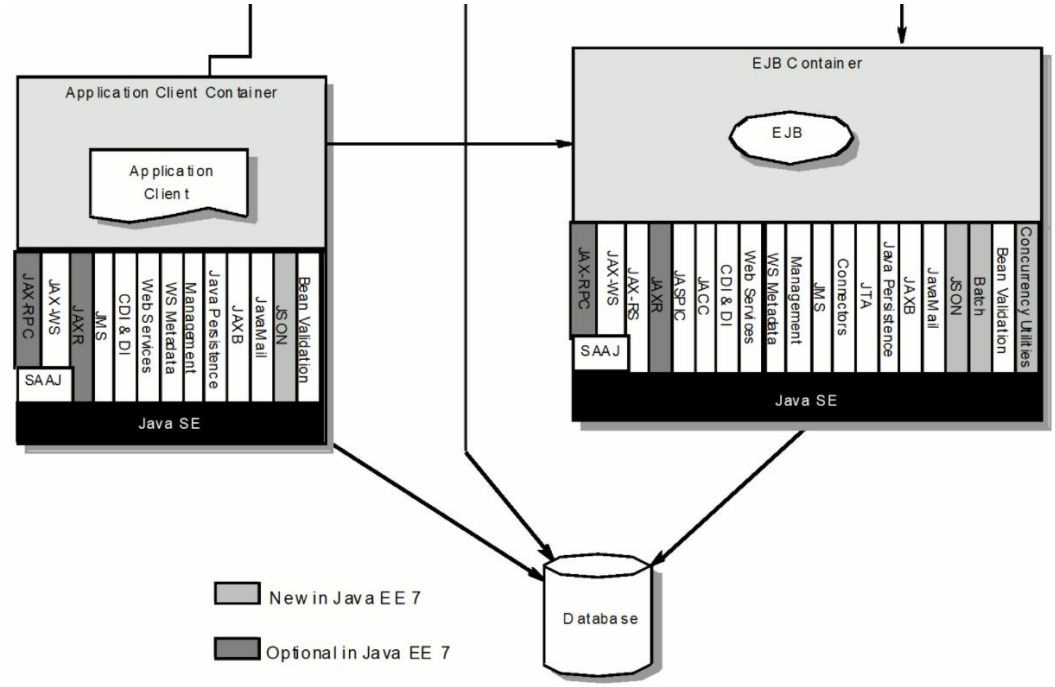
B) CMP (Container-Managed Persistence)

This is a paradigm in which the data persistence is managed by the EJB container. Because the EJB container performs activities such as connecting to databases and saving data, these procedures do not need to be performed by the Enterprise Bean's business methods.

(3) Message-driven Bean

A Message-driven Bean is one that works with JMS. When a JMS message is received from the JMS Destination, the EJB container executes a Bean. Unlike a Session Bean or an Entity Bean, the Message-driven Bean cannot be invoked directly from the client since it lacks a home interface and a component interface.

EJB differ in Jakarta 8 or higher specification



Jakarta EE Architecture Diagram

A Jakarta EE server is a server application that uses the Jakarta EE platform APIs to deliver typical Jakarta EE services. Jakarta EE servers are also known as application servers because they let you to deliver application data to clients in the same way that web servers serve web pages to web browsers.

Let's Find out the Jakarta EE 8 version's EJB differ from the others from below Comparison.

Specifically, Before Jakarta EE 8 & Earlier versions these 3 types of EJB were established and evolved with technology, security, and additional functionalities.

1) Difference Between Session and Entity Bean

Basis of Comparison	Session Bean	Entity Bean
Stateless/ Stateful	It may be stateful or stateless.	It is stateful.
Performance	It depicts a single customer conversation.	It encapsulates persistence business data.
Persistence of Data	It simply saves data for the duration of the discussion with the customer.	Persistence that extends beyond the life of a client instance. Persistence may be managed in two ways: container-managed or bean-managed.
Recovery	It cannot be recovered if the EJB server crashes. As a result, it may be destroyed.	If a failure occurs, it is recoverable.

Transformation to Java EE to Jakarta EE 8

Java EE now has a new home and a new identity. The project was transferred from Oracle to the Eclipse Foundation, and it is now known as Jakarta EE, and it is part of the Eclipse Enterprise for Java (EE4J) project. **On September 10, the Eclipse** Foundation launched Jakarta EE 8, and in this post, we'll look at what it implies for corporate Java.

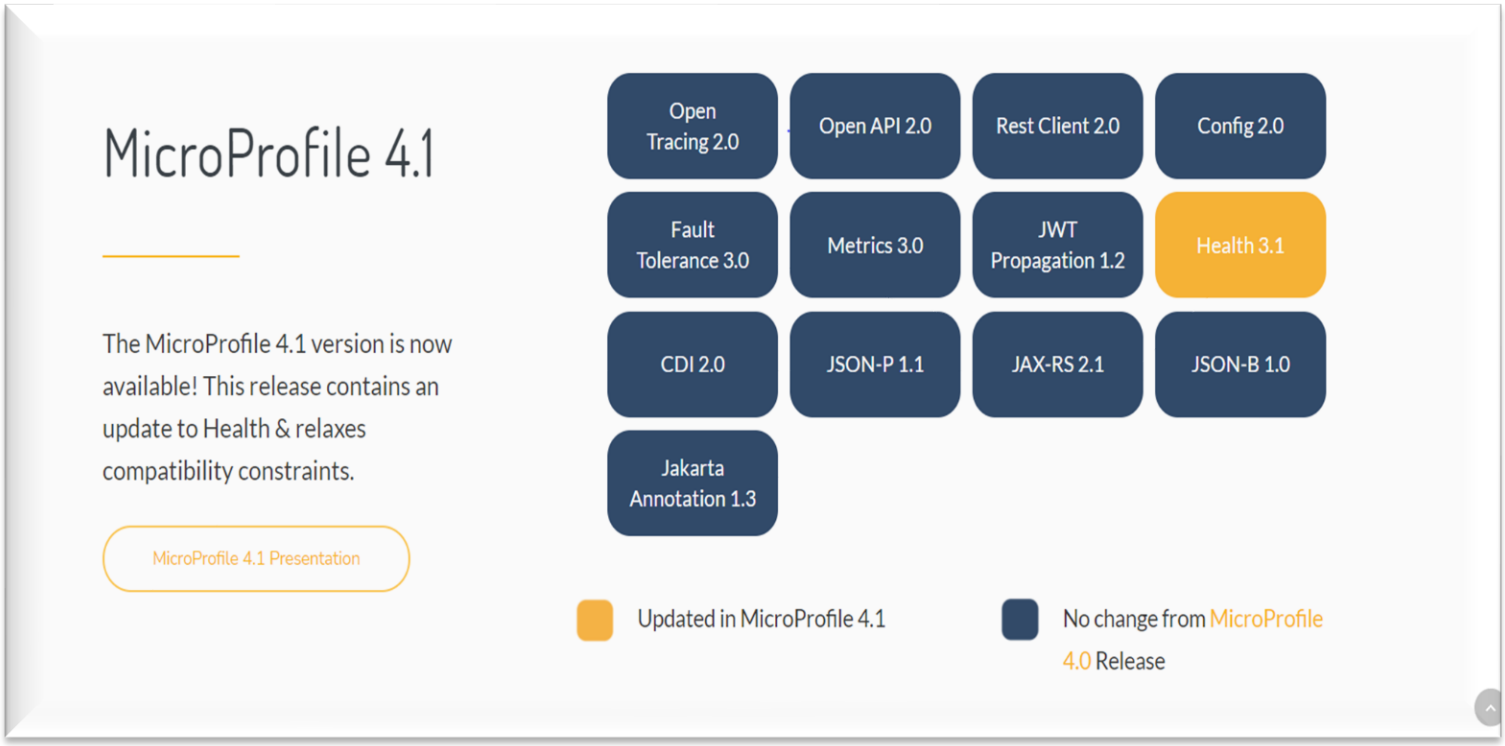
Why Jakarta EE 8 Differ from Java EE?

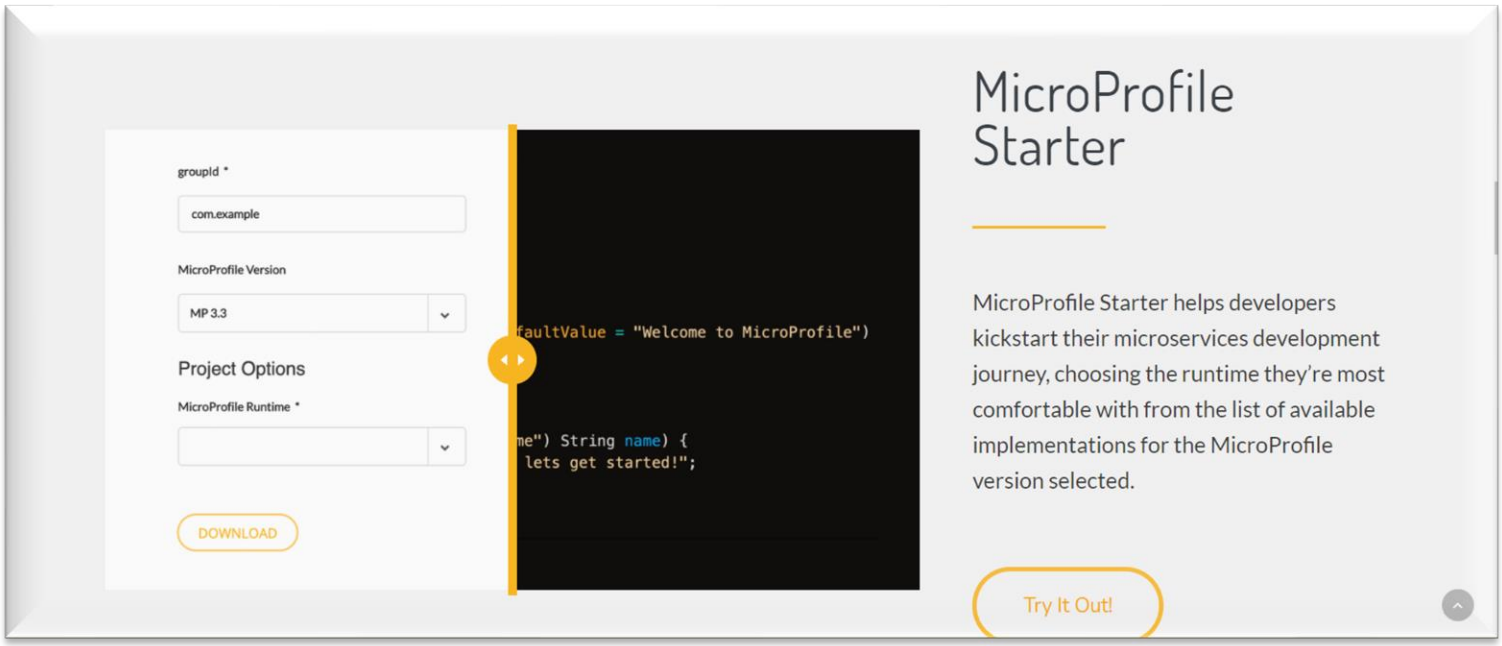
The Java ecosystem is shifting its attention to **cloud computing,** and Jakarta EE is a crucial component of that strategy.

Jakarta EE 8 has the same set of Specification as Java EE 8, with no differences in functionality. The only difference is the Jakarta EE 8 marks a watershed moment in Java enterprise history by incorporating these standards into a new methodology that elevates the specs to a cloud-native application strategy.

Jakarta EE 8 Enterprise Java Bean	Java EE 8 Enterprise Java Bean
New methodologies adapted for cloud Native Applications	Don’t incorporate with Cloud Native Applications
Jakarta Enterprise Beans references are special entries in the application component’s naming environment. **Deployer** binds the Jakarta Enterprise Beans reference to the enterprise bean’s business interface, no-interface view, or home interface in the target operational environment.	Bean Validation 2.0 (JSR 380) provides an annotation-based model for validating JavaBeans.
The Application Assembler can use the ejb-link element in the deployment descriptor to link a Jakarta Enterprise Beans reference to a target enterprise bean.	Contexts and Dependency Injection (CDI) 2.0 (JSR 365) adds support for firing asynchronous events, ordering event observers, using configuration SPIs, using built in annotation literals, and applying interceptors on producers.
Deployment tools provided by the Jakarta EE Product Provider must be able to process the information supplied in class file annotations and in the **ejb-ref** and **ejb-local-ref** elements in the deployment descriptor.	**For each entity bean, complete work to handle persistence operations.** For EJB 3.x modules, consider using the Java Persistence API (JPA) specification to develop plain old Java Object (POJO) persistent entities
A **resource manager connection factory** is an object that is used to create connections to a resource manager. For example, an object that implements the “javax.sql.DataSource” interface is a resource manager connection factory for “java.sql.Connection” objects that implement connections to a database management system.	**In message driven bean** , basically consider a generic enterprise application that uses one message-driven bean to retrieve messages from a JMS queue destination, and passes the messages on to another enterprise bean that implements the business logic.

