**Chapter 6**

**API Platform and Data Handler**

Sanjay Patnia\*

a UCSC Extension Santa Clara, California, USA

**Abstract**

This chapter starts with API Platform Architecture and then gets into data handler pattern for integration of RESTful APIs with actual data sources within enterprise to make it more meaningful to the consumers through APIs.

**Keywords**

Business Logic; Unstructured Data; Platform Architecture; NoSQL Database; Persistence Mechanism.

This chapter starts with API Platform Architecture and then gets into data handler pattern for integration of RESTful APIs with actual data sources within enterprise to make it more meaningful to the consumers through APIs.

**API Platform Architecture**

API Platforms are used by API Providers to realize APIs efficiently. We will review the following:

* Why do we need API Platform?
* What is an API Platform?
* Which capabilities does an API Platform have?
* How is API Platform organized? What is the architecture of API Platform?
* How does API architecture fit in the surrounding technical architecture of an Enterprise?

**Why do we need API Platform?**

It is certainly technically feasible to build APIs without any platform or framework. But why would you? For a moment, let's think about databases, which provide a platform for building applications. You could certainly build your application without a database and write your own data storage library. But we typically do not do that. We use an existing database as a platform. And this is the best practice for good reasons. It allows us to focus on building an application that serves the business case, because we can reuse existing, proven components and build the application quicker. The same augmentation applies to API platforms: API platforms allow us to focus on building APIs that consumers love, since we can reuse existing, proven API building blocks and build APIs quicker.

**So what is an API Platform?**

An API Platform consists of one of the following three components:

* API development platform
* It offers tools to design and development of APIs quicker.
* It offers building blocks, which are proven, reusable, and configurable.
* API runtime platform
* This primarily executes API.
* It serves API responses for incoming API requests of the consumers with non-functional properties like high throughput and low latency.
* API engagement platform
* This platform allows API providers to manage their interaction with API consumers. It offers API Documentation, Credentials, and Rate plans for the consumers.

**So which capabilities does the API platform have?**

The following are the capabilities offered by three components of API Platform:

**API Development Platform**

API development platform offers a toolbox for API design and development targeted for API developers who works for API Providers. Toolbox contains API building blocks, which are proven, reusable, and configurable. When building APIs, certain functionality is needed over and over again. This can be accomplished by building blocks. Building blocks can be reused. Building blocks are tested so bugs are not there and these are configurable so they can be adopted for many purposes. The building blocks offered by API Development Platform span the following features at the minimum:

* Processing of HTTP requests and responses
* Header
* Query
* HTTP: status code
* Methods
* Security: IP-based access limitation, location-based access limitation, time-based access limitation, front-end authentication and authorization, OAuth, basic authorization, API key, back-end authentication and authorization (with LDAP, SAML)
* Front-end protocols: HTTP (REST), SOAP, RPC, RMI
* Data format transformation: XML to JSON and JSON to XML
* Structural transformation: XLST, XPATH
* Data integrity and protection: encryption
* Routing to one or more back ends
* Aggregation of multiple APIs and or multiple back ends
* Throttling to protect back-end rate limitation and throughput limitation
* Load balancing for incoming requests to the API platform and outgoing requests to the back ends
* Hooks for logging
* Hooks for analytics
* Monetization capabilities
* Language for implementing APIs: Java, JavaScript, etc. (Jersey, Restlet, Spring)
* IDE for API development with editor, debugger, and deployment tools: Eclipse, JDeveloper, NetBeans
* Language for designing APIs: YAML, RAML, etc.
* Design tools for creating API interface designs: RAML, Swagger, Blueprint
* Tools for generating documentation and API code skeletons based upon design: RAML, Swagger

API Runtime Platform: API Runtime platform primarily executes APIs. It enables the APIs to accept incoming requests from API consumers and serve responses.

* It should deliver non-functional properties like:
* High availability, high security, high throughput
* To meet above these properties platform offers:
* Load balancing
* Connection pooling
* Caching
* It should also offer capabilities for Monitoring of API, Logging, and Analytics to check desired non-functional properties are met.

