7-Types of APIs

Author: Vincent Lau

Note: This material is intended for educational purposes only. All rights reserved. Any unauthorized sharing or copying of this material, in any form, to any individual or party, for any use without prior permission, is strictly prohibited.

Learning Objectives

- Have a basic understanding of different types of API

Introduction

There are several types of APIs, which have different architectural styles for designing web services. We need different types of APIs to cater to a diverse range of application needs and scenarios. Each type of API serves specific purposes, optimizing performance, flexibility, and compatibility.

For example, RESTful APIs simplify web service interactions, GraphQL allows flexible data querying, WebSocket supports real-time communication, gRPC offers efficient cross-language communication, SOAP enforces strong standards, and RMI facilitates Java-based distributed computing.

By having various API types available, developers can choose the most suitable one for their project, ensuring efficient data exchange and functionality for different use cases.

In our bootcamp, we would focus on learning the implementation of RESTful API by Spring Boot framework.

Types of API

Here's an introduction to various types of APIs, including RESTful, GraphQL, WebSocket, gRPC, SOAP, and RMI.

RESTful API (Representational State Transfer)

Description: RESTful APIs are designed based on the principles and constraints of REST architecture. They use standard HTTP methods (GET, POST, PUT, DELETE) to interact with resources identified by URIs. They emphasize statelessness, a uniform interface, and resource representations.

Use Cases: Commonly used for web services and APIs where simplicity, scalability, and ease of use are important.

GraphQL API

Description: GraphQL is a query language and runtime for APIs. It allows clients to request precisely the data they need, reducing over-fetching and under-fetching of data. Clients can shape the structure of the response.

Use Cases: Ideal for applications requiring flexible data querying and when multiple data sources need to be combined.

gRPC

Description: gRPC is a high-performance RPC (Remote Procedure Call) framework developed by Google. It uses Protocol Buffers (protobuf) for data serialization and offers features like bi-directional streaming and authentication.

Use Cases: Suitable for building efficient, language-agnostic APIs and microservices.

WebSocket API

Description: WebSocket is a communication protocol that provides full-duplex, bidirectional communication channels over a single TCP connection. It enables real-time, interactive communication between clients and servers.

Use Cases: Used for real-time applications such as chat applications, online gaming, and live data streaming.

SOAP (Simple Object Access Protocol)

Description: SOAP is a protocol for exchanging structured information in the implementation of web services. It uses XML for message format and relies on HTTP or other transport protocols for communication.

Use Cases: Often used in enterprise environments and scenarios requiring strong typing and comprehensive standards.

RMI (Remote Method Invocation)

Description: RMI is a Java-based API that allows remote method calls between Java objects in different Java virtual machines (JVMs). It enables distributed computing in Java applications.

Use Cases: Primarily used in Java applications for distributed computing and inter-JVM communication.

In summary, these are some of the common types of APIs, each tailored to specific use cases and requirements. The choice of API type depends on factors like the nature of the application, the technologies in use, and the specific goals of the API. Each type has its advantages and trade-offs, so selecting the right one is crucial for a successful software project.

Code Examples

RESTful API

Here's a simple example of creating a RESTful API using Spring Boot. In this example, we'll build a RESTful API for managing a list of tasks. We'll demonstrate how to create endpoints for creating, reading, updating, and deleting tasks.

Task Model

Create a `Task` model class to represent tasks.

```
1 package com.example.demo.model;
2
3 public class Task {
4     private Long id;
5     private String title;
6     private String description;
7
8     // Getters and setters
9 }
```

Controller

Create a controller class that defines the RESTful endpoints. For this example, let's create a `TaskController`:

```java
package com.example.demo.controller;

import com.example.demo.model.Task;
import org.springframework.web.bind.annotation.*;

import java.util.ArrayList;
import java.util.List;

@RestController
@RequestMapping("/tasks")
public class TaskController {

    private List<Task> tasks = new ArrayList<>();

    @GetMapping("/")
    public List<Task> getAllTasks() {
        return tasks;
    }

    @GetMapping("/{id}")
    public Task getTaskById(@PathVariable Long id) {
        // Implement logic to find and return a task by ID
        // Return null or handle errors if the task is not found
    }

    @PostMapping("/")
    public Task createTask(@RequestBody Task task) {
        // Implement logic to create a new task
        // Assign a unique ID and add it to the list of tasks
        // Return the created task
    }

    @PutMapping("/{id}")
    public Task updateTask(@PathVariable Long id, @RequestBody Task task) {
        // Implement logic to update an existing task by ID
        // Return the updated task
    }

    @DeleteMapping("/{id}")
    public void deleteTask(@PathVariable Long id) {
        // Implement logic to delete a task by ID
```

```
        // Handle errors if the task is not found
    }
}
```

Run your Spring Boot application, and it will start an embedded Tomcat server. Your RESTful API for managing tasks will be accessible at the specified endpoints (e.g., `http://localhost:8080/tasks`).