Microprofile.io


MicroProfile 4.1
The MicroProfile 4.1 version is now available! This release contains an update to Health & relaxes compatibility constraints.
MicroProfile 4.1 Presentation
Open Tracing 2.0
Open API 2.0
Rest Client 2.0
Config 2.0
Fault Tolerance 3.0
Metrics 3.0
JWT Propagation 1.2
Health 3.1
CDI 2.0
JSON-P 1.1
JAX-RS 2.1
JSON-B 1.0
Jakarta Annotation 1.3
Updated in MicroProfile 4.1
No change from MicroProfile 4.0 Release

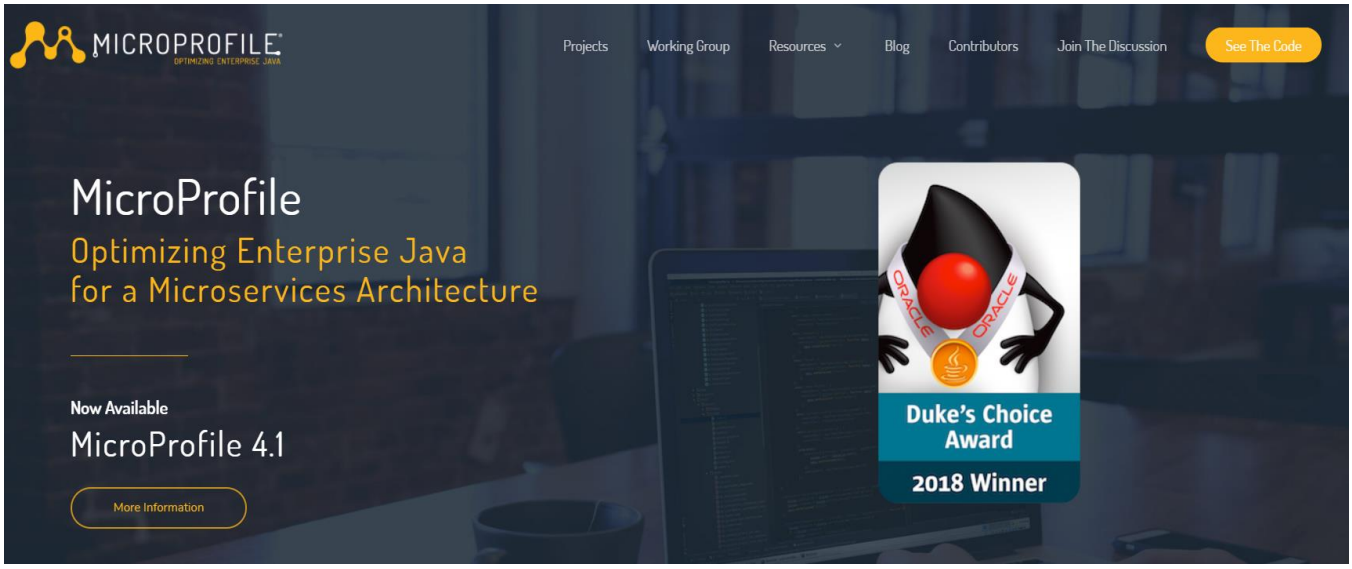

groupId *
com.example
MicroProfile Version
MP 3.3
Project Options
MicroProfile Runtime *
DOWNLOAD
MicroProfile Starter
MicroProfile Starter helps developers kickstart their microservices development journey, choosing the runtime they're most comfortable with from the list of available implementations for the MicroProfile version selected.
Try It Out!

Part 2

Microprofile.io is an initiative of making JEE features available for cloud-native and microservices architectures.

Question 1

Name at-least four **open-source Microprofile.io** runtimes currently available in the market.



Microprofile.io is an open forum for **optimizing Enterprise Java** for a microservices architecture through innovation across many implementations and collaboration on common areas of interest, with the objective of standardization.

➔ **Open-source Microprofile.io runtimes**

1) Payara Micro

is a lightweight middleware platform for containerized Jakarta EE application deployments that does not require installation, configuration, or code rewrites for fast deployments.

2) Quarkus

A Kubernetes Native Java stack crafted from best-of-breed Java libraries and standards for GraalVM and OpenJDK HotSpot.

3) Wildfly

WildFly, formerly known as JBoss AS or just JBoss, is a JBoss-authored application server that is now maintained by Red Hat. WildFly is developed in Java and adheres to the specifications of the Java Platform, Enterprise Edition. It is available on a variety of platforms.

4) Helidon

Oracle launched Helidon, a lightweight microservices framework, in September 2018. Helidon is a set of Java libraries for building microservices-based systems. Helidon was meant to be easy and quick, and it comes in two versions: Helidon SE and Helidon MP.

5) KumuluzEE

is a **lightweight framework** for developing microservices with standard Java/JavaEE/JakartaEE/EE4J technologies and APIs with optional extensions, such as Node.js, Go, and other languages, as well as migrating existing applications to cloud-native architecture and microservices for easier cloud-native microservices development.

6) Hammock

This is a bootstrapping CDI-based Java Enterprise **Microservices framework** used for building applications due to its flexibility and simplicity.

Microprofile.io implementations

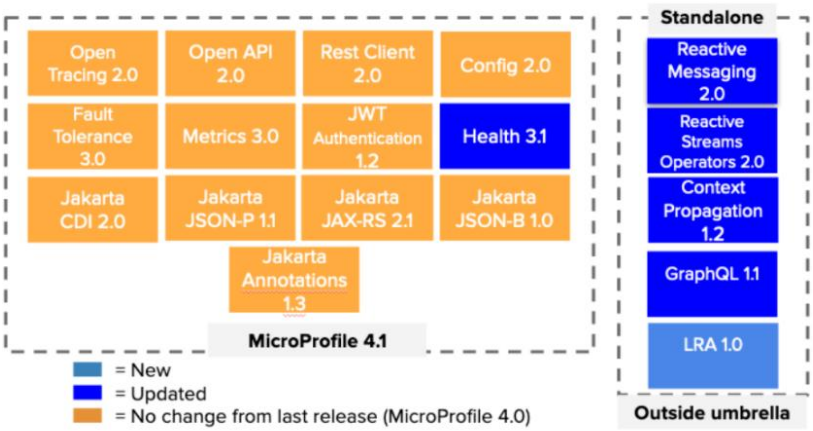
Question 2

Briefly describe the components of the latest **MicroProfile.io specification** (eg.opentraining, open-api) and how they are important for the microservices architecture.

What is Microprofile.io?

The **MicroProfile** is a foundational platform specification that optimizes Enterprise Java for a microservices architecture while also delivering application portability across various MicroProfile runtimes.

MicroProfile 4.1 component specification updates



Let's Analyze the Components in Latest Microprofile 4.1

1) MicroProfile Config 2.0

Importance of the Component

MicroProfile Config helps to achieve this objective by collecting configuration from many Config Sources and presenting the user with a single unified view. This enables the program to incorporate default settings into the application. It also allows you to change the defaults or delete the property by simply specifying the property name without a value or an empty string as the value, for example, via an environment variable, a Java system property, or a container like Docker.

A) Functional Changes

- ➔ **Support Configuration Profiles so that the corresponding properties associated with the active profile are used**
- ➔ **Provide built-in Converters: OptinalInt, OptionalLong and OptionalDouble**
- ➔ **Support Property Expressions: This provides a way to set and expand variables in property values**

B) API/SPI Changes

➔ **Enable bulk-extraction of config properties into a separate POJO by introducing** `@ConfigProperties`

➔ **Convenience methods have been added to Config allowing for the retrieval of multi-valued properties as lists instead of arrays**

2)MicroProfile Fault Tolerance 3.0

Introduction to Fault Tolerance 3.0

Building fault-tolerant microservices is becoming increasingly essential. Fault tolerance is the use of several techniques to influence the execution and outcome of some logic. Popular concepts in this field include retry policies, bulkheads, and circuit breakers. They determine whether and when executions should take place, and fallbacks provide an alternate outcome if an execution does not go as planned.

Importance of the Component to Microservices Architecture

A) Backward incompatible changes

➔ **The metrics added automatically by MicroProfile Fault Tolerance have been updated to take advantage of support for metric tags which was added to MicroProfile Metrics in version 2.0.**

- Eg:- Metrics are now exported under `/metrics` and `/metrics/base`, instead of `/metrics` and `/metrics/application` as in previous versions.

B) Functional changes

➔ **Updated metrics to use tags**

➔ **Specified lifecycle of circuit breakers and bulkheads**

The main design is to separate execution logic from execution. The execution can be configured with fault tolerance policies, such as **RetryPolicy, fallback, Bulkhead** & **CircuitBreaker**.

Hystrix and Failsafe are two popular libraries for handling failures. This specification is to define a standard API and approach for applications to follow in order to achieve the fault tolerance.

Circuit Breaker (One of the Fault tolerance policies)

A Circuit Breaker prevents repeated failures, so that dysfunctional services or APIs fail fast. If a service is failing frequently, the circuit breaker opens and no more calls to that service are attempted until a period of time has passed.

3) MicroProfile Health 3.1

Introduction to Health 3.1

The **Eclipse MicroProfile Health** standard specifies a single container runtime method for checking a MicroProfile implementation's availability and state. This is primarily designed to be used as a machine-to-machine (M2M) method in containerized settings such as cloud providers.

Example of existing specifications from those environments include **Cloud Foundry Health Checks** and **Kubernetes Liveness, Readiness and Startup Probes**.

Importance of the Component to Microservices Architecture

The MicroProfile Health architecture consists of three **/health/ready, /health/live and /health**/started endpoints in a MicroProfile runtime that respectively represent the readiness, the liveness and the startup health of the entire runtime. These endpoints are linked to health check procedures defined with specifications API and annotated respectively with `@Readiness`, `@Liveness` and `@Startup` annotations.

A) Functional changes

- ➔ **Specify the** `mp.health.default.startup.empty.response` **config property**
- ➔ **Integrated MicroProfile Parent POM**

REST interfaces specifications

Context	Verb	Status Code	Kind of procedure called
/health/live	GET	200, 500, 503	Liveness
/health/ready	GET	200, 500, 503	Readiness
/health/started	GET	200, 500, 503	Startup
/health	GET	200, 500, 503	Liveness + Readiness + Startup

B) API/SPI Changes

- ➔ **Added** `@Startup` **qualifier for the Kubernetes startup probes health check procedures.**
- ➔ **Introduction of** `/health/started` **endpoint that must call all the startup procedures**
- ➔ **Clarified** `mp.health.default.readiness.empty.response` **usage**

4) MicroProfile OpenTracing 2.0

Introduction to OpenTracing 2.0

Distributed tracing allows you to trace the flow of a request across service boundaries. This is particularly important in a microservices environment where a request typically flows through multiple services. To accomplish distributed tracing, each service must be instrumented to log messages with a correlation id that may have been propagated from an upstream service. A common companion to distributed trace logging is a service where the distributed trace records can be stored.

Importance of the Component to Microservices Architecture

OpenTracing is MicroProfile framework to interact effectively with a distributed trace system that is part of a broader microservices ecosystem. This standard specifies an API and MicroProfile characteristics that enable services to easily participate in a distributed tracing environment.

The currently used OpenTracing API version is {**opentracingversion**}.

- ➔ **There are two operation modes**
- ➔ **Without instrumentation of application code with explicit code instrumentation**

Enabling distributed tracing with no code instrumentation

A) Changes in OpenTracing 2.0

- ➔ **Exclude transitive dependency on** `javax.el-api`
- ➔ **Remove OpenTracing API from WAR in TCK**
- ➔ **Use Jakarta EE 8 APIs instead of Java EE 7 and remove dependency on Jackson**

API Changes in OpenTracing 2.0

- ➔ `Scope = ScopeManager.active()`**: no alternative, the reference Scope has to be kept explicitly since the scope was created.**
- ➔ `Scope = ScopeManager.activate(Span, boolean)`**: no alternative auto-finishing has been removed.**
- ➔ `Span = Scope.span()` : use `ScopeManager.activeSpan()` **or hold the reference to Span explicitly since the span was started.**

Enabling distributed tracing with no code instrumentation

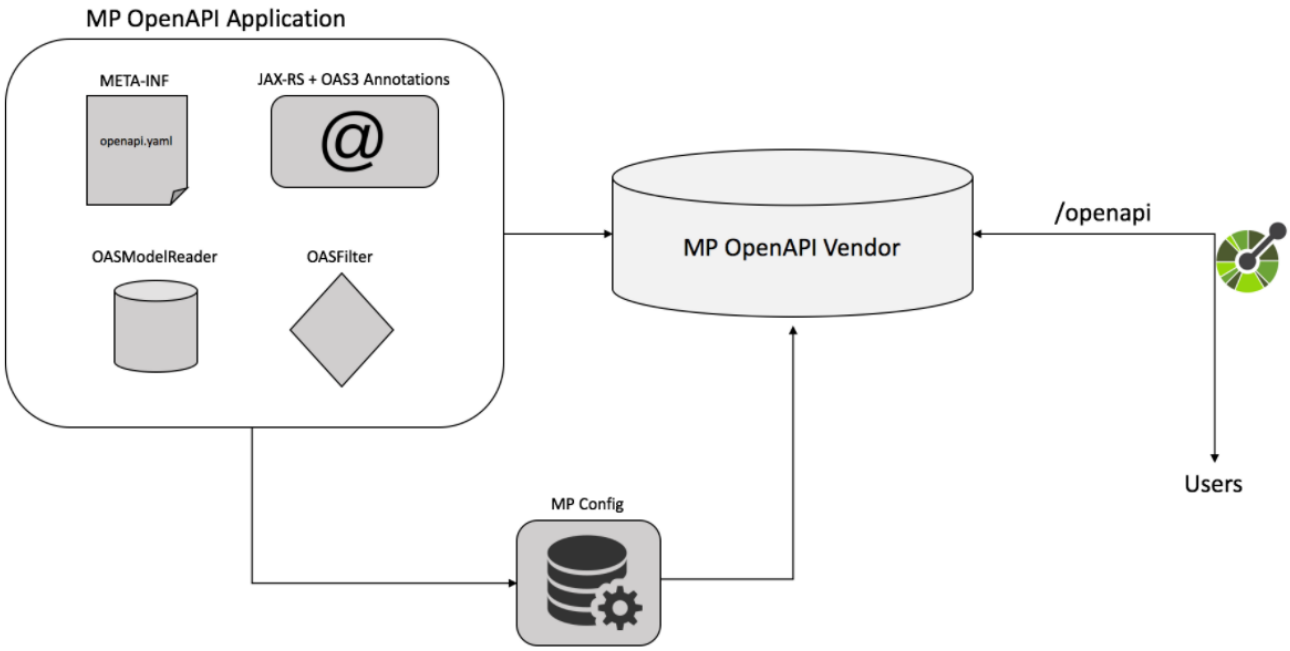
Thie MicroProfile component implementation will allow JAX-RS apps to participate in distributed tracing without developers having to contribute any distributed tracing code to their applications or knowing anything about the distributed tracing environment in which their JAX-RS application will be deployed.

5) MicroProfile OpenAPI 2.0

Introduction to OpenAPI 2.0

A clear and thorough contract is required for the clients and suppliers of these services to connect. The OpenAPI v3 specification is the contract for RESTful Services, similar to the WSDL contract for traditional Web Services.

There are different ways to augment a JAX-RS application in order to produce an OpenAPI document.This illustration provides you brief understanding about OpenAPI.



Hierarchy of OpenAPI Component

Importance of the Component to Microservices Architecture

Implementing the MP OpenAPI standard can potentially provide additional native means for these configuration variables to be injected into the framework (for example, via a server configuration file), if they also implement the MP Config specification.

For convenience of vendors (and application developers using custom ConfigSources), the full list of supported configuration keys is available as constants in the **OASConfig class**.

A) API/SPI Changes

➔ The `@SchemaProperty` annotation has been added to allow the properties for a schema to be defined inline.