**API Engagement Platform**

API Engagement Platform is used by API providers to interact with its community of API consumers. API Providers use the following capabilities of API Engagement Platform:

* API management: configuration and reconfiguration of APIs without need for deployment
* API discovery: a mechanism for clients to obtain information about APIs.
* Consumer onboarding: app key generation, API Console
* Community management: blogs
* Documentation
* Version management
* Management of monetization and service-level SLAs

API Consumers use engagement platform for:

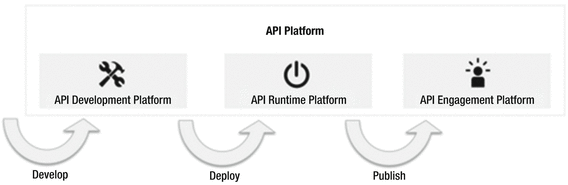
* Overview of API portfolio
* Documentation of APIs
* Possibility of trying API interactively
* Example source code for integration
* Self service to get access to APIs
* Client tooling, such as code generation for client

**How is API Platform organized? What is architecture of API Platform?**

Usually, APIs are not only deployed on the production system, but need to be deployed on different stages of increasing maturity. The stages are also sometimes called environments. Each of the stages has a specific purpose and is separated from the other stages to isolate potential errors.

* Simulation: used for playing with interface design, provides mocks or simulation of an API
* Development: used for development, which will eventually go to production
* Testing: used for manual black box testing and integration testing
* Pre-production: used as a practice for production and for acceptance testing
* Production: used as a real system for consumers

As shown in Figure [6-1](#bookmark), API Development Platform is used for Design and Development. API Runtime Platform is used for deployment. API Engagement Platform is used for publishing the API.



***Figure 6-1.*** *API Platform architecture*

**How does API architecture fit in surrounding technical architecture of an Enterprise?**

API Platform is not isolated but it needs to be integrated in existing architecture in the enterprise. Firewall is used to improve security. Load balancers are used to improve performance, and are usually placed between the Internet and the API platform. IAM (Identity and Access Management) systems are for managing identity information and LDAP or Active Directory as shown in Figure [6-2](#bookmark1).



***Figure 6-2.*** *API Architecture in Enterprise*

Back-end systems for providing the core functionality of the enterprise: Back-end systems form the heart of the enterprise data and services typically reside in the back-end system. Back ends may be databases, applications, enterprise service buses, web services using SOAP, message queues, and REST services.

**Data Handler**

As mentioned in previous section, we use an existing database as a platform. A Data Handler, a Data Access Object (DAO), or Command Query Responsibilities segmentation (CQRS) all provide an abstract interface to some type of database or any other persistence mechanism. Data Handler is a layer which handles data in the framework. Data Access Object is a design pattern used to implement the access from the database inside data handler. CQRS pattern, on the other hand, provides a mechanism to segment query and transactional data in the data handler.

**Data Access Object**

By mapping application calls to the persistence layer, a DAO provides some specific data operations without exposing details of the database. The advantage of using data access objects is the relative simplicity and it provides separation between two important parts of an application that can but should not know anything about each other, and which can be expected to evolve frequently and independently. Changing business logic can rely on the same DAO interface, while changes to persistence logic do not affect DAO clients as long as the interface remains correctly implemented. All details of storage are hidden from the rest of the application (see information hiding). Thus, possible changes to the persistence mechanism can be implemented by just modifying one DAO implementation while the rest of the application isn't affected. DAOs act as an intermediary between the application and the database. DAOs move data back and forth between objects and database records.

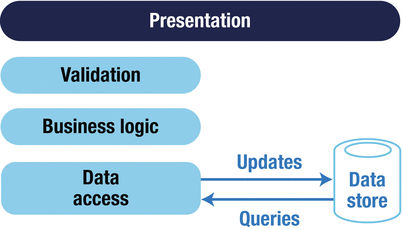
For accessing databases, there are different APIs available (for example, JPA, which will be used in class lab).

