**Netty/jetty webflux reactive programing**

could you please explain in details process of network application server development with Netty/Jetty(preferably Netty Reactor-based Spring WebFlux - with some sample code project and then explain process of deployments

**ChatGPT said:**

ChatGPT

Memory updated

**Developing a Network Application Server with Netty (Reactor-Based Spring WebFlux)**

Netty is a high-performance, non-blocking, asynchronous I/O framework. Combined with **Spring WebFlux**, it is widely used to build reactive, scalable web applications. Let's walk through the process step-by-step.

**Why Use Netty with Spring WebFlux?**

1. **Asynchronous I/O**: Ideal for handling many simultaneous connections with minimal resource usage.
2. **Non-Blocking**: Efficient processing, no thread blocking, and better scalability.
3. **Reactor Framework**: WebFlux leverages Reactor, enabling declarative programming and backpressure.

**Step-by-Step Development**

**1. Setting Up the Project**

* Use **Spring Initializr** to generate a project:
  + Add dependencies:
    - **Spring WebFlux** (reactive programming).
    - **Spring Boot Actuator** (monitoring).
    - **Lombok** (optional for boilerplate reduction).

**pom.xml Dependencies:**

xml

Copy code

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-webflux</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>org.projectreactor</groupId>

<artifactId>reactor-netty</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

**2. Implementing a Simple Reactive Application**

**Reactive REST Controller**

java

Copy code

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RestController;

import reactor.core.publisher.Flux;

import java.time.Duration;

@RestController

public class ReactiveController {

@GetMapping("/stream")

public Flux<String> stream() {

return Flux.interval(Duration.ofSeconds(1))

.map(i -> "Message #" + i)

.take(10); // Simulates a finite stream

}

}

* **Explanation**:
  + Flux.interval creates a reactive stream emitting events every second.
  + take(10) limits the stream to 10 messages.

**3. Customizing the Netty Server**

Spring WebFlux uses Netty by default. You can customize it for specific use cases like SSL, ports, or thread tuning.

**Custom Netty Server Configuration**

java

Copy code

import org.springframework.boot.web.embedded.netty.NettyReactiveWebServerFactory;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import reactor.netty.http.server.HttpServer;

@Configuration

public class NettyConfig {

@Bean

public NettyReactiveWebServerFactory reactiveWebServerFactory() {

NettyReactiveWebServerFactory factory = new NettyReactiveWebServerFactory();

factory.addServerCustomizers(httpServer ->

httpServer.port(8081) // Custom port

.wiretap(true)); // Enables logging of requests/responses

return factory;

}

}

**4. Testing the Application**

* Run the application and access:
  + http://localhost:8081/stream

Expected output:

plaintext

Copy code

Message #0

Message #1

Message #2

...

Message #9

**Deployment Process**

**1. Package the Application**

* Use Maven to build the application:

bash

Copy code

mvn clean package

**2. Deploy on Docker**

* Create a Dockerfile:

dockerfile

Copy code

FROM openjdk:17-jdk-alpine

VOLUME /tmp

ARG JAR\_FILE=target/reactive-server.jar

COPY ${JAR\_FILE} app.jar

ENTRYPOINT ["java", "-jar", "/app.jar"]

* Build the Docker image:

bash

Copy code

docker build -t reactive-server .

* Run the container:

bash

Copy code

docker run -p 8081:8081 reactive-server

**Optional: Deploy to Kubernetes**

**1. Create Deployment and Service**

**deployment.yaml**:

yaml

Copy code

apiVersion: apps/v1

kind: Deployment

metadata:

name: reactive-server

spec:

replicas: 3

selector:

matchLabels:

app: reactive-server

template:

metadata:

labels:

app: reactive-server

spec:

containers:

- name: reactive-server

image: reactive-server:latest

ports:

- containerPort: 8081

**service.yaml**:

yaml

Copy code

apiVersion: v1

kind: Service

metadata:

name: reactive-server

spec:

selector:

app: reactive-server

ports:

- protocol: TCP

port: 80

targetPort: 8081

type: LoadBalancer

**2. Apply Kubernetes Configuration**

bash

Copy code

kubectl apply -f deployment.yaml

kubectl apply -f service.yaml

**Key Features of Netty in Spring WebFlux**

1. **High Throughput**: Handles millions of requests efficiently.
2. **Backpressure Support**: Ensures resources are not overwhelmed.
3. **Customizability**: You can fine-tune Netty’s configuration.