B) Functional Changes

➔ Getter methods on model interfaces that return a list or map now return a copy of the list/map containing the same items. This list/map CAN be immutable.
➔ Setter methods on model interfaces that take a list or a map as a parameter MUST not use the list/map instance directly
➔ TCK updates updates to verify that getter methods on model interfaces return a list or map, return a copy of underlying collection

6) MicroProfile Rest Client 2.0

Introduction to Rest Client 2.0

MicroProfile TypeSafe Rest Clients are defined as Java interfaces.

```
public interface MyServiceClient {
    @GET
    @Path("/greet")
    Response greet();
}
```

Importance of the Component to Microservices Architecture

1) **MicroProfile Rest Client Programmatic Lookup**

For use, Type Safe Rest Clients offer both programmatic lookup and CDI injection methods. Both use cases are intended to be supported by a MicroProfile Rest Client implementation.

2) **MicroProfile Rest Client Provider Registration**

The RestClientBuilder interface enhances the JAX-RS Configurable interface by allowing users to register custom providers while the application is being constructed. The JAX-RS Client API standard defines the behavior of the available providers. The following provider types are required to be supported by an implementation:

- ClientResponseFilter
- ClientRequestFilter
- MessageBodyReader
- MessageBodyWriter

MicroProfile Config

MP Rest Client employs MP Config to declaratively configure client behavior. MP Config may be used to set the remote URI, client providers and priority, connect and read timeouts, and so forth**.**

MicroProfile Rest Client 2.0 Changes

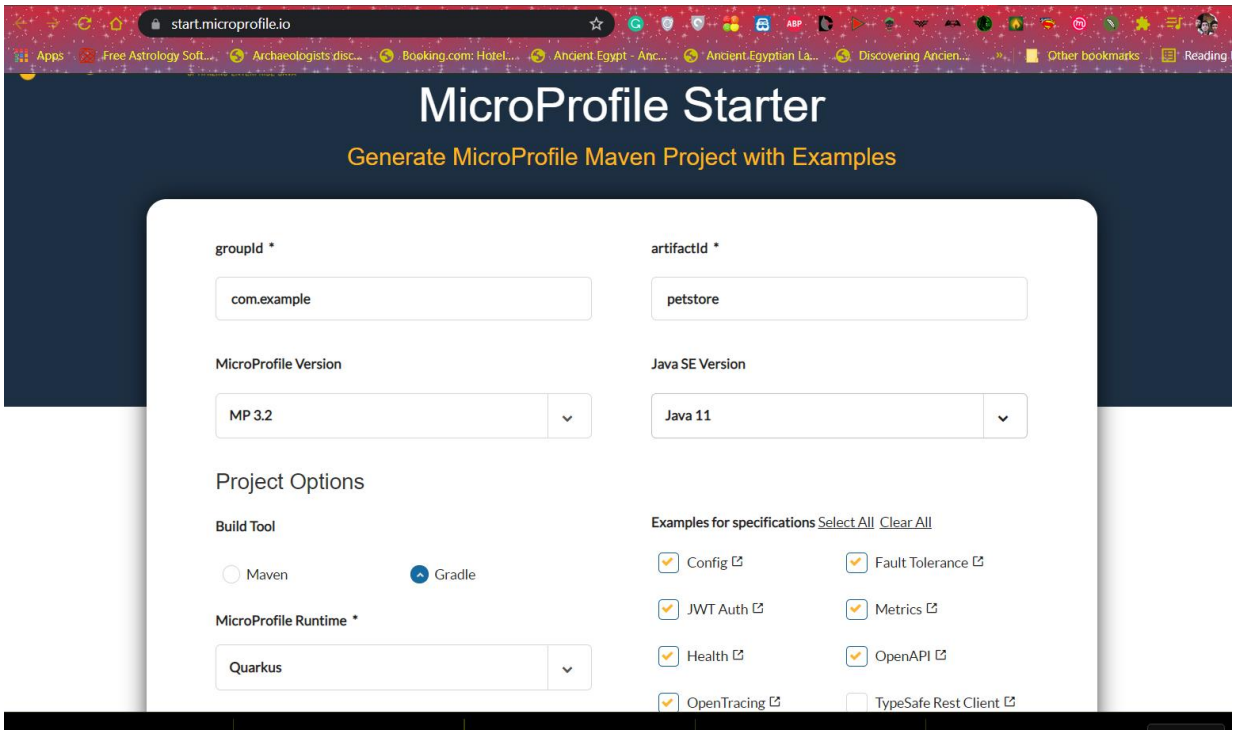
- **Defined that CDI-managed providers should be used instead of creating a new instance, if applicable.**
- **Support different configurations for collections used in query parameters.**
- **Configuration for automatically following redirect answers has been added.**
- **Added support for Server Sent Events.**

Part 3

Assume that you have been selected for implementing a **Pet-Store API** which will be consumed by the front-end developers for developing the Pet-Store web-application.

Question 1

1. Pick a **MicroProfile.io** runtime of your choice and justify the selection.



So When I working with Microprofile.io , to Implement **Pet-Store API ,** as the runtime environment I have used **"Quarkus" runtime** enviornment .

Let me justify my choice to select **Quarkus Environment** as the Runtime environment,

Quarkus is a **full-stack, Kubernetes-native Java** framework optimized for Java virtual machines (JVMs) and native compilation, allowing Java to become a viable platform for serverless, cloud, and Kubernetes settings.

Quarkus is built to operate with well-known Java standards, frameworks, and libraries such as Eclipse MicroProfile and Spring (demonstrated together as part of a session in this Red Hat Summit 2020 track).

QUARKUS

The Reason for selecting quarkus ?

1) Designed for developers

Quarkus was built from the ground up to be simple to use, with features that perform effectively with little to no configuration.

So implementing **Pet-Store API ,** this feature pursuit me to select quarkus and as a novice java developer, this runtime environment aided me to learn important fetaures in API handling.

Quarkus was designed with developers in mind, and it has the following features:

1) Live coding allows me to immediately see the impact of code changes and debug them.