**Command Query Responsibilities Segmentation - CQRS**

New demands are being put on IT organizations every day to deliver agile, high-performance integrated mobile and web applications. In the meantime, the technology landscape is getting complex every day with the advent of new technologies like REST, NoSQL, and Cloud, while existing technologies like SOAP and SQL still rule everyday work. Rather than taking a religious side of the debate, NoSQL can successfully co-exist with SQL in this “polyglot” of data storage and formats. However, this integration also adds another layer of complexity both in architecture and implementation. We will talk about the following:

**SQL Development Process**

The application development lifecycle means changes to the database schema first, followed by the bindings, then internal schema mapping, and finally the SOAP or JSON services, and eventually the client code. This all costs the project time and money. It also means that the “code” (pick your language here) and the business logic would also need to be modified to handle the changes to the model. Figure [6-3](#bookmark2) shows the traditional CRUD architecture.



***Figure 6-3.*** *Traditional CRUD architecture*

**NoSQL Process**

NoSQL is gaining supporters among many SQL shops for various reasons including low cost, the ability to handle unstructured data, scalability, and performance. The first thing database folks notice is that there is no schema. These document-style storage engines can handle huge volumes of structured, semi-structured, and unstructured data. The very nature of schema-less documents allows change to a document structure without having to go through the formal change management process (or data architect).

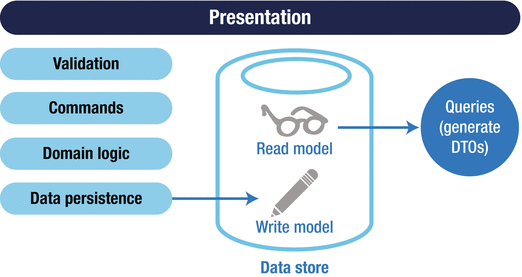
**Do I have to choose between SQL and NoSQL?**

The bottom line is both have their place and are suited for certain types of data—SQL for structured data and NoSQL for unstructured data. NoSQL databases are more scalable than SQL databases. So why not have the capability to mix and match this data depending on the application? This can be done by creating a single REST API across both SQL and NoSQL databases.

**Why a single REST API?**

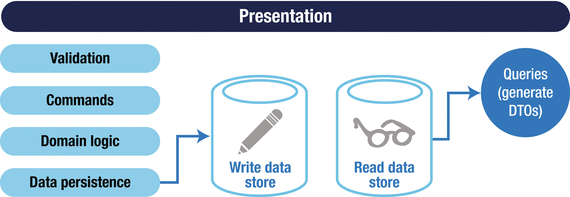
The answer is simple—the new agile and mobile world demands this “mash-up” of data into a document-style JSON response.

Martin Fowler described the pattern called “CQRS” that is more relevant today in a “polyglot” of servers, data, services, and connections (Figure [6-4](#bookmark3)).



**Figure 6-4.** Basic CQRS architecture

In this design pattern, the REST API requests (GET) return documents from multiple sources (e.g., mash-ups). In the update process, the data is subject to business logic derivations, validations, event processing, and database transactions. This data may then be pushed back into the NoSQL using asynchronous events. The advantage of NoSQL databases over SQL for this purpose is that NoSQL has dynamic schema for unstructured data. Also, NoSQL databases are horizontally scalable, which means NoSQL databases are scaled by increasing the database servers in the pool of resources to reduce the load, whereas SQL databases are scaled by increasing horsepower of the server where the database is hosted. Figure [6-5](#bookmark4) shows CQRS architecture with a separate read and write store. When you have a requirement of very, very large data volumes, you would choose separate stores.



**Figure 6-5.** CQRS architecture with separate read and write store

**Framework - Data Handler**

This exercise will implement data handler or data access object for the quote domain object using Java Persistence API (JPA) . The JPA is a Java specification for accessing, persisting, and managing data between Java objects/classes and a relational database. We will use our domain object message and implement CRUD operations using JPA in DAO.