**Spring boot webclient simple code example GET PUT POST DELETE**

**To use WebClient, ensure you have the following dependency in your pom.xml:**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-webflux</artifactId>

</dependency>

**GET Request**

import org.springframework.web.reactive.function.client.WebClient;

import reactor.core.publisher.Mono;

public class WebClientExample {

private final WebClient webClient;

public WebClientExample(WebClient.Builder webClientBuilder) {

this.webClient = webClientBuilder.baseUrl("https://jsonplaceholder.typicode.com").build();

}

public Mono<String> getPostById(Long postId) {

return webClient.get()

.uri("/posts/{id}", postId)

.retrieve()

.bodyToMono(String.class);

}

}

**POST Request**

import org.springframework.web.reactive.function.client.WebClient;

import reactor.core.publisher.Mono;

public class WebClientExample {

public Mono<String> createPost(Post post) {

return webClient.post()

.uri("/posts")

.body(Mono.just(post), Post.class)

.retrieve()

.bodyToMono(String.class);

}

}

class Post {

private Long userId;

private String title;

private String body;

// Getters and setters

}

**PUT Request**

public Mono<String> updatePost(Long postId, Post updatedPost) {

return webClient.put()

.uri("/posts/{id}", postId)

.body(Mono.just(updatedPost), Post.class)

.retrieve()

.bodyToMono(String.class);

}

**DELETE Request**

public Mono<Void> deletePost(Long postId) {

return webClient.delete()

.uri("/posts/{id}", postId)

.retrieve()

.bodyToMono(Void.class);

}

**Usage Example**

public static void main(String[] args) {

WebClientExample example = new WebClientExample(WebClient.builder());

example.getPostById(1L).subscribe(System.out::println);

Post newPost = new Post();

newPost.setUserId(1L);

newPost.setTitle("New Post Title");

newPost.setBody("This is the body of the new post.");

example.createPost(newPost).subscribe(System.out::println);

Post updatedPost = new Post();

updatedPost.setUserId(1L);

updatedPost.setTitle("Updated Title");

updatedPost.setBody("Updated body content.");

example.updatePost(1L, updatedPost).subscribe(System.out::println);

example.deletePost(1L).subscribe();

}

Each of these methods returns a Mono, which allows for asynchronous, non-blocking execution of HTTP requests. Adjust the base URL and uri paths as needed for your API.

**Create spring boot GET PUT POST Request using WebFlux reactive api and deploy it with custom netty server**

Here’s a step-by-step guide to building a **Spring Boot WebFlux** application with GET, PUT, and POST requests and deploying it with a custom Netty server.

**Project Setup**

**1. Add Maven Dependencies**

Include the following in pom.xml:

xml

Copy code

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-webflux</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.projectreactor</groupId>

<artifactId>reactor-netty</artifactId>

</dependency>

</dependencies>

**2. Create Reactive Models and Repository**

**Model Class:**

java

Copy code

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

@Data

@AllArgsConstructor

@NoArgsConstructor

public class Employee {

private String id;

private String name;

private String department;

}

**Repository Class:**

java

Copy code

import org.springframework.stereotype.Repository;

import reactor.core.publisher.Flux;

import reactor.core.publisher.Mono;

import java.util.Map;

import java.util.concurrent.ConcurrentHashMap;

@Repository

public class EmployeeRepository {

private final Map<String, Employee> employeeData = new ConcurrentHashMap<>();

public Flux<Employee> findAll() {

return Flux.fromIterable(employeeData.values());

}

public Mono<Employee> findById(String id) {

return Mono.justOrEmpty(employeeData.get(id));

}

public Mono<Employee> save(Employee employee) {

employeeData.put(employee.getId(), employee);

return Mono.just(employee);

}

public Mono<Employee> update(String id, Employee employee) {

employeeData.put(id, employee);

return Mono.just(employee);