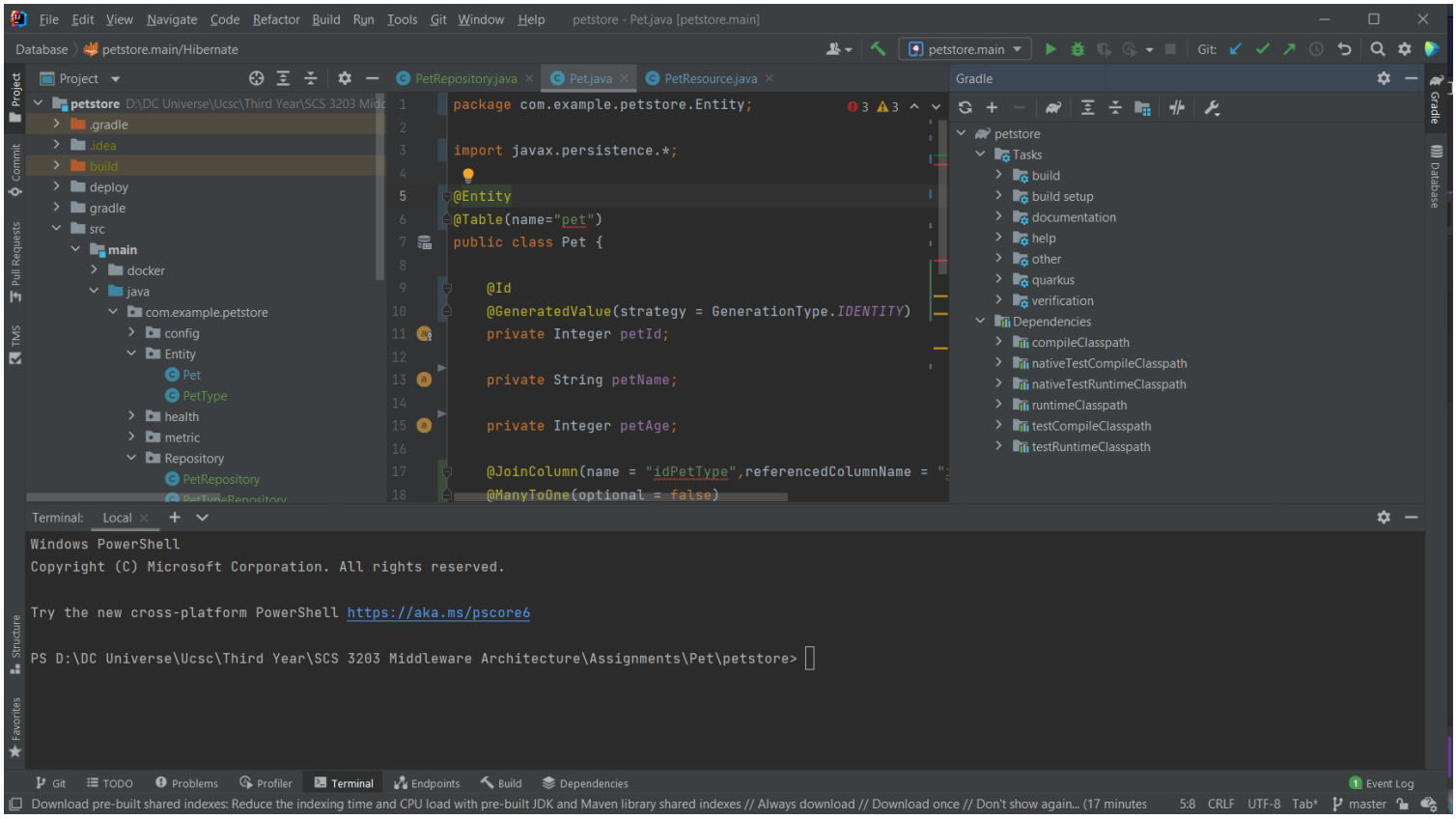


Figure 1:- Pet-Store in **Quarkus** Environment with **Gradle** Build Tool

2) Easy native executable generation

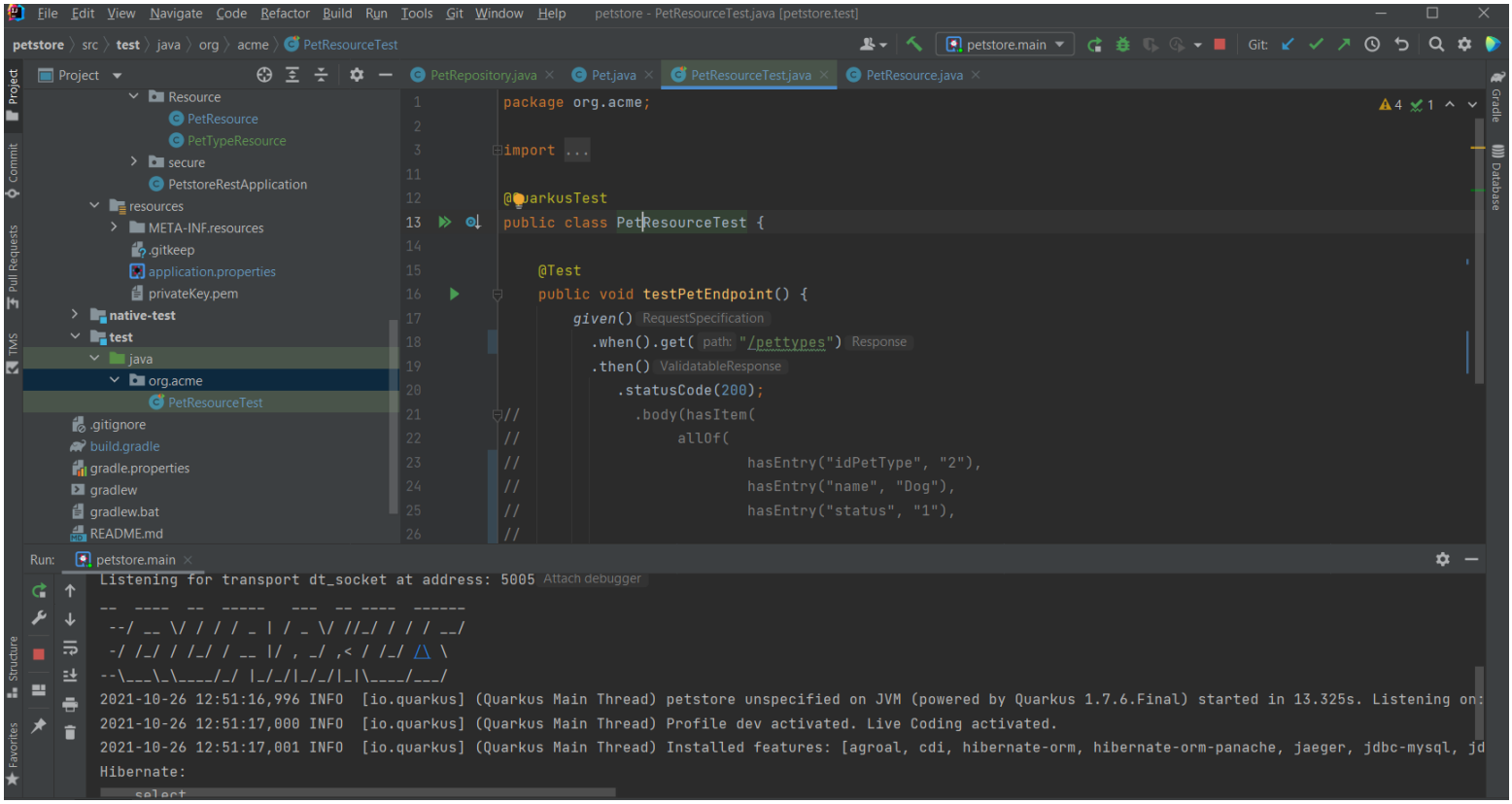


Figure 2: - Pet-Store Executed in **Quarkus** Environment

3) With an embedded managed event bus, you can combine imperative and reactive

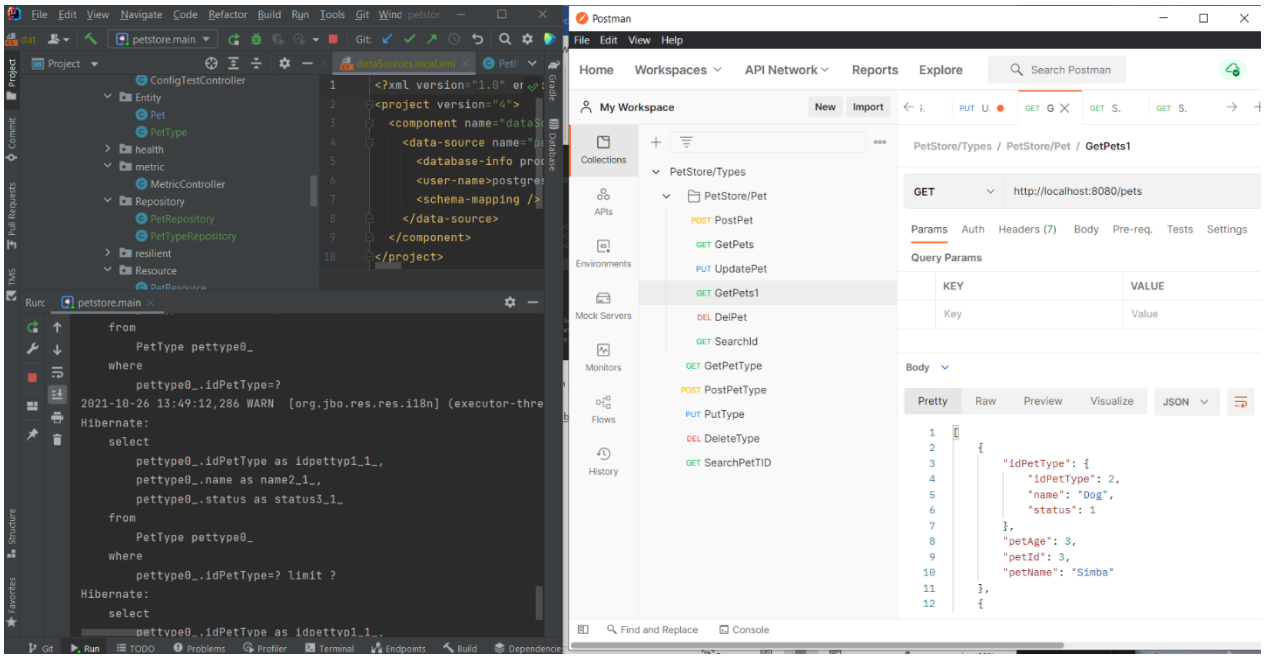


Figure 3: - Executing Query with **Quarkus** Environment via Postman

2) Container-first

Quarkus was designed with a container-first mindset in mind, which means it's optimized for lower memory use and faster startup times in the following ways:

- First-class support for Graal/SubstrateVM
- Build-time metadata processing

As a result, Quarkus produces apps that utilize 1/10th the memory of typical Java and have a shorter startup time (up to 300 times faster), both of which significantly lower the cost of cloud resources.

3) Imperative and reactive code

When designing apps, Quarkus is meant to effortlessly integrate the conventional imperative style code and the non-blocking, reactive form.

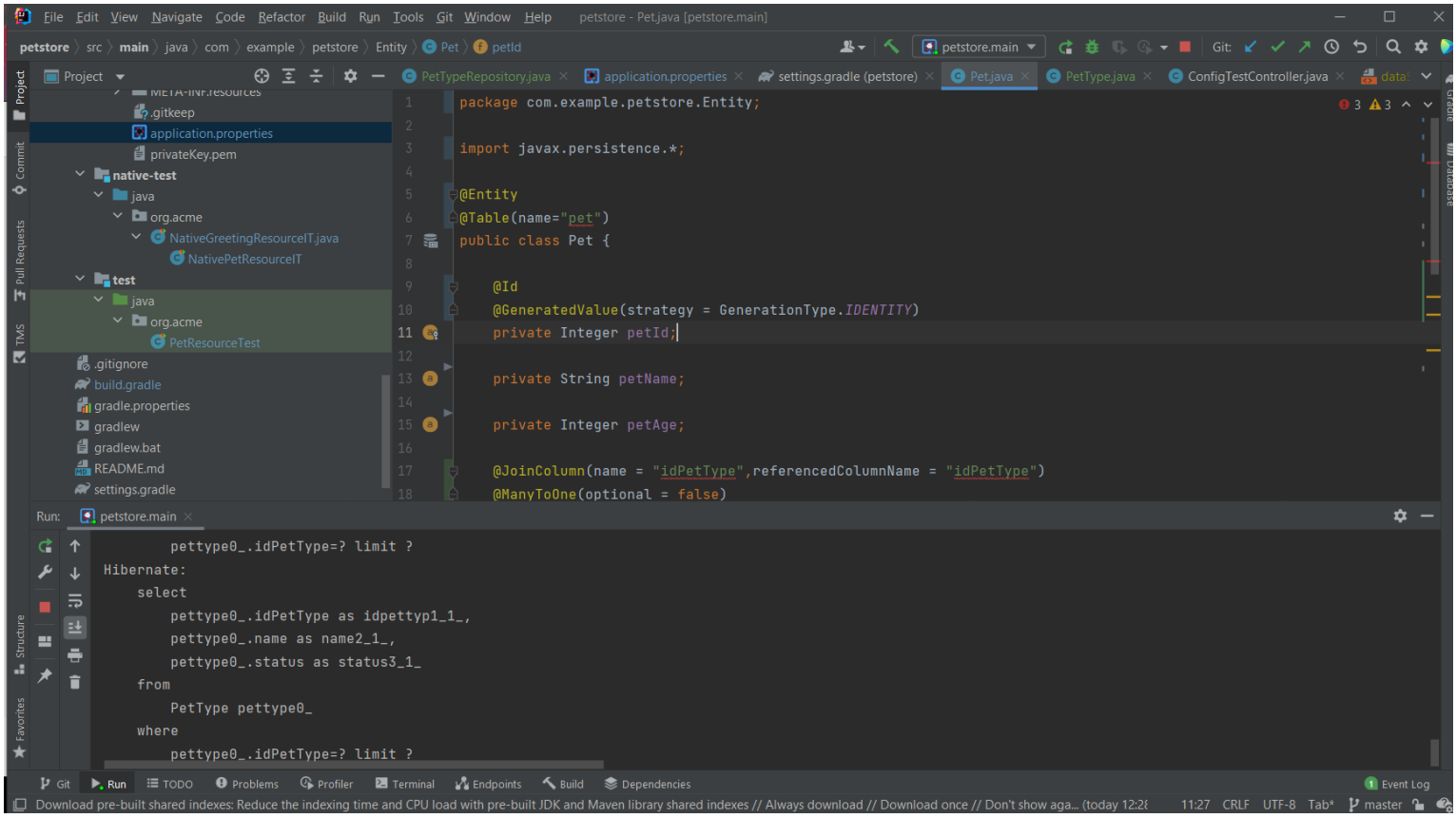


Figure 4: - Pet-Store API Building Via **Quarkus** is convenient because of ease of resources accessibility and Reactive Coding

So these Circumstances reasons for Selecting Runtime Environment as **Quarkes** Build Pet-Store API Application.

Question 2

2. Explain and justify "Content type" you suggest and "Webservice type" you will be using to implement the API.

1) Content Type

In my API implementation I have selected Content Type as **JSON** to pass the data into database **"PetStore".**

JSON structure

JSON is a string whose format very much resembles JavaScript object literal format. You can include the same basic data types inside JSON as you can in a standard JavaScript object — strings, numbers, arrays, booleans, and other object literals.
JSON notation is a built-in format of javascript which also used in front-end integration because of ease of adaptabuility.

Advantages of JSON

- Can use pure text received from a server as a JavaScript object. (Essential for data interpretation to the forntend)
- Can work with data as JavaScript objects, with no complicated parsing and translations
- It is more vital for websites to load data **rapidly and asynchronously**, or in the background, without slowing page rendering. Changing the contents of a specific element within one of our layouts without requiring a page refresh

These Circumstances Conclude me to Select Content Type as **JSON** to pass the data elements via API.

2) Webservice Type

In the content type selection, I have chosen **REST** web-service for implementing **Pet-Store** API.

REST, or Representational State Transfer, may be utilized over practically any protocol; however, when used for online APIs, it commonly uses HTTP. This implies that while developing a REST API, Specially do not need to install any additional software or libraries when I am Developing **Pet-Store API.**

Also These charateristics in **REST** invloved for Selecting webservice down below,

- **Client-Server:** This restriction is based on the idea that the client and server should be kept separate and allowed to grow independently.
- **REST** is a lightweight protocol
- Because there is no contract between the server and the client, the implementation is loosely Coupled.
- REST methods are simple to test in a browser.
- Supports multiple technologies for data transfer such as text, xml, json, image etc.

These Cicumstances pursuit me to Select **REST API** as the Web Serivce for **Pet-Store.**

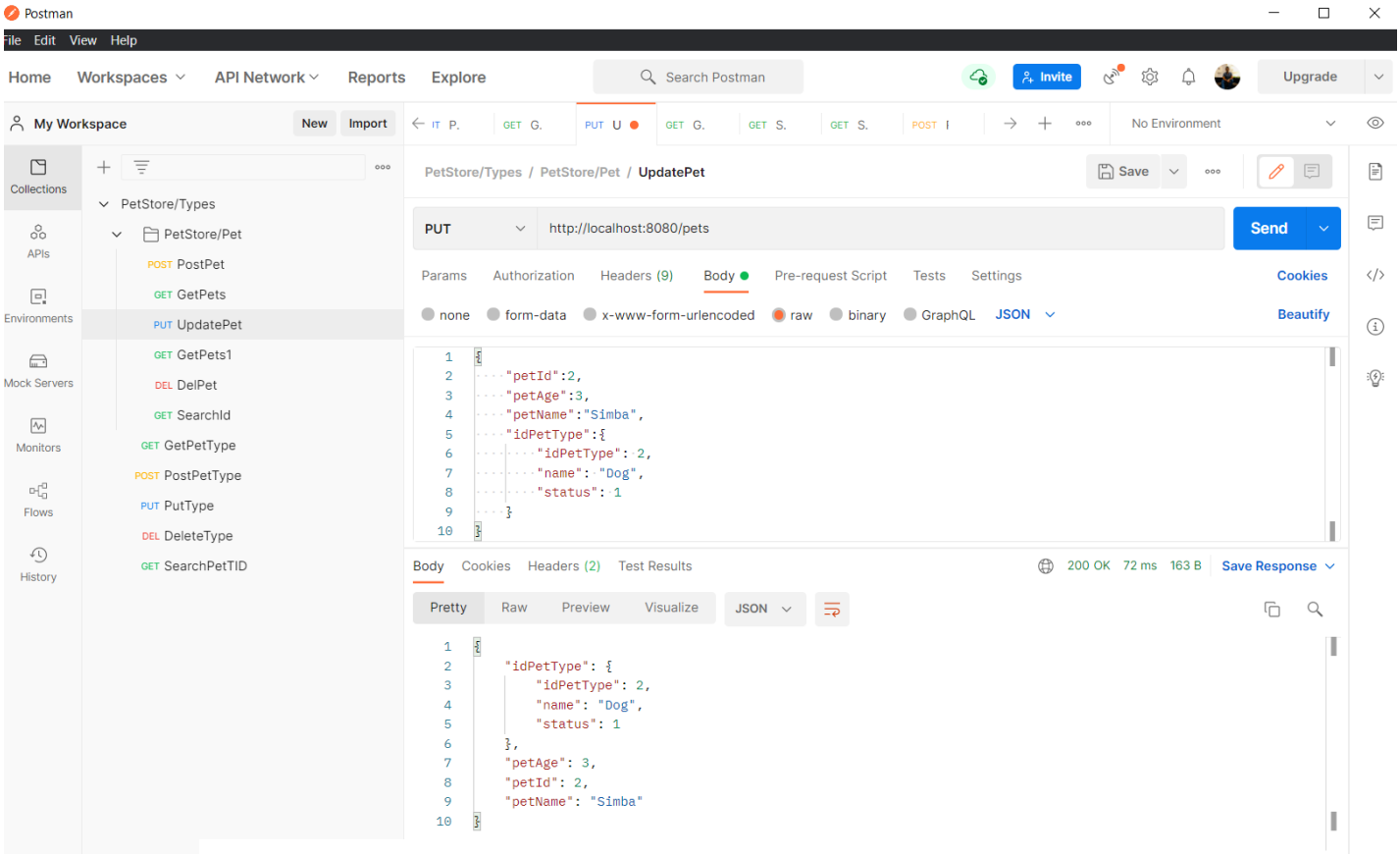


Figure 5: - Testing **Pet-Store API** by Passing **JSON** data format via **REST API**

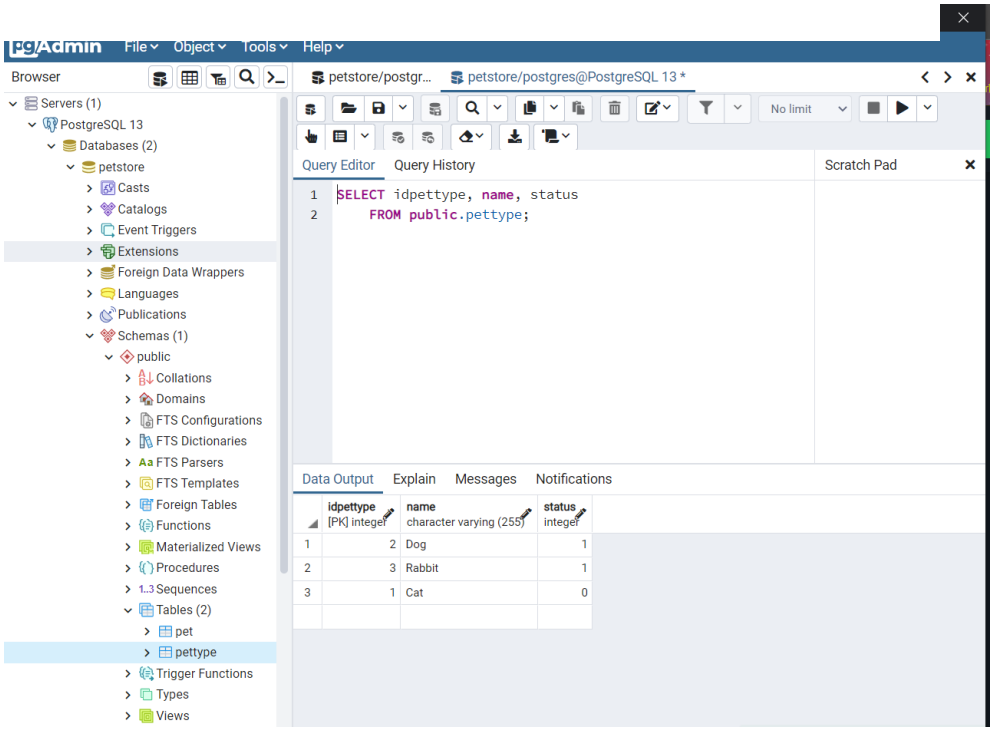


Figure 6: - **Pet & PetType** databases Created on Postgres from **REST** using **Quarkus** Environment

Question 3

Both **back-end** and **front-end** teams expected to start development in **parallel**. Any solution for front-end developers to continue a non-existent implementation?