**Pom.xml**

Create pom.xml with following dependencies.

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.rest</groupId>

<artifactId>quote</artifactId>

<version>0.1</version>

<packaging>${packaging}</packaging>

<parent>

<groupId>io.micronaut</groupId>

<artifactId>micronaut-parent</artifactId>

<version>3.4.3</version>

</parent>

<properties>

<packaging>jar</packaging>

<jdk.version>11</jdk.version>

<micronaut.version>3.4.3</micronaut.version>

<micronaut.data.version>3.3.0</micronaut.data.version>

<exec.mainClass>com.rest.Application</exec.mainClass>

<micronaut.runtime>netty</micronaut.runtime>

</properties>

<repositories>

<repository>

<id>central</id>

<url>https://repo.maven.apache.org/maven2</url>

</repository>

</repositories>

<dependencies>

<dependency>

<groupId>io.micronaut</groupId>

<artifactId>micronaut-inject</artifactId>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>io.micronaut</groupId>

<artifactId>micronaut-validation</artifactId>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-api</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-engine</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>io.micronaut.test</groupId>

<artifactId>micronaut-test-junit5</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>io.micronaut</groupId>

<artifactId>micronaut-http-client</artifactId>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>io.micronaut</groupId>

<artifactId>micronaut-http-server-netty</artifactId>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>io.micronaut</groupId>

<artifactId>micronaut-jackson-databind</artifactId>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>io.micronaut</groupId>

<artifactId>micronaut-http-server-netty</artifactId>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>io.micronaut</groupId>

<artifactId>micronaut-jackson-databind</artifactId>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>io.micronaut</groupId>

<artifactId>micronaut-runtime</artifactId>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>io.micronaut.data</groupId>

<artifactId>micronaut-data-jdbc</artifactId>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>io.micronaut.reactor</groupId>

<artifactId>micronaut-reactor</artifactId>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

<dependency>

<groupId>io.micronaut.reactor</groupId>

<artifactId>micronaut-reactor-http-client</artifactId>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>io.micronaut.sql</groupId>

<artifactId>micronaut-jdbc-hikari</artifactId>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>io.micronaut.data</groupId>

<artifactId>micronaut-data-hibernate-jpa</artifactId>

</dependency>

<dependency>

<groupId>io.micronaut.sql</groupId>

<artifactId>micronaut-hibernate-jpa</artifactId>

</dependency>

<dependency>

<groupId>io.swagger.core.v3</groupId>

<artifactId>swagger-annotations</artifactId>

</dependency>

<dependency>

<groupId>jakarta.annotation</groupId>

<artifactId>jakarta.annotation-api</artifactId>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>io.micronaut.build</groupId>

<artifactId>micronaut-maven-plugin</artifactId>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<!-- Uncomment to enable incremental compilation -->

<!-- <useIncrementalCompilation>false</useIncrementalCompilation> -->

<annotationProcessorPaths combine.children="append">

<path>

<groupId>io.micronaut</groupId>

<artifactId>micronaut-http-validation</artifactId>

<version>${micronaut.version}</version>

</path>

<path>

<groupId>io.micronaut.data</groupId>

<artifactId>micronaut-data-processor</artifactId>

<version>${micronaut.data.version}</version>

</path>

</annotationProcessorPaths>

<compilerArgs>

<arg>-Amicronaut.processing.group=com.rest</arg>

<arg>-Amicronaut.processing.module=quote</arg>

</compilerArgs>

</configuration>

</plugin>

</plugins>

</build>

</project>

**Product**

Here is a POJO defining properties of product or catalog.

package com.rest.domain;

import io.swagger.v3.oas.annotations.media.Schema;

import javax.persistence.\*;

import javax.validation.constraints.Size;

@Schema(description="Product")

@Entity

public class Product {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name="ID")

private Long id;

@Column(name="NAME")

@Size(max = 20)

private String name;