GraphQL API

Creating a GraphQL API in Spring Boot involves integrating the Spring framework with GraphQL. You can use libraries like `graphql-java`, which provides the core functionality for building GraphQL APIs in Java applications. Here's a step-by-step example of creating a simple GraphQL API using Spring Boot:

Data Model

For this example, let's assume you're building a GraphQL API for managing books. Create a `Book` entity class:

```
package com.example.demo.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;
    private String author;

    // Getters and setters
}
```

Create a Repository

Create a repository interface for accessing `Book` data. Spring Data JPA will provide the implementation:

```java
package com.example.demo.repository;

import com.example.demo.model.Book;
import org.springframework.data.jpa.repository.JpaRepository;

public interface BookRepository extends JpaRepository<Book, Long> {
}
```

Define GraphQL Schema

Create a GraphQL schema for your API. In this example, we'll use GraphQL's SDL (Schema Definition Language) to define the schema. You can use the `graphql-java` library for this:

```
# src/main/resources/schema.graphqls

type Book {
    id: ID!
    title: String!
    author: String!
}

type Query {
    getAllBooks: [Book]
    getBookById(id: ID!): Book
}
```

Implement GraphQL Resolvers

Create resolvers to fetch data for the defined GraphQL queries. You'll need a `GraphQLDataFetchers` class to implement the resolvers:

```java
package com.example.demo.datafetchers;

import com.example.demo.model.Book;
import com.example.demo.repository.BookRepository;
import graphql.schema.DataFetcher;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import java.util.List;

@Component
public class GraphQLDataFetchers {
```

```
13
14      @Autowired
15      private BookRepository bookRepository;
16
17      public DataFetcher<List<Book>> getAllBooksDataFetcher() {
18          return dataFetchingEnvironment -> bookRepository.findAll();
19      }
20
21      public DataFetcher<Book> getBookByIdDataFetcher() {
22          return dataFetchingEnvironment -> {
23              Long id =
    Long.parseLong(dataFetchingEnvironment.getArgument("id"));
24              return bookRepository.findById(id).orElse(null);
25          };
26      }
27  }
```

Configure GraphQL Endpoint

Configure the GraphQL endpoint and wire up the schema and resolvers:

```
1   package com.example.demo;
2
3   import graphql.GraphQL;
4   import graphql.schema.GraphQLSchema;
5   import graphql.schema.idl.RuntimeWiring;
6   import graphql.schema.idl.SchemaGenerator;
7   import graphql.schema.idl.SchemaParser;
8   import graphql.schema.idl.TypeDefinitionRegistry;
9   import org.springframework.beans.factory.annotation.Autowired;
10  import org.springframework.context.annotation.Bean;
11  import org.springframework.context.annotation.Configuration;
12
13  import javax.annotation.PostConstruct;
14  import java.io.IOException;
15  import java.nio.charset.StandardCharsets;
16
17  @Configuration
18  public class GraphQLConfig {
19
20      @Autowired
21      private GraphQLDataFetchers graphQLDataFetchers;
22
23      @Bean
24      public GraphQL graphQL() {
```

```
25          return GraphQL.newGraphQL(graphQLSchema()).build();
26      }
27
28      @Bean
29      public GraphQLSchema graphQLSchema() {
30          String sdl = loadSchemaFile("schema.graphqls");
31          TypeDefinitionRegistry typeRegistry = new SchemaParser().parse(sdl);
32          RuntimeWiring wiring = buildRuntimeWiring();
33          return new SchemaGenerator().makeExecutableSchema(typeRegistry,
    wiring);
34      }
35
36      private RuntimeWiring buildRuntimeWiring() {
37          return RuntimeWiring.newRuntimeWiring()
38                  .type("Query", typeWiring -> typeWiring
39                          .dataFetcher("getAllBooks",
    graphQLDataFetchers.getAllBooksDataFetcher())
40                          .dataFetcher("getBookById",
    graphQLDataFetchers.getBookByIdDataFetcher()))
41                  .build();
42      }
43
44      private String loadSchemaFile(String filename) {
45          try {
46              return new String(
47
    this.getClass().getClassLoader().getResourceAsStream(filename).readAllBytes(),
48                      StandardCharsets.UTF_8
49              );
50          } catch (IOException e) {
51              throw new RuntimeException("Error loading GraphQL schema file: " +
    filename, e);
52          }
53      }
54  }
```

Run your Spring Boot application. Your GraphQL API will be accessible at the endpoint `/graphql`. You can use tools like GraphiQL or Postman to test your GraphQL queries.