}

}

**3. Create Reactive REST Controller**

**Controller Class:**

java

Copy code

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.\*;

import reactor.core.publisher.Flux;

import reactor.core.publisher.Mono;

@RestController

@RequestMapping("/employees")

public class EmployeeController {

@Autowired

private EmployeeRepository employeeRepository;

@GetMapping

public Flux<Employee> getAllEmployees() {

return employeeRepository.findAll();

}

@GetMapping("/{id}")

public Mono<Employee> getEmployeeById(@PathVariable String id) {

return employeeRepository.findById(id);

}

@PostMapping

public Mono<Employee> createEmployee(@RequestBody Employee employee) {

return employeeRepository.save(employee);

}

@PutMapping("/{id}")

public Mono<Employee> updateEmployee(@PathVariable String id, @RequestBody Employee employee) {

employee.setId(id);

return employeeRepository.update(id, employee);

}

}

**4. Customize the Netty Server**

**Netty Configuration Class:**

java

Copy code

import org.springframework.boot.web.embedded.netty.NettyReactiveWebServerFactory;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import reactor.netty.http.server.HttpServer;

@Configuration

public class NettyServerConfig {

@Bean

public NettyReactiveWebServerFactory reactiveWebServerFactory() {

NettyReactiveWebServerFactory factory = new NettyReactiveWebServerFactory();

factory.addServerCustomizers(httpServer ->

httpServer.port(9090) // Custom port

.wiretap(true)); // Enable request/response logging

return factory;

}

}

**5. Deploying the Application**

**5.1. Build the Application**

Run the Maven command to package the application:

bash

Copy code

mvn clean package

**5.2. Create a Dockerfile**

dockerfile

Copy code

FROM openjdk:17-jdk-alpine

VOLUME /tmp

ARG JAR\_FILE=target/reactive-server.jar

COPY ${JAR\_FILE} app.jar

ENTRYPOINT ["java", "-jar", "/app.jar"]

Build the Docker image:

bash

Copy code

docker build -t reactive-server .

Run the Docker container:

bash

Copy code

docker run -p 9090:9090 reactive-server

**6. Test the Application**

* **GET All Employees:**

bash

Copy code

curl http://localhost:9090/employees

* **GET Employee by ID:**

bash

Copy code

curl http://localhost:9090/employees/{id}

* **POST New Employee:**

bash

Copy code

curl -X POST http://localhost:9090/employees -H "Content-Type: application/json" -d '{"id":"1","name":"John Doe","department":"Engineering"}'

* **PUT Update Employee:**

bash

Copy code

curl -X PUT http://localhost:9090/employees/1 -H "Content-Type: application/json" -d '{"id":"1","name":"Jane Doe","department":"HR"}'

**7. Deployment to Kubernetes**

**7.1 Create Kubernetes Manifests**

**deployment.yaml**:

yaml

Copy code

apiVersion: apps/v1

kind: Deployment

metadata:

name: reactive-server

spec:

replicas: 2

selector:

matchLabels:

app: reactive-server

template:

metadata:

labels:

app: reactive-server

spec:

containers:

- name: reactive-server

image: reactive-server:latest

ports:

- containerPort: 9090

**service.yaml**:

yaml

Copy code

apiVersion: v1

kind: Service

metadata:

name: reactive-server

spec:

selector:

app: reactive-server

ports:

- protocol: TCP

port: 80

targetPort: 9090

type: LoadBalancer

**7.2 Apply Configuration**

bash

Copy code

kubectl apply -f deployment.yaml

kubectl apply -f service.yaml

**Key Features**

* **Custom Netty Server**: Configured with a custom port (9090).
* **Reactive API**: Non-blocking and efficient.
* **Deployable on Docker and Kubernetes**: Scalable and easy to manage.

Let me know if you'd like further clarification or additions!