@Column(name="DESCRIPTION")

@Size(max = 50)

private String description;;

@Column(name="CREATE\_DATE")

@Size(max = 40)

private String createDate;;

@Column(name="CHANGE\_DATE")

@Size(max = 40)

private String changeDate;;

@Column(name="UNIT\_PRICE")

@Size(max = 20)

private float unitPrice;;

@Column(name="CREATOR")

private String creator;

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getDescription() {

return description;

}

public void setDescription(String description) {

this.description = description;

}

public String getCreateDate() {

return createDate;

}

public void setCreateDate(String createDate) {

this.createDate = createDate;

}

public String getChangeDate() {

return changeDate;

}

public void setChangeDate(String changeDate) {

this.changeDate = changeDate;

}

public float getUnitPrice() {

return unitPrice;

}

public void setUnitPrice(float unitPrice) {

this.unitPrice = unitPrice;

}

public String getCreator() {

return creator;

}

public void setCreator(String creator) {

this.creator = creator;

};

}

**Quote**

package com.rest.domain;

import io.swagger.v3.oas.annotations.media.Schema;

import javax.persistence.\*;

import javax.validation.constraints.Size;

import java.util.List;

@Schema(description="Quote")

@Entity

public class Quote {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name="ID")

private Long id;

@Column(name="CUSTOMER\_ID")

private Long customerId;

@Column(name="QUOTE\_DATE")

@Size(max = 50)

private String quoteDate;;

@Column(name="BILLING\_ADDRESS")

@Size(max = 20)

private String billingAddress;

@Column(name="BILLING\_CITY")

@Size(max = 20)

private String billingCity;;

@Column(name="BILLING\_STATE")

@Size(max = 20)

private String billingState;;

@Column(name="BILLING\_COUNTRY")

@Size(max = 20)

private String billingCountry;;

@Column(name="BILLING\_POSTAL\_CODE")

@Size(max = 20)

private String billingPostalCode;;

@Column(name="TOTAL")

@Size(max = 20)

private float total;

@OneToMany (fetch = FetchType.EAGER, cascade = CascadeType.ALL)

@JoinTable(name = "Quote\_Line\_Mapping",

joinColumns = @JoinColumn(name = "quote\_id"),

inverseJoinColumns = @JoinColumn(name = "id"))

private List<QuoteLine> quoteLines;

public void setQuoteLines(List<QuoteLine> quoteLines) {

this.quoteLines = quoteLines;

}

public List<QuoteLine> getQuoteLines() {

return quoteLines;

}

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public Long getCustomerId() {

return customerId;

}

public void setCustomerId(Long customerId) {

this.customerId = customerId;

}

public String getQuoteDate() {

return quoteDate;

}

public void setQuoteDate(String quoteDate) {

this.quoteDate = quoteDate;

}

public String getBillingAddress() {

return billingAddress;

}

public void setBillingAddress(String billingAddress) {

this.billingAddress = billingAddress;

}

public String getBillingCity() {

return billingCity;

}

public void setBillingCity(String billingCity) {

this.billingCity = billingCity;

}

public String getBillingState() {

return billingState;

}

public void setBillingState(String billingState) {

this.billingState = billingState;

}

public String getBillingCountry() {

return billingCountry;

}

public void setBillingCountry(String billingCountry) {

this.billingCountry = billingCountry;

}

public String getBillingPostalCode() {

return billingPostalCode;

}

public void setBillingPostalCode(String billingPostalCode) {

this.billingPostalCode = billingPostalCode;

}

public float getTotal() {

return total;

}

public void setTotal(float total) {

this.total = total;

};

**QuoteLine**

package com.rest.domain;

import io.swagger.v3.oas.annotations.media.Schema;

import javax.persistence.\*;

@Schema(description="QuoteLine")

@Entity

public class QuoteLine {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name="ID")

private Long id;

@Column(name="QUOTE\_ID")

private Long quoteId;

@OneToOne(cascade = CascadeType.ALL)