This example demonstrates the basics of setting up a GraphQL API with Spring Boot. You can expand upon this foundation by adding more types, queries, and mutations to your schema as needed.

gRPC

Creating a gRPC API in Spring Boot involves integrating gRPC with Spring Boot and defining gRPC services and message types. Here's a step-by-step example of creating a simple gRPC API using Spring Boot:

GRPC Dependency

Create a new Spring Boot project using Spring Initializer or your preferred IDE. Include the "Spring Web" and "gRPC" dependencies, `grpc-core` & `grpc-netty`.

gRPC Service and Message Types

Define your gRPC service and message types using Protocol Buffers (protobuf). Create a `.proto` file (e.g., `book.proto`) in the `src/main/proto` directory:

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "com.example.grpc";
option java_outer_classname = "BookServiceProto";

package com.example.grpc;

service BookService {
    rpc GetAllBooks (Empty) returns (BookList);
}

message Empty {}

message Book {
    string id = 1;
    string title = 2;
    string author = 3;
}

message BookList {
    repeated Book books = 1;
}
```

Generate Java Code from .proto File

Generate Java code from the `.proto` file using the Protocol Buffers compiler (`protoc`). You can use the `protobuf-maven-plugin` to automate this task.

Add the plugin configuration to your `pom.xml`. This configuration generates Java code from the `.proto` file during the Maven build process.

```xml
<build>
    <extensions>
        <extension>
            <groupId>kr.motd.maven</groupId>
            <artifactId>os-maven-plugin</artifactId>
            <version>1.6.0</version>
        </extension>
    </extensions>
    <plugins>
        <plugin>
            <groupId>com.google.protobuf.tools</groupId>
            <artifactId>maven-protoc-plugin</artifactId>
            <version>0.6.0</version>
            <executions>
                <execution>
                    <goals>
                        <goal>compile</goal>
                        <goal>compile-custom</goal>
                    </goals>
                    <phase>generate-sources</phase>
                </execution>
            </executions>
            <configuration>

<protocArtifact>com.google.protobuf:protoc:3.17.3:exe:${os.detected.classifier}
</protocArtifact>
                <inputDirectories>
                    <include>src/main/proto</include>
                </inputDirectories>
                <outputTargets>
                    <outputTarget>
                        <type>java</type>
                        <outputDirectory>${project.build.directory}/generated-
sources/protobuf/grpc-java</outputDirectory>
                    </outputTarget>
                </outputTargets>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Implement gRPC Service

Create a gRPC service implementation in your Spring Boot application:

```java
package com.example.grpc;

import io.grpc.stub.StreamObserver;
import net.devh.boot.grpc.server.service.GrpcService;
import org.springframework.beans.factory.annotation.Autowired;

import java.util.List;

@GrpcService
public class BookServiceGrpcImpl extends BookServiceGrpc.BookServiceImplBase {

    @Autowired
    private BookRepository bookRepository;

    @Override
    public void getAllBooks(Empty request, StreamObserver<BookList> responseObserver) {
        List<Book> books = bookRepository.findAll();
        BookList bookList = BookList.newBuilder().addAllBooks(books).build();
        responseObserver.onNext(bookList);
        responseObserver.onCompleted();
    }
}
```

Configure gRPC Server

Configure the gRPC server in your Spring Boot application:

```java
package com.example.grpc;

import net.devh.boot.grpc.server.service.GrpcService;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class GrpcConfig {

    @Bean
    public ServiceDiscoveryGrpc.ServiceDiscoveryBlockingStub serviceDiscoveryBlockingStub() {
        return ServiceDiscoveryGrpc.newBlockingStub(channel());
    }

    @Bean
    public ServiceDiscoveryGrpc.ServiceDiscoveryStub serviceDiscoveryStub() {
```

```
        return ServiceDiscoveryGrpc.newStub(channel());
    }

    private ManagedChannel channel() {
        return ManagedChannelBuilder.forAddress("localhost", 6565)
                .usePlaintext()
                .build();
    }
}
```

Run your Spring Boot application. The gRPC server will be accessible at `localhost:6565`. You can use gRPC clients to interact with your service.

This example demonstrates the basics of setting up a gRPC API with Spring Boot. You can expand upon this foundation by adding more gRPC services and implementing client-side communication as needed.