- Inorder to continuation of **Frontend Developers in** non-existent implementation, both front & backend developers can be agreed on Common Specifications before start developing a project.

  Here are Some Solutions for Non-Existent Implementation

  ➔ **Fake API's**
  Websites such as **JSON Placehlder, DummyAPI, and FakeAPI** offer this service as a 'Data sandbox.' Data Sandbox is a collection of dummy data that responds to API requests.

  ➔ **Mircoprofile REST Client**
  It is based on the JAX-RS 2.0 client APIs and provides a type-safe approach to **calling RESTful services.** This means creating client applications with a model-centric foundation and less 'plumbing.' Client APIs, on the other hand, must be installed on a server.

  ➔ **API Documentation**
  System design & API documentation can be prepared before development containing the **Input , Output format** of the **API Calls.**

Testing (Using Postman) :- Question 4

4) Testing in the API

Pettype Table Testing (CRUD OPERATIONS)

Test Case No 1:- Inserting PetType From **PetStore API**

Test Case:- Insert	**Description: -** New Pet Type inserting via PetStore API		
Input Steps	Expected Output	Actual Output	Result (Pass/Fail)
1)Inserting Pet Type with Respective Credentials **Name =** Dog **Status =** 1	**Inserting Successfully from Quarkus into PostgreSQL Database "pettype" table as a New Entry.**	**Inserted Successfully from Quarkus into "pettype" table as a New Entry**	**Pass**
Remark of Errors: No Errors, Successfully Insert (Results Below)			

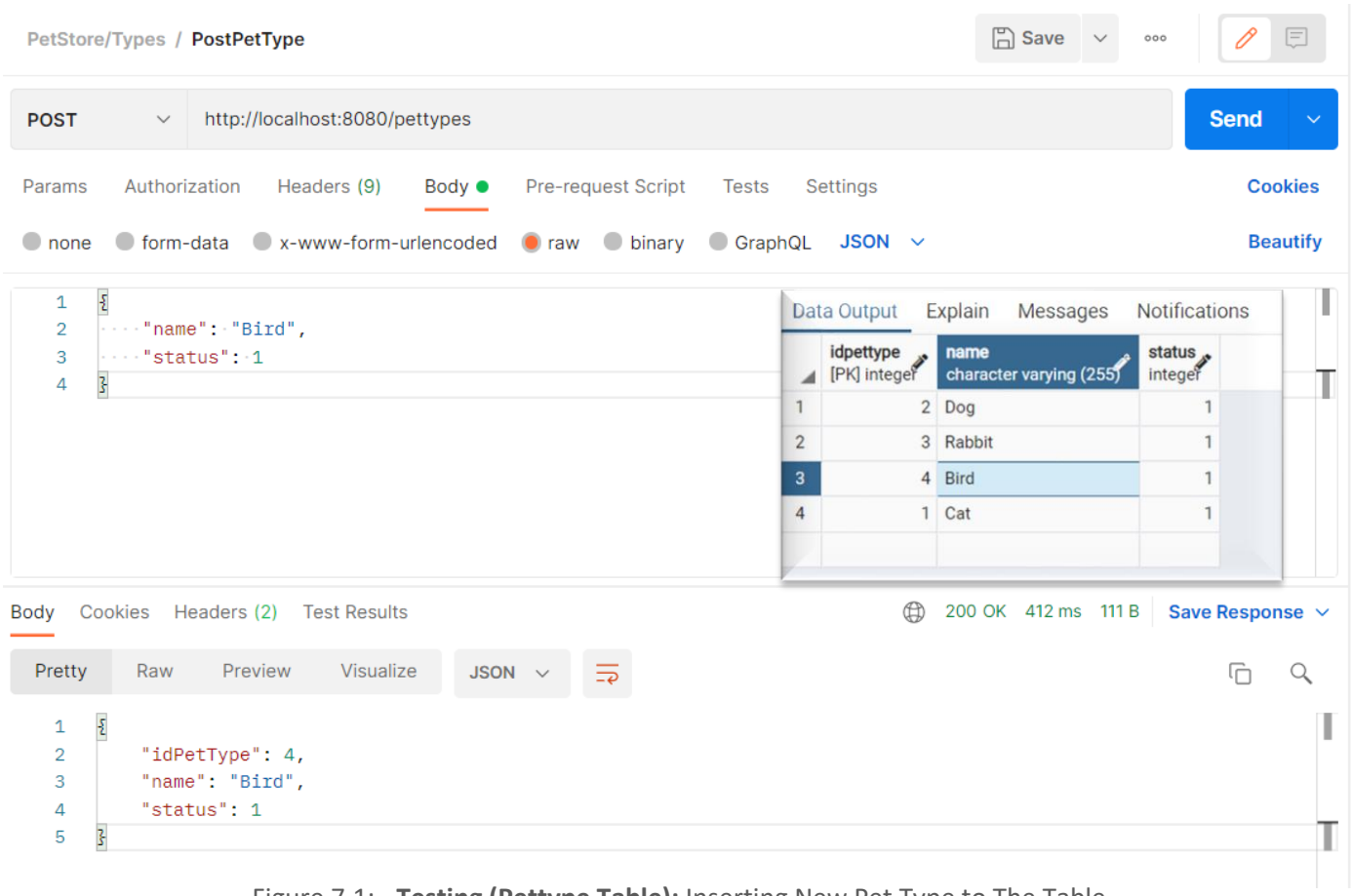


Figure 7.1: - **Testing (Pettype Table):** Inserting New Pet Type to The Table.

Test Case No 2:- Updating PetType From **PetStore API**

Test Case: - Update	**Description: -** Updating PetType Status via PetStore API		
Input Steps	Expected Output	Actual Output	Result (Pass/Fail)
1)Updating Pet Type with Respective Credentials OldName = Cat **NewName =** GoldFish	**Setting** 'Cat' **pettype name into** 'GoldFish' **and update the Entry.**	**Set the** 'Cat' **to Pettype** 'Goldfish' **from Quarkus and Updated Entry.**	**Pass**
Remark of Errors: No Errors, Successfully Updated (Results Below)			

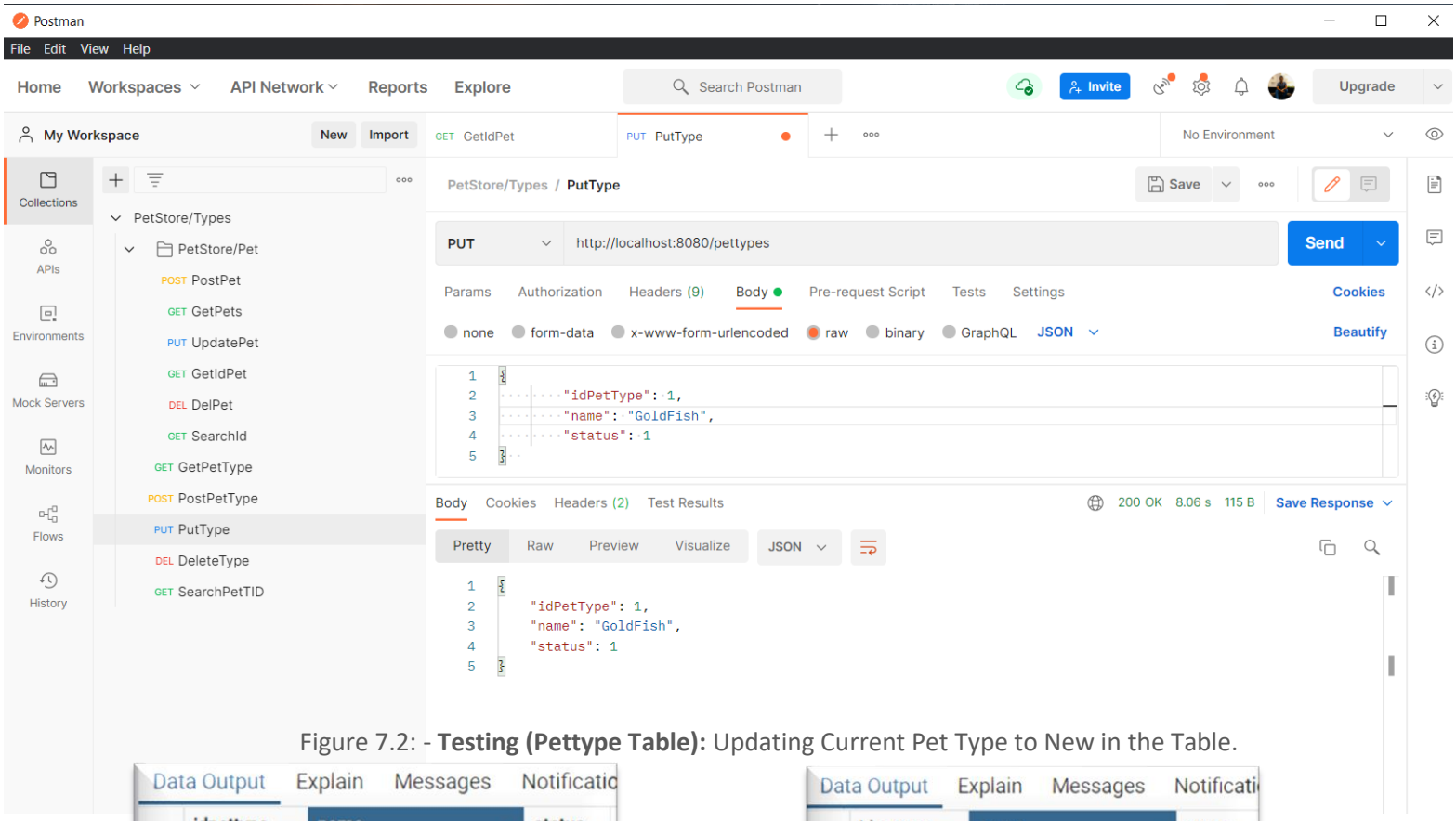


Figure 7.2: - **Testing (Pettype Table):** Updating Current Pet Type to New in the Table.

Data Output | Explain | Messages | Notificatic

	idpettype [PK] integer	name character varying (255)	status integer
1	2	Dog	1
2	3	Rabbit	1
3	4	Bird	1
4	1	Cat	1

Before Update Pettype

Data Output | Explain | Messages | Notificati

	idpettype [PK] integer	name character varying (255)	status integer
1	2	Dog	1
2	3	Rabbit	1
3	4	Bird	1
4	1	GoldFish	1

After Update Pettype

Test Case No 3:- Searching PetType From **PetStore API**

Test Case: - Search	**Description: -** Searching PetType by Id via PetStore API		
Input Steps	Expected Output	Actual Output	Result (Pass/Fail)
1)Provide Pettypeid in the Url. Pettypeld = 2	**Search Pettype id =2 and displaying the results**	**Search Pettype id =2 and displayed all the results respective to id.**	**Pass**
Remark of Errors: No Errors, Successfully Searching (Results Below)			

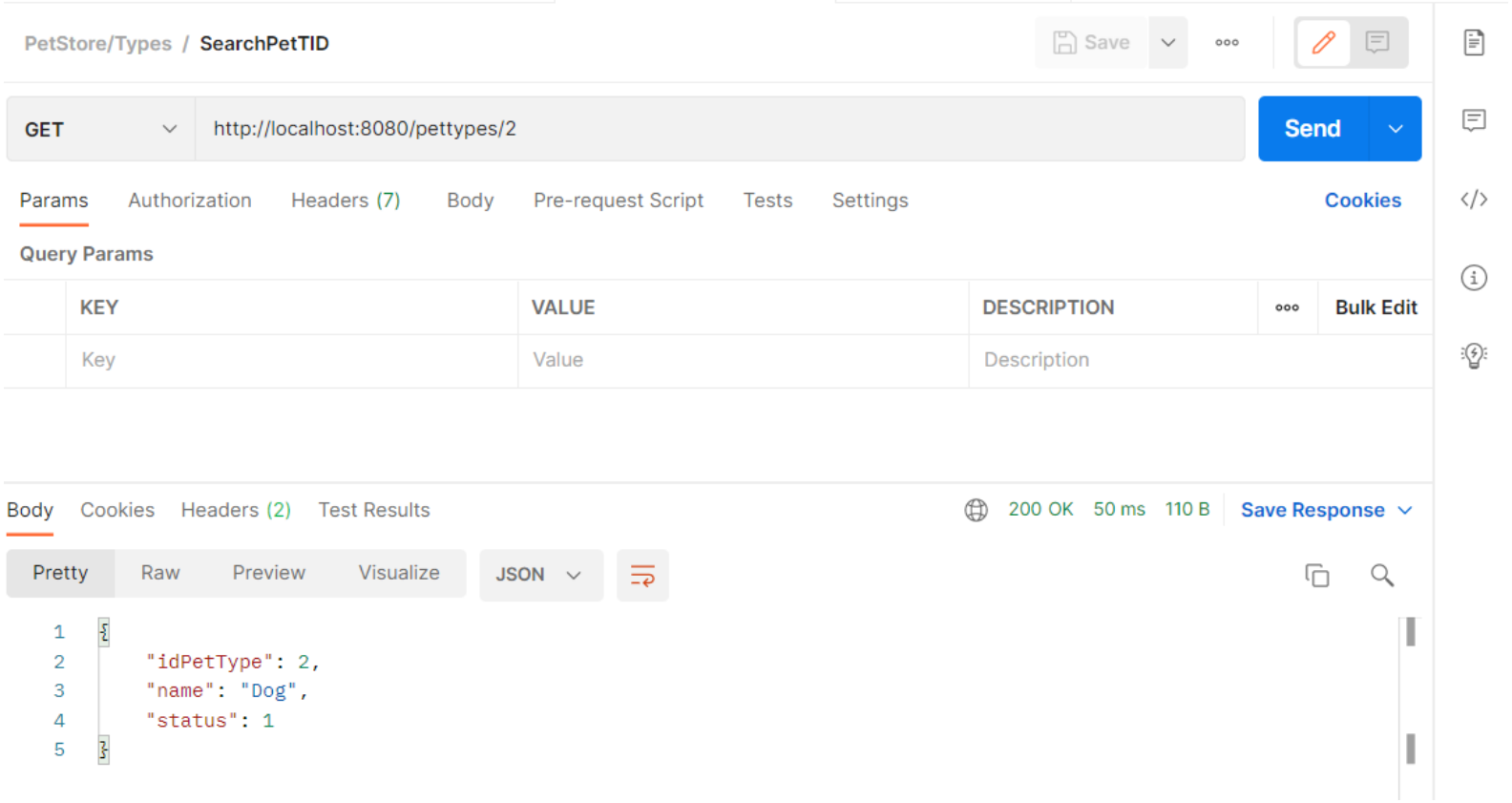


Figure 7.3: - **Testing (Pettype Table):** Searching New Pet Type by **Id** to The Table.