4o

**You said:**

What is difference between mono and flux if any other wrapper please explain all those also

**ChatGPT said:**

ChatGPT

In **Spring WebFlux**, **Mono** and **Flux** are reactive data types provided by **Project Reactor**, which is the foundation of reactive programming in Spring WebFlux. They represent streams of data and asynchronous processing. Here's a detailed explanation of their differences, along with other notable wrappers like Optional, CompletableFuture, and Observable:

**1. Mono**

* **Definition**: Represents a single value or no value (similar to Optional in Java).
* **When to Use**: Use Mono when you expect at most one element (0 or 1) as the result of an operation.
* **Key Methods**:
  + just(value): Wrap a single value.
  + empty(): Represent no value.
  + map(), flatMap(), filter(): Transform the data.
* **Example**:

java

Copy code

Mono<String> monoExample = Mono.just("Hello, Mono");

monoExample.subscribe(System.out::println); // Prints: Hello, Mono

* **Use Case**: Fetching a single resource like a user by ID.

**2. Flux**

* **Definition**: Represents a stream of 0 to *N* elements.
* **When to Use**: Use Flux when you expect multiple values or an unbounded stream of data.
* **Key Methods**:
  + just(values...): Wrap multiple values.
  + fromIterable(list): Wrap an iterable as a flux.
  + concat(), merge(), zip(): Combine multiple streams.
* **Example**:

java

Copy code

Flux<String> fluxExample = Flux.just("Hello", "Flux", "World");

fluxExample.subscribe(System.out::println);

// Prints:

// Hello

// Flux

// World

* **Use Case**: Fetching all employees or a stream of events.

**3. Key Differences Between Mono and Flux**

| **Feature** | **Mono** | **Flux** |
| --- | --- | --- |
| **Represents** | 0 or 1 element | 0 to N elements |
| **Operations** | Single-resource fetch | Multi-resource fetch |
| **Examples** | Single user by ID | List of users/events |

**4. Comparison with Other Wrappers**

| **Wrapper** | **Description** | **Similar To** | **Use Case** |
| --- | --- | --- | --- |
| **Mono** | Reactive, 0 or 1 element, supports asynchronous programming. | CompletableFuture, Optional | Single database fetch or API call. |
| **Flux** | Reactive, 0 to N elements, supports asynchronous programming. | Stream | Fetch multiple records or stream events. |
| **Optional** | Represents 0 or 1 element but is **blocking**. | Mono | Basic non-reactive operations. |
| **CompletableFuture** | Asynchronous computation but lacks backpressure and advanced operators provided by Mono or Flux. | Mono | Single value fetch with background computation. |
| **Observable (RxJava)** | Reactive type from RxJava that is similar to Flux. Does not enforce non-blocking IO as Mono and Flux do. | Flux | Cross-platform reactive programming. |
| **Stream** | Blocking, works on finite collections. Cannot handle asynchronous or infinite streams. | Flux | Operations on collections in memory. |

**5. Key Advantages of Mono and Flux**

* **Non-blocking**: They operate asynchronously, freeing threads for other tasks.
* **Backpressure**: Ability to handle scenarios where the producer generates data faster than the consumer can consume.
* **Rich Operators**: Support transformations (map, flatMap, etc.) and combinations (concat, merge, etc.).
* **Error Handling**: Provide powerful error-handling mechanisms (onErrorResume, onErrorReturn, etc.).

**6. Example: Mono vs. Flux in a REST API**

**Mono Example: Single Employee by ID**

java

Copy code

@GetMapping("/employee/{id}")

public Mono<Employee> getEmployeeById(@PathVariable String id) {

return employeeRepository.findById(id);

}

**Flux Example: All Employees**

java

Copy code

@GetMapping("/employees")

public Flux<Employee> getAllEmployees() {

return employeeRepository.findAll();

}

**7. Advanced Operators in Mono and Flux**

| **Operator** | **Mono** | **Flux** |
| --- | --- | --- |
| **map()** | Transform the single value. | Transform each element. |
| **flatMap()** | Asynchronous transformation. | Async transformation for each value. |
| **filter()** | Filter based on a condition. | Filter based on conditions. |
| **zip()** | Combine with another Mono. | Combine multiple Flux streams. |
| **concat()** | Not applicable (only 1 value). | Concatenate multiple Flux streams. |

Let me know if you'd like additional clarification or a specific use case example!