SOAP API

Creating a SOAP API in Java typically involves using **Java's JAX-WS (Java API for XML Web Services)** for building web services and generating client code. Here's a step-by-step example of creating a simple SOAP API using Java:

Define a Web Service

Create a Java class that defines your web service. In this example, we'll create a `CalculatorService`:

```
import javax.jws.WebService;

@WebService
public class CalculatorService {

    public int add(int a, int b) {
        return a + b;
    }

    public int subtract(int a, int b) {
        return a - b;
    }
}
```

Publish the Web Service

To publish the web service, you can use a simple Java program. Create a class with a `main` method to publish the service:

```
import javax.xml.ws.Endpoint;

public class CalculatorServicePublisher {

    public static void main(String[] args) {
        String url = "http://localhost:8080/calculator";
        Endpoint.publish(url, new CalculatorService());

        System.out.println("CalculatorService is running at " + url);
    }
}
```

Generate Client Code

Generate client code to consume the SOAP service. You can use the `wsimport` tool, which is included with Java, to generate client code from the service's WSDL (Web Services Description Language) file.

Run the following command in your project directory, which generates client classes in the `src` directory.

```
wsimport -s src http://localhost:8080/calculator?wsdl
```

Create a SOAP Client

Now, create a SOAP client to consume the service. Here's an example:

```
import com.example.calculator.CalculatorService;
import com.example.calculator.CalculatorServiceService;

public class CalculatorClient {

    public static void main(String[] args) {
        CalculatorServiceService service = new CalculatorServiceService();
        CalculatorService calculator = service.getCalculatorServicePort();

        int resultAdd = calculator.add(10, 5);
        int resultSubtract = calculator.subtract(10, 5);

```

```
        System.out.println("Addition Result: " + resultAdd);
        System.out.println("Subtraction Result: " + resultSubtract);
    }
}
```

Run the Service and Client

Run the `CalculatorServicePublisher` class to start the SOAP service.

Then, run the `CalculatorClient` class to consume the service. You should see the results of addition and subtraction displayed in the client.

This is a basic example of creating a SOAP API in Java. In practice, you may need to configure web service security, handle exceptions, and work with more complex service operations and data types.

WebSocket API

Below is a simple WebSocket server and client example in Java using the Java API for WebSocket (javax.websocket). This example demonstrates a basic WebSocket echo server that receives messages from clients and sends them back.

WebSocket Server (Echo Server)

The `WebSocketEchoServer` class is the WebSocket server, annotated with `@ServerEndpoint("/echo")`. It defines methods annotated with `@OnOpen`, `@OnMessage`, and `@OnClose` to handle WebSocket events.

```
import javax.websocket.*;
import javax.websocket.server.ServerEndpoint;
import java.io.IOException;

@ServerEndpoint("/echo")
public class WebSocketEchoServer {

    @OnOpen
    public void onOpen(Session session) {
        System.out.println("WebSocket opened: " + session.getId());
    }

    @OnMessage
    public void onMessage(String message, Session session) {
        System.out.println("Message received: " + message);
        try {
            session.getBasicRemote().sendText("Server: " + message);
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @OnClose
    public void onClose(Session session, CloseReason closeReason) {
        System.out.println("WebSocket closed: " + session.getId() + " Reason: "
 + closeReason.getReasonPhrase());
    }
}
```

WebSocket Client

The `WebSocketClient` class is a WebSocket client that connects to the server at the specified URI (`ws://localhost:8080/your-web-app/echo`). It sends a message to the server and listens for incoming messages.

Make sure to replace `"ws://localhost:8080/your-web-app/echo"` with the actual WebSocket server endpoint URL in your environment. Additionally, you'll need to configure a web server (e.g., Apache Tomcat) to deploy and run the WebSocket server code.

```
import javax.websocket.*;
import java.net.URI;

@ClientEndpoint
public class WebSocketClient {

    @OnMessage
    public void onMessage(String message) {
        System.out.println("Received message from server: " + message);
    }

    public static void main(String[] args) {
        String serverUri = "ws://localhost:8080/your-web-app/echo"; // Replace
with your server URL
        WebSocketContainer container =
ContainerProvider.getWebSocketContainer();

        try (Session session =
container.connectToServer(WebSocketClient.class, URI.create(serverUri))) {
            session.getBasicRemote().sendText("Hello, WebSocket Server!");
            Thread.sleep(5000); // Keep the connection alive for a few seconds
        } catch (Exception e) {
            e.printStackTrace();
```

```
        }
    }
}
```

This is a basic example to get you started with WebSocket communication in Java. WebSocket is often used for more complex real-time applications like chat systems, online gaming, and live data streaming.