Test Case No 4:- Deleting PetType Status format From **PetStore API**

Test Case: - Deletion	**Description: -** Deletion PetType by Id via PetStore API		
Input Steps	Expected Output	Actual Output	Result (Pass/Fail)
1)Deleting Pettypeid in the Status	**Deleting Status to 0, in Petname=**Cat	**Deletes Status of Petname=** Cat **into 0.**	**Pass**
Remark of Errors: No Errors, Successfully Deleting (Results Below)			

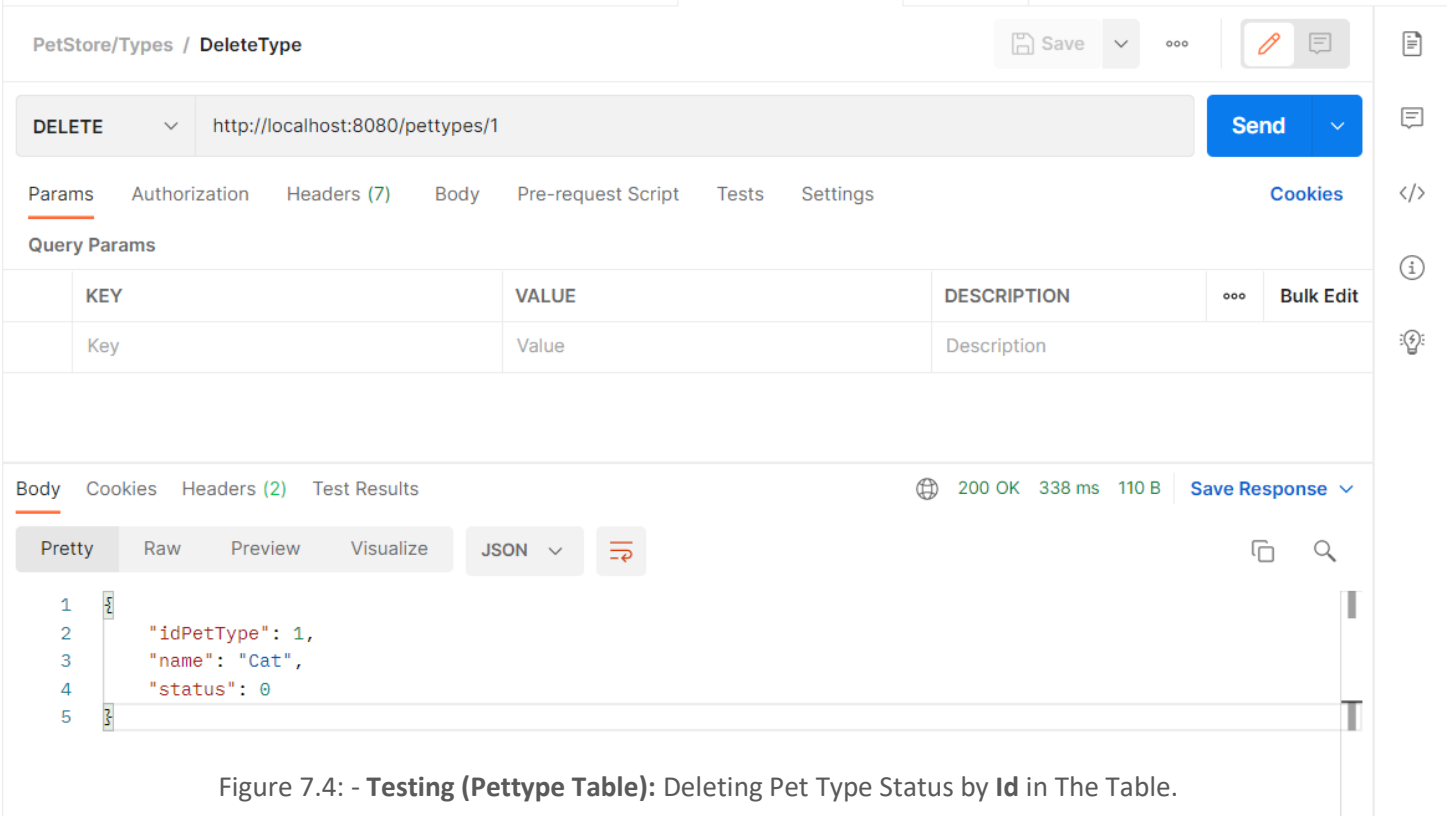
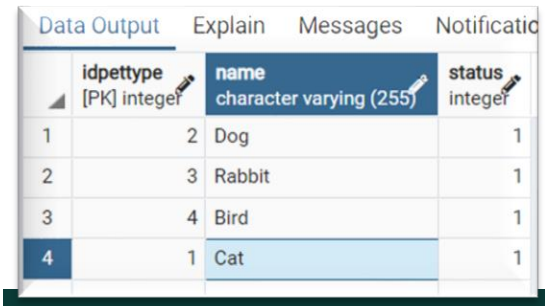
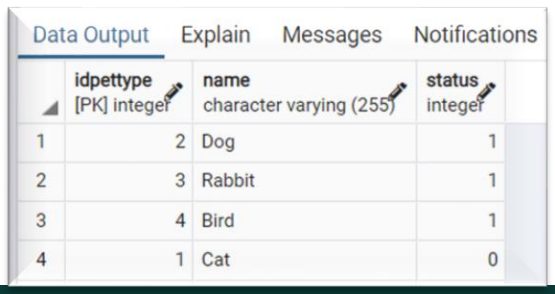


Figure 7.4: - **Testing (Pettype Table):** Deleting Pet Type Status by **Id** in The Table.





Before Deletion Pettype **After Deletion Pettype**

Test Case No 5:- Searching All PetType Credentials From **PetStore API**

Test Case: - Select	**Description: -** Displaying PetTypes via PetStore API		
Input Steps	Expected Output	Actual Output	Result (Pass/Fail)
1)Selecting Pet Types from 'pettype table	**Selecting all the Pet Types from Post request & Display available Pet Types**	**Selecting all the Pet Types from Existing pettype table and Displayed**	**Pass**
Remark of Errors: No Errors, Successfully Selection Query Executed (Results Below)			

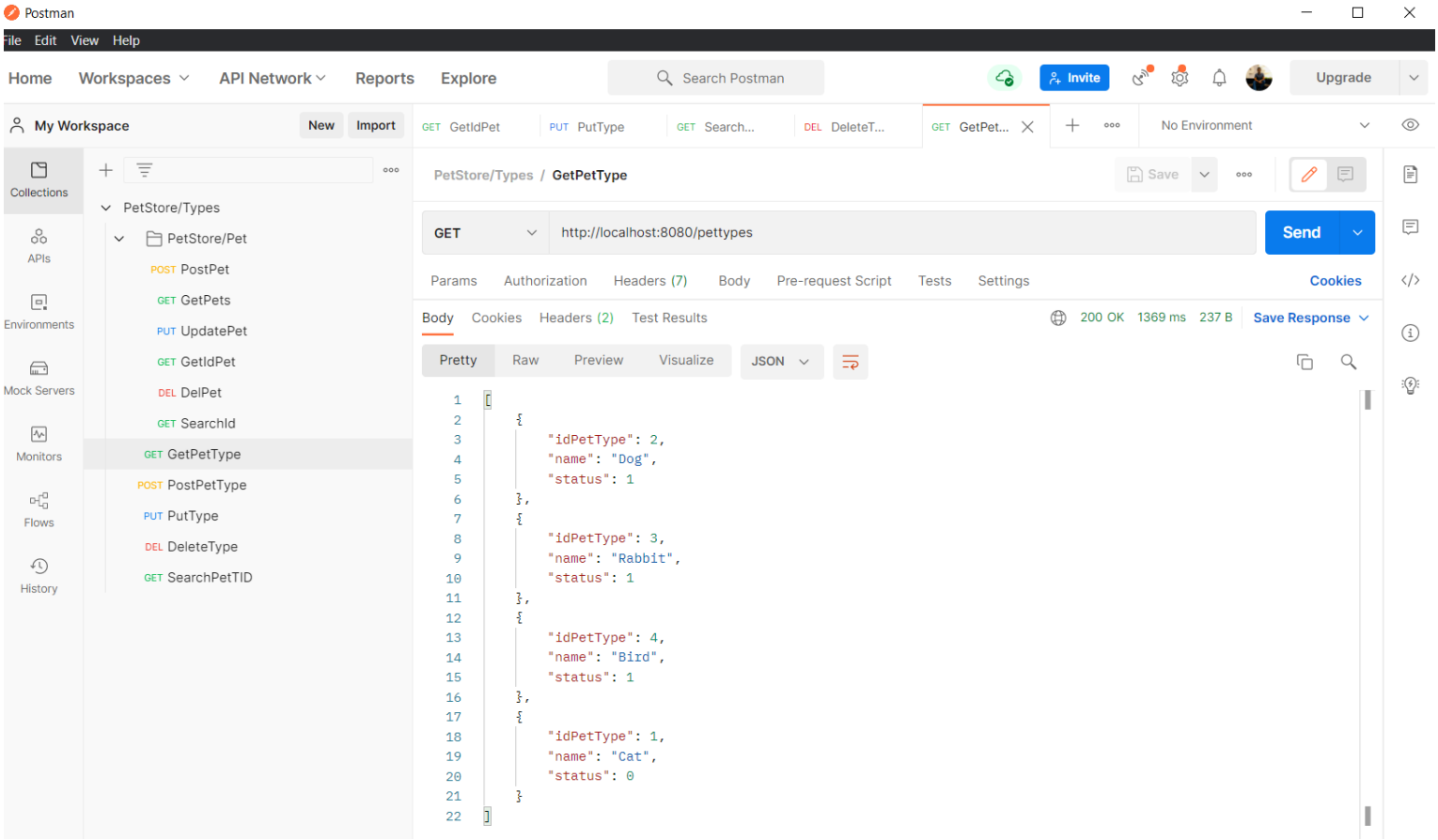


Figure 7.5: - **Testing (Pettype Table):** Retrieve All Pet Types in The Table.

Test Case No 6:- Inserting New Pet to Table From **PetStore API**

Test Case: - Insert	**Description: -** Inserting New Pet in Pet Table via PetStore API		
Input Steps	Expected Output	Actual Output	Result (Pass/Fail)
1)Inserting Pets Credentials. **Name: -** Maxsteel **Age: -** 2 **Pet Type: -** Dog	**Inserting Pets Credentials according to given columns and from Quarkus, it will be executed, and New Pet's information entered to table.**	**Inserting Pets Credentials according to given columns and from Quarkus , it will executed and new record added to table**	**Pass**
Remark of Errors: No Errors, Successfully Insertion Query Executed (Results Below)			

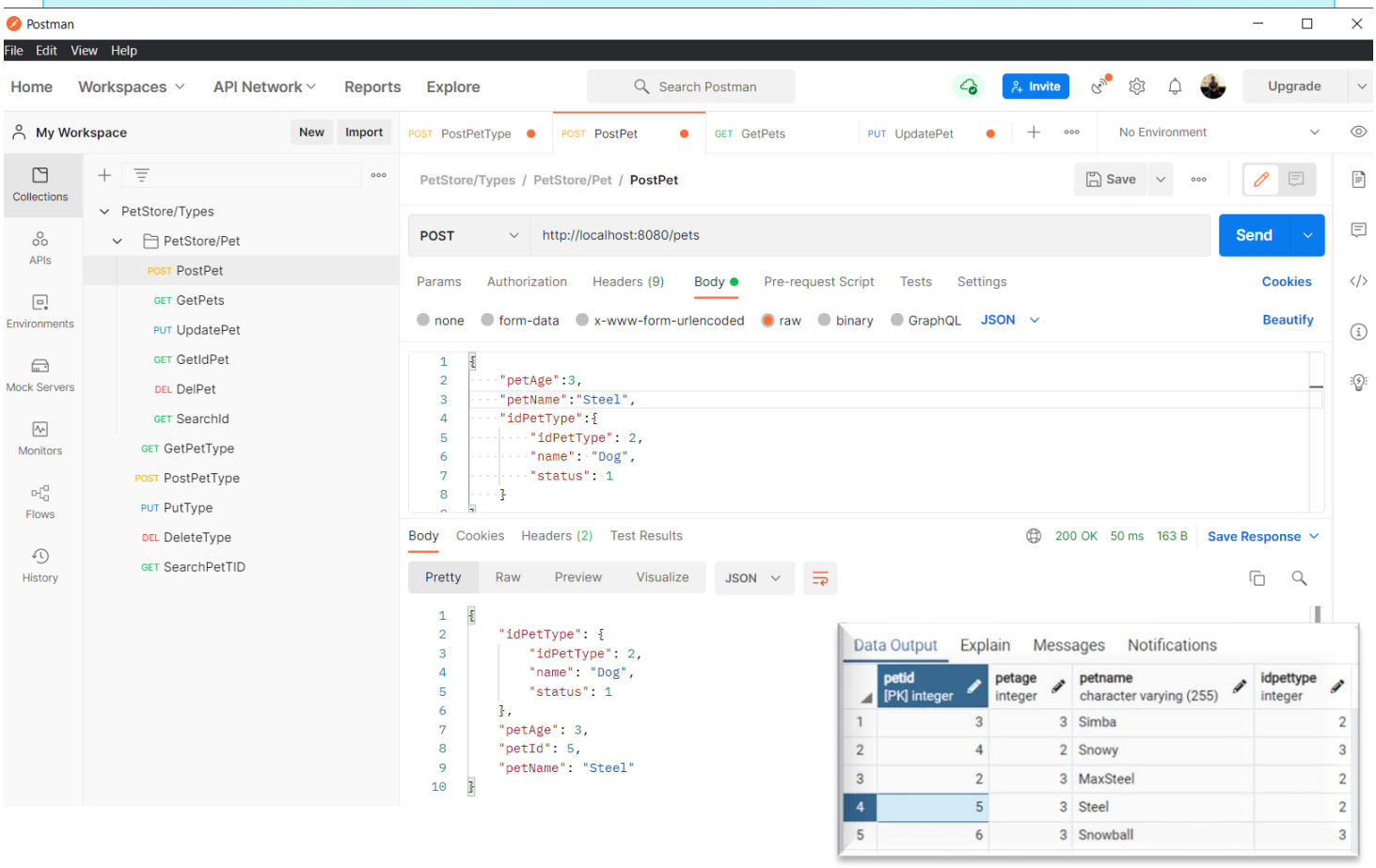


Figure 7.6: - **Testing (Pet Table):** Inserting New Pet to the Table.