4o

**You said:**

let i am calling above api '/employees' using resttemplate then how will i retrieve data from response also if calling from angular service component using httpclient give code example

**ChatGPT said:**

ChatGPT

Here's how you can retrieve data from the /employees API using **RestTemplate** in **Spring Boot** and using **HttpClient** in **Angular**.

**1. Spring Boot: Calling /employees API using RestTemplate**

**Step 1: Set up a RestTemplate Bean**

First, you'll need to configure RestTemplate in your Spring Boot application as a bean.

java

Copy code

@Configuration

public class RestTemplateConfig {

@Bean

public RestTemplate restTemplate() {

return new RestTemplate();

}

}

**Step 2: Use RestTemplate to Call the API**

Now, you can use the RestTemplate in your service class to make a call to the /employees endpoint.

java

Copy code

@Service

public class EmployeeService {

@Autowired

private RestTemplate restTemplate;

private final String EMPLOYEE\_API\_URL = "http://localhost:8080/employees"; // URL of the employee API

public Flux<Employee> getAllEmployees() {

// Using WebClient to get a Flux response from REST API

return restTemplate.exchange(EMPLOYEE\_API\_URL, HttpMethod.GET, null, Flux.class).getBody();

}

}

**Step 3: Call the Service from Controller**

You can then call the service from your controller to expose the employees' data through an endpoint.

java

Copy code

@RestController

@RequestMapping("/api")

public class EmployeeController {

@Autowired

private EmployeeService employeeService;

@GetMapping("/employees")

public Flux<Employee> getAllEmployees() {

return employeeService.getAllEmployees();

}

}

**Explanation:**

* RestTemplate.exchange is used to call the REST API and retrieve the Flux<Employee> response.
* The getBody() method extracts the response body.

**2. Angular: Calling /employees API using HttpClient**

**Step 1: Set Up Angular Service to Call the API**

In Angular, you use the HttpClient to make HTTP requests. Here's an example of how to call the /employees API.

typescript

Copy code

import { Injectable } from '@angular/core';

import { HttpClient } from '@angular/common/http';

import { Observable } from 'rxjs';

import { Employee } from './employee.model'; // Make sure to create a model for Employee

@Injectable({

providedIn: 'root'

})

export class EmployeeService {

private apiUrl = 'http://localhost:8080/api/employees'; // Backend API URL

constructor(private http: HttpClient) { }

getAllEmployees(): Observable<Employee[]> {

return this.http.get<Employee[]>(this.apiUrl); // Return an observable of Employee array

}

}

**Step 2: Use the Service in Angular Component**

Now, inject the EmployeeService in your Angular component and subscribe to the observable to get the data.

typescript

Copy code

import { Component, OnInit } from '@angular/core';

import { EmployeeService } from './employee.service';

import { Employee } from './employee.model';

@Component({

selector: 'app-employee-list',

templateUrl: './employee-list.component.html',

styleUrls: ['./employee-list.component.css']

})

export class EmployeeListComponent implements OnInit {

employees: Employee[] = [];

constructor(private employeeService: EmployeeService) { }

ngOnInit(): void {

this.loadEmployees();

}

loadEmployees(): void {

this.employeeService.getAllEmployees().subscribe(

(data: Employee[]) => {

this.employees = data;

},

error => {

console.error('Error loading employees!', error);

}

);

}

}

**Step 3: Create an Employee Model (Optional)**

If you haven't already, define an Employee model for the TypeScript application:

typescript

Copy code

export interface Employee {

id: number;

name: string;

position: string;

}

**Step 4: Display Data in the Template**

You can display the employee data in your component's HTML template:

html

Copy code

<div \*ngIf="employees.length > 0">

<ul>

<li \*ngFor="let employee of employees">

{{ employee.name }} - {{ employee.position }}

</li>

</ul>

</div>

**Summary of Steps:**

* **Spring Boot**:
  + Create a RestTemplate bean.
  + Use RestTemplate.exchange to call the /employees API.
  + Return the data as a Flux<Employee>.
* **Angular**:
  + Create an Angular service using HttpClient to call the /employees API.
  + Subscribe to the Observable and display the data in the component template.

Let me know if you need more clarification!