@JoinColumn(name = "product\_id", referencedColumnName = "id")

private Product product;

@Column(name="UNIT\_PRICE")

private float unitPrice;

@Column(name="QUANTITY")

private Long quantity;

public Product getProduct() {

return product;

}

public void setProduct(Product product) {

this.product = product;

}

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public void setId(Long id) {

this.id = id;

}

public Long getQuoteId() {

return quoteId;

}

public void setQuoteId(Long quoteId) {

this.quoteId = quoteId;

}

public float getUnitPrice() {

return unitPrice;

}

public void setUnitPrice(float unitPrice) {

this.unitPrice = unitPrice;

}

public Long getQuantity() {

return quantity;

}

public void setQuantity(Long quantity) {

this.quantity = quantity;

};

}

Create Repositories

**ProductRepo**

package com.rest.repository;

import io.micronaut.data.annotation.Repository;

import io.micronaut.data.repository.CrudRepository;

import com.rest.domain.Product;

@Repository

public interface ProductRepo extends CrudRepository<Product, Long> {

}

**QuoteRepo**

package com.rest.repository;

import io.micronaut.data.annotation.Repository;

import io.micronaut.data.repository.CrudRepository;

import com.rest.domain.Quote;

@Repository

public interface QuoteRepo extends CrudRepository<Quote, Long> {

}

**QuoteLineRepo**

package com.rest.repository;

import io.micronaut.data.annotation.Repository;

import io.micronaut.data.repository.CrudRepository;

import com.rest.domain.QuoteLine;;

@Repository

public interface QuoteLineRepo extends CrudRepository<QuoteLine, Long> {

}

Create controller

**QuoteController**

package com.rest.controller;

import com.rest.domain.Quote;

import com.rest.repository.QuoteRepo;

import io.micronaut.http.annotation.Get;

import io.micronaut.http.annotation.Controller;

import io.micronaut.http.annotation.Delete;

import io.micronaut.http.annotation.Post;

import io.micronaut.http.annotation.Put;

import io.micronaut.http.annotation.Body;

import java.util.List;

import java.util.ArrayList;

@Controller("/quote") // <2>

public class QuoteController {

QuoteRepo quoteRepo;

public QuoteController(QuoteRepo quoteRepo) { // <3>

this.quoteRepo = quoteRepo;

}

@Post

public Quote createQuote(@Body Quote quote) {

return quoteRepo.save(quote);

}

@Get("/{id}")

public Quote getQuote (Long id) {

Quote quote = quoteRepo.findById(id).get();

return quote;

}

@Get

public List<Quote> getQuotes() {

Iterable<Quote> quotes = quoteRepo.findAll();

List<Quote> result = new ArrayList<Quote>();

quotes.forEach(result::add);

return result;

}

@Put("/{id}")

public void updateQuote (Long id, Quote update) {

Quote quote = quoteRepo.findById(id).get();

quoteRepo.delete(quote);

quoteRepo.save(update);

}

@Delete("/{id}")

public void deleteQuote(Long id) {

Quote quote = quoteRepo.findById(id).get();

quoteRepo.delete(quote);

}

}

Creating quote

curl -d '{ "customerId":"123", "quoteDate":"11/07/2022", "billingAddress":"722 Main St", "billingCity":"San Jose", "billingState": "CA", "billingCountry": "USA", "billingPostalCode": "95035", "total" : 123, "quoteLines" : [{"quoteId" : 1, "product" : {"name":"test", "description":"test", "createDate":"test", "changeDate":"12/12//2012", "unitPrice": 1.0, "creator": "creat" }, "unitPrice": 12, "quantity" : 1}]}' -H 'Content-Type: application/json' <http://localhost:8080/quote>

Reading quote

curl http://localhost:8080/quote/1

**To Review**

In this chapter we started with API Platform Architecture and then got into data handler pattern for integration of RESTful APIs with actual data sources. In the exercise we demonstrated implementation of Data Handler using JPA.