Test Case No 7:- Searching a Pet using **PetId** via **PetStore API**

Test Case: - Update	**Description: -** Update Pet's Credentials using PetId via PetStore API		
Input Steps	Expected Output	Actual Output	Result (Pass/Fail)
1)Selecting Pet from Table and Update Details. **NewName =** Dusty **NewAge =** 1	**Selecting Id of the Pet to update Pet =** Snowy **to Dusty & Age into** 1. **After execution cells updated to new values.**	**Selects the id of the Pet and put new credentials. Execution with Quarkus , cells are updated with new values**	**Pass**
Remark of Errors: No Errors, Successfully Update Query Executed (Results Below)			

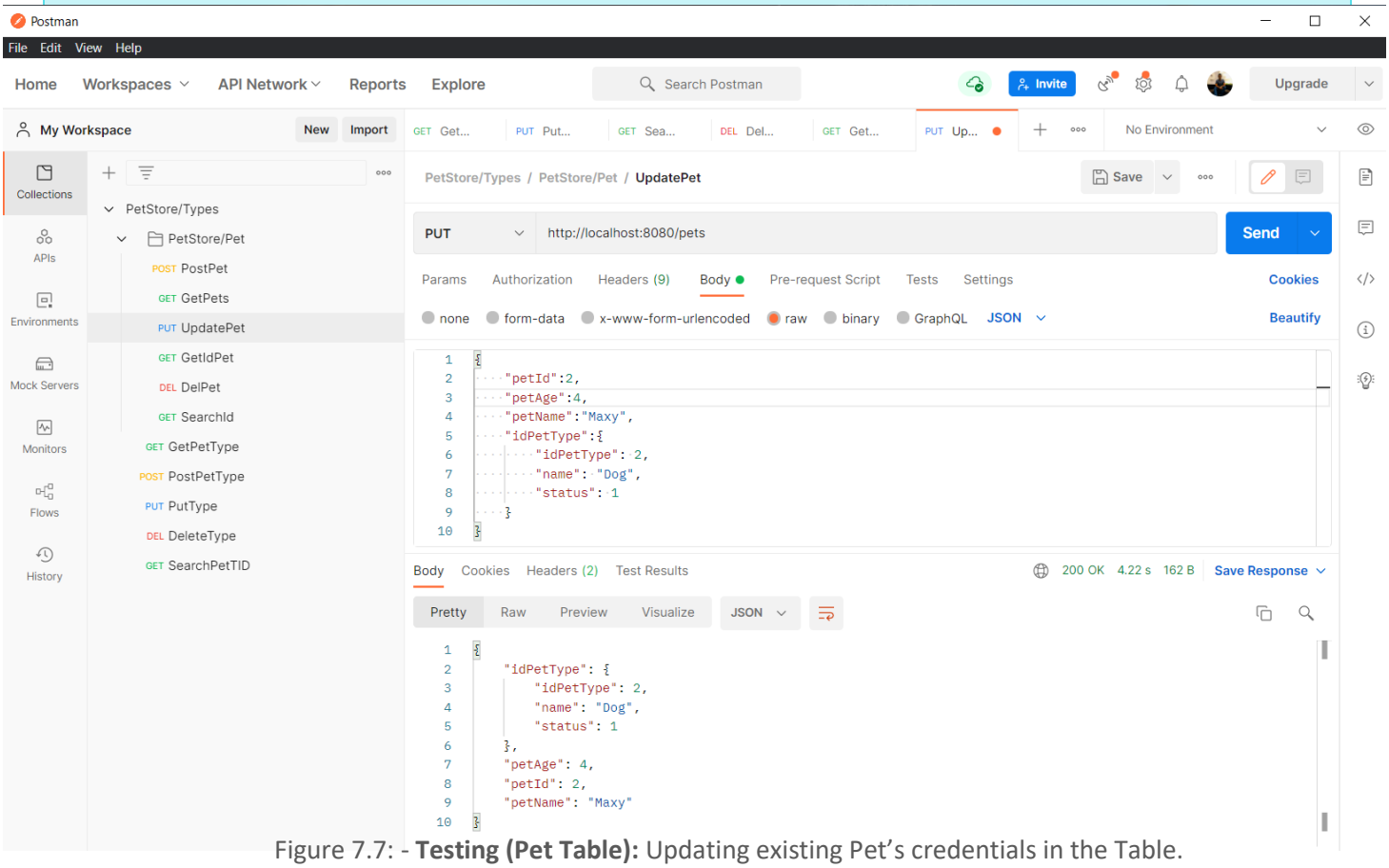


Figure 7.7: - **Testing (Pet Table):** Updating existing Pet's credentials in the Table.

Data Output | Explain | Messages | Notifications

	petid [PK] integer	petage integer	petname character varying (255)	idpettype integer
1	3	3	Simba	2
2	4	2	Snowy	3
3	2	3	MaxSteel	2
4	5	3	Steel	2
5	6	3	Snowball	3

Before Update Pet Table

Data Output | Explain | Messages | Notifications

	petid [PK] integer	petage integer	petname character varying (255)	idpettype integer
1	3	3	Simba	2
2	4	2	Snowy	3
3	5	3	Steel	2
4	6	3	Snowball	3
5	2	4	Maxy	2

After Update Pet Table

R NAME]

Test Case No 8:- Retrieve Particular Pet Credentials via **PetId** From **PetStore API**

Test Case: - Search	**Description: -** Retrieve data Using PetId via PetStore API		
Input Steps	Expected Output	Actual Output	Result (Pass/Fail)
1)Get the PetId wanted to Retrieve Data. **PetId =** 3	**Petid = 3 posted from the API & retrieve credentials relevant to Petid.**	**PetId =** 3, Posted **and running the Quarkus , retrieved the Credentials of PetId =3.**	**Pass**
Remark of Errors: No Errors, Successfully Searching Query Executed (Results Below)			

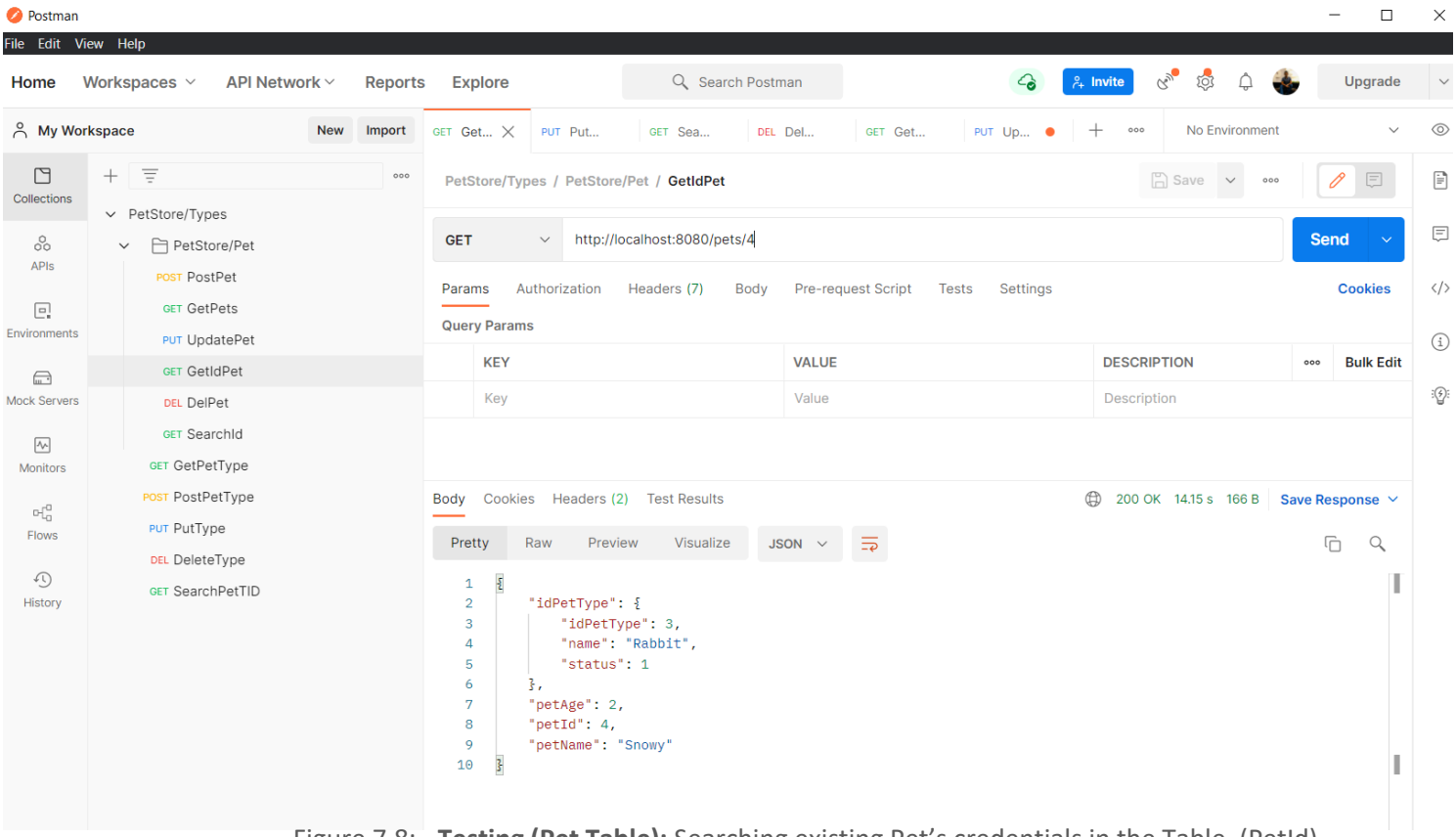


Figure 7.8: - **Testing (Pet Table):** Searching existing Pet's credentials in the Table. (PetId)

	petid [PK] integer	petage integer	petname character varying (255)	idpettype integer
1	3	3	Simba	2
2	4	2	Snowy	3
3	5	3	Steel	2
4	6	3	Snowball	3
5	2	4	Maxy	2

Test Case No 9:- Searching All Pet's Credentials From **PetStore API**

Test Case: - Select All	**Description: -** Displaying All Pets via PetStore API		
Input Steps	Expected Output	Actual Output	Result (Pass/Fail)
1)Searching All Pets from the Database.	**Selecting all the Pets from the Database and Display all the credentials.**	**Selecting all the Pet Types from Existing pettype table and Display.**	**Pass**
Remark of Errors: No Errors, Successfully Searching Query Executed (Results Below)			

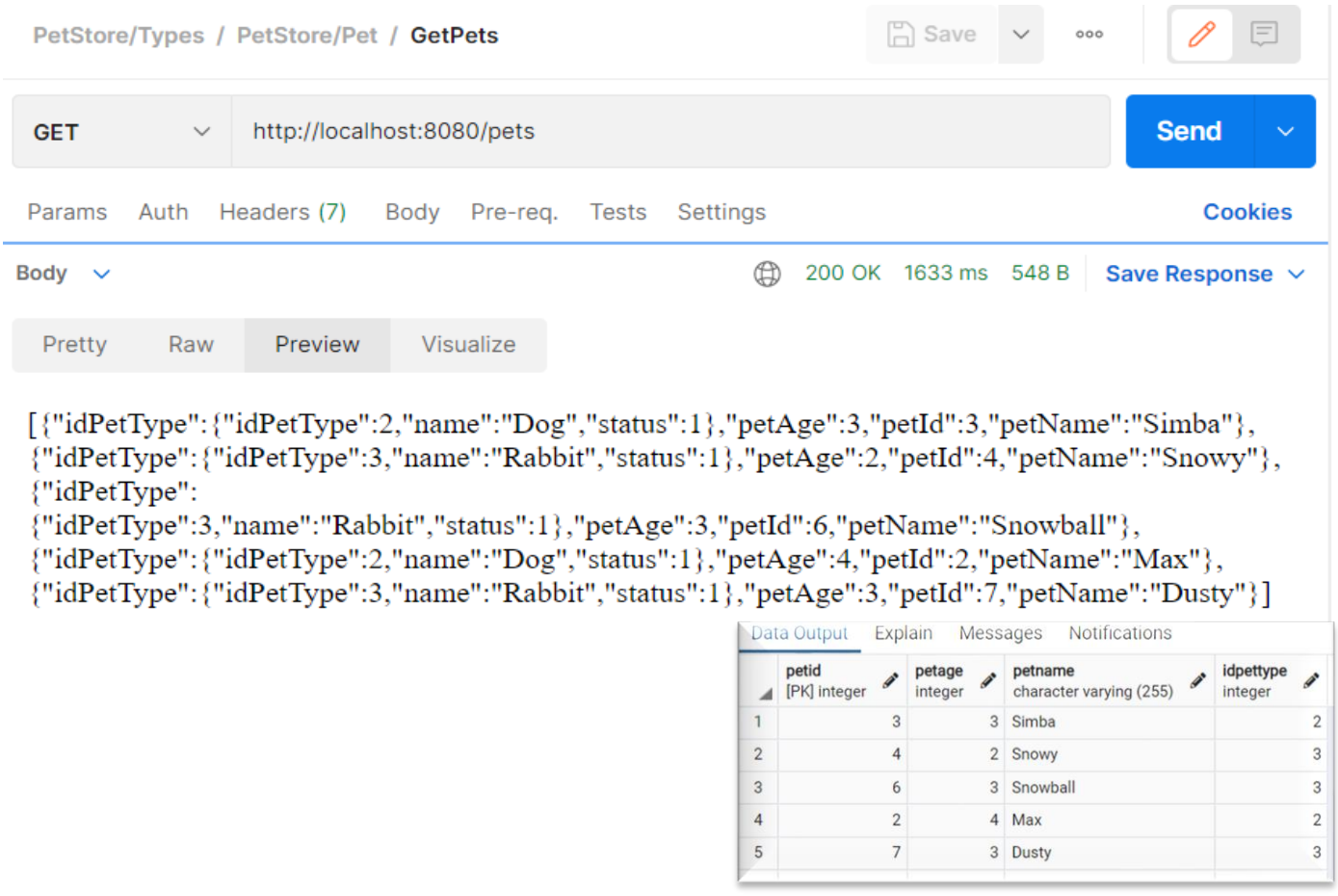


Figure 7.9: - **Testing (Pet Table):** Retrieve all existing Pet's credentials in the Table.

Test Case No 10:- Searching All PetType Credentials From **PetStore API**

Test Case: - Deletion	**Description: -** Deleting PetId Pet via PetStore API		
Input Steps	Expected Output	Actual Output	Result (Pass/Fail)
1)Provide PetId to Delete a Pet. **PetName=** Steel **PetId =** 5	**Get the Petid you wanted to delete and passing the id.After execution Relevant id entry is deleted.**	**Retrieve PetId wanted to delete & execute via Quarkus. Successfully executed and relevant entry is deleted from table.**	**Pass**
Remark of Errors: No Errors, Successfully Selection Query Executed (Results Below)			

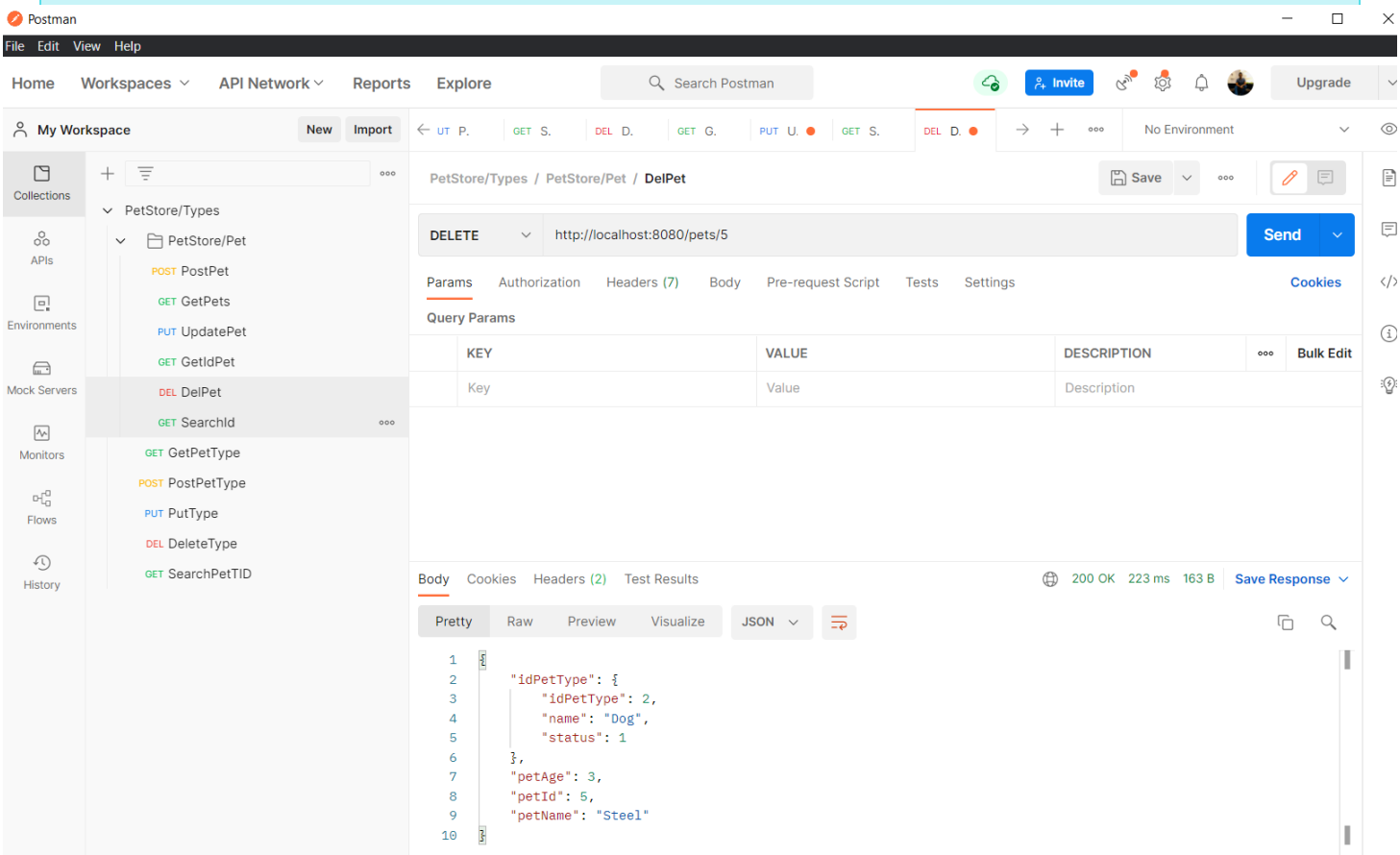


Figure 7.10: - **Testing (Pet Table):** Deleting existing Pet's credentials in the Table. (PetId)

Data Output | Explain | Messages | Notifications

	petid [PK] integer	petage integer	petname character varying (255)	idpettype integer
1	3	3	Simba	2
2	4	2	Snowy	3
3	5	3	Steel	2
4	6	3	Snowball	3
5	2	4	Maxy	2

Before **Petid =** 5 Deletion

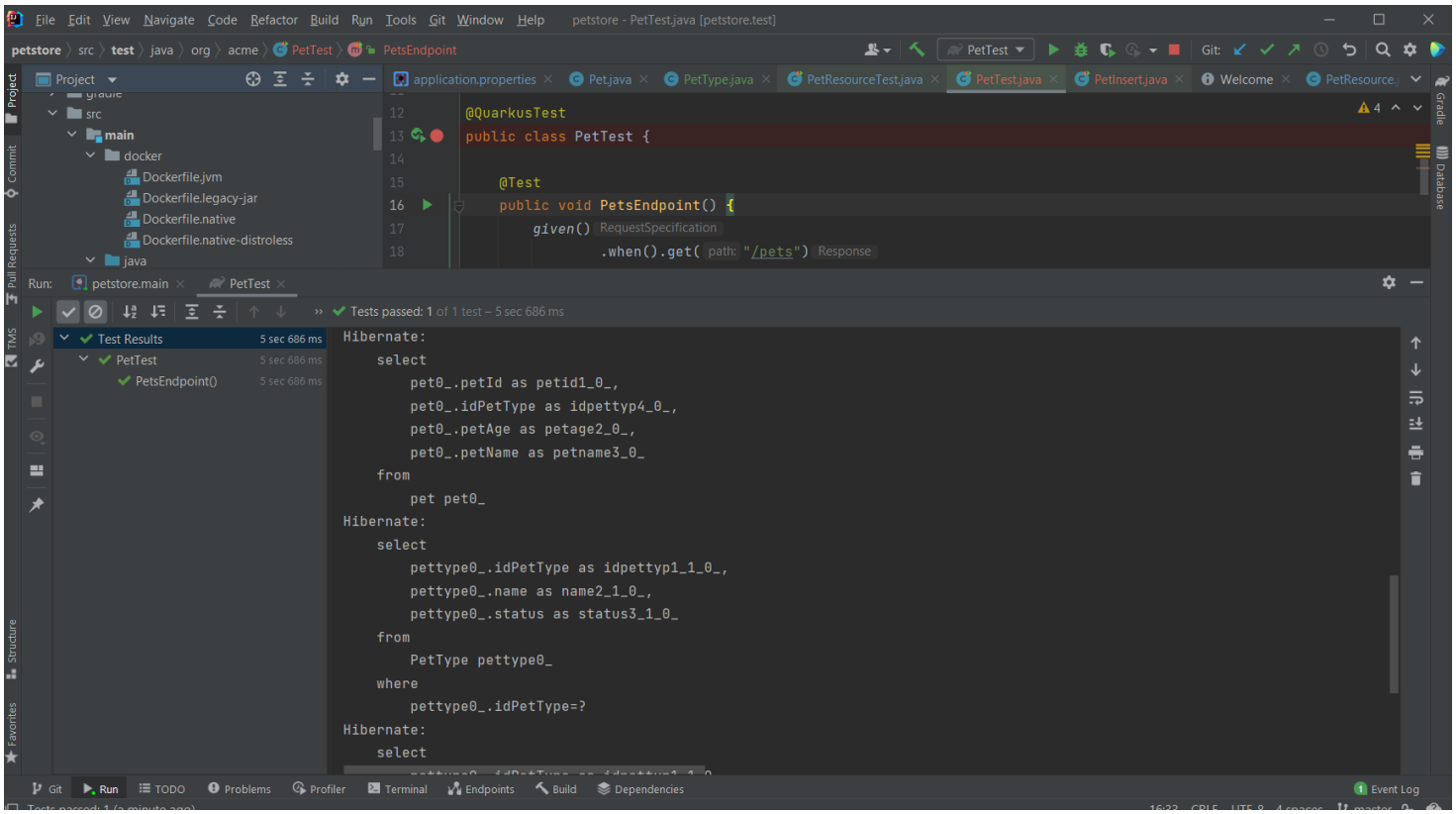
Data Output | Explain | Messages | Notifications

	petid [PK] integer	petage integer	petname character varying (255)	idpettype integer
1	3	3	Simba	2
2	4	2	Snowy	3
3	6	3	Snowball	3
4	2	4	Maxy	2

After **Petid =** 5 Deletion

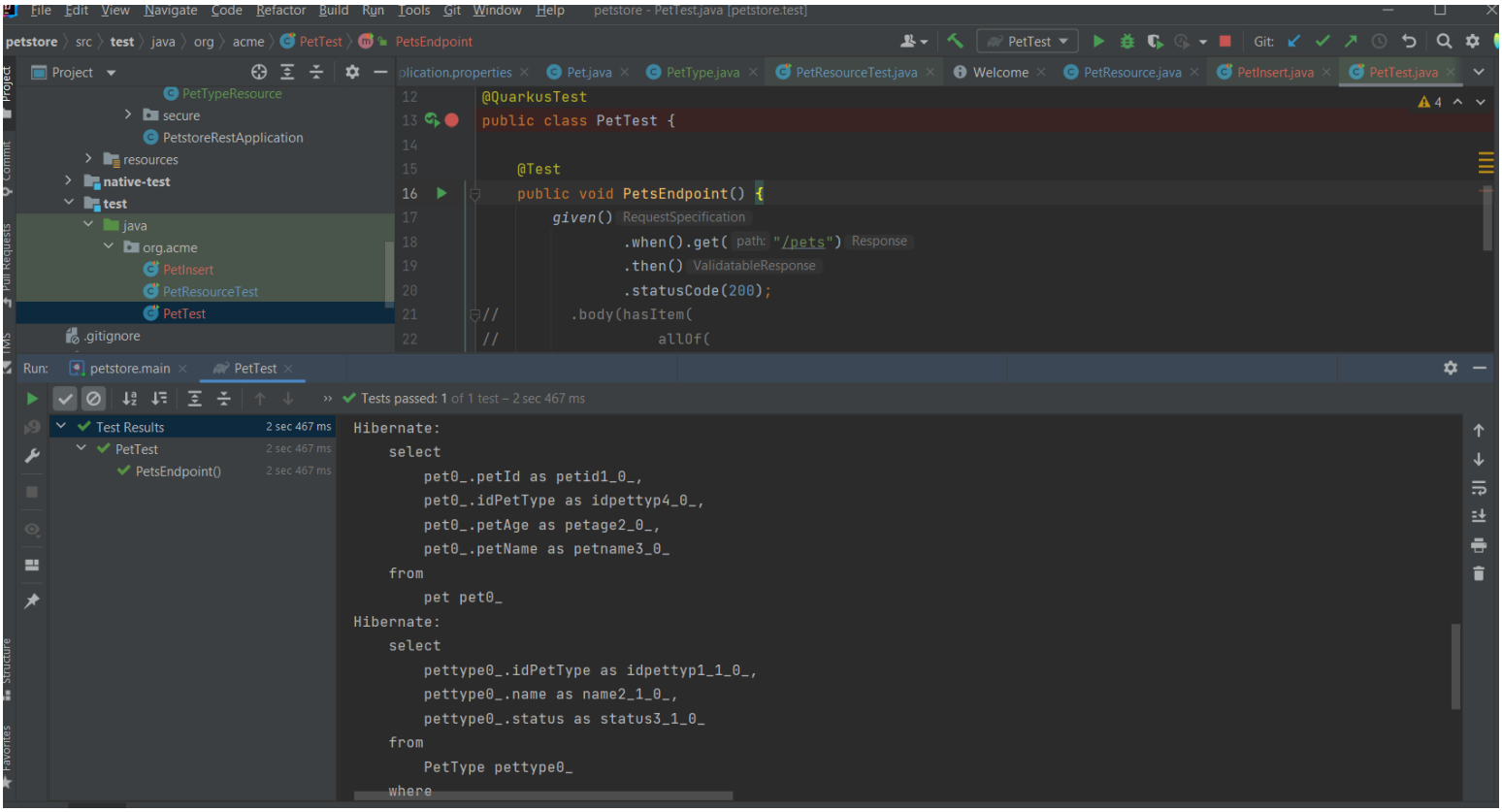
Automated Testing from IntelliJ IDEA

1) Pet type Selection Test



Result :- Passed the Unit Test Successfully

2) **Pet's Selection Test**



Result :- Passed the Unit Test Successfully

GitHub Repositary for the API Implementation for Pet Store API

- Repo Link :- [Repositary Link]
- Git Clone Link:- https://github.com/Pandula1234/Pet-Store-API-MiddlewareProject.git

In the Github all the Implementation files including the installtion process & Test Cases stored in **Read.Me file.**

Referances

1) Quarkus - Supersonic Subatomic Java - https://quarkus.io/

2) Spring Boot vs Quarkus [Baeldung] [Auguest 15, 2021] - https://www.baeldung.com/spring-boot-vs-quarkus

3) Microservices: Quarkus vs Spring Boot [Ualter Junior] [October 13, 2021] - https://dzone.com/articles/microservices-quarkus-vs-spring-boot

4) Java EE vs J2EE vs Jakarta EE [Rodrigo Graciano] [September 4, 2020] – https://www.baeldung.com/java-enterprise-evolution

5) JAKARTA EE COMPATIBLE PRODUCTS - https://jakarta.ee/compatibility/

6) MicroProfile - Optimizing Enterprise Java for a Microservices Architecture - https://microprofile.io/

7) Java Documentation - https://docs.oracle.com/en/java/