**Reactive Programming in Spring Boot**

Spring Webflux Official Logo

**Why Reactive Spring?**

In this article, we will discuss the reactive approaches and why they are better than the traditional approaches.

Nowadays, reactive became a buzzword but it's a bit confusing. Also with microservice architecture, we start using this term so often.

So, What exactly are reactive systems?

*Reactive systems have certain characteristics that make them ideal for low-latency, high-throughput workloads. Project Reactor and the Spring portfolio work together to enable developers to build enterprise-grade reactive systems that are responsive, resilient, elastic, and message-driven.*

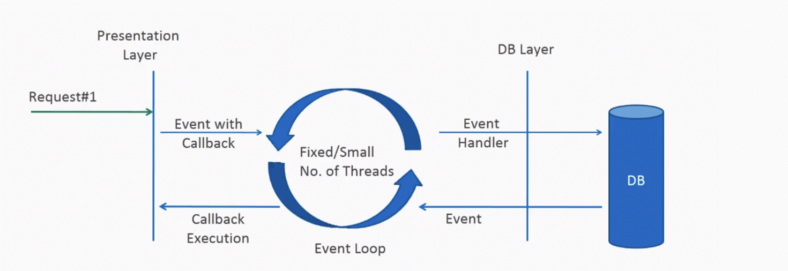
As the definition mentioned, Project React and the Spring portfolio work together and created a solution for Spring Developers. From this cooperation, Spring Web flux turned out. The traditional approach to Spring was Spring MVC which has purpose-built for the Servlet API and Servlet Containers. After adding Spring Webflux, it provides non-blocking support for Reactive Streams.

Before starting to our hands dirty, I want to give more details about what is Spring Webflux and how it is working.

**Blocking vs Non-blocking Requests**

In the traditional Spring MVC approach, when a request comes to the server, immediately a new servlet thread is creating. And this thread delegates the other small worker threads like database connection, I/O operations, etc. During this time the servlet thread is blocked.

Our server can handle a finite amount of threads, but we know that applications need flexibility. Imagine that you have an online shopping website. And you have a server to handle a finite amount of customers. But for your business, it can be some rush hours. During this time, you should increase your server abilities. At this moment, reactive programming turns out. In Non-blocking requests, there is no thread in waiting status. It works observer pattern principle. All incoming requests come with an event handler and callback information.



Non-blocking request

**What is Spring WebFlux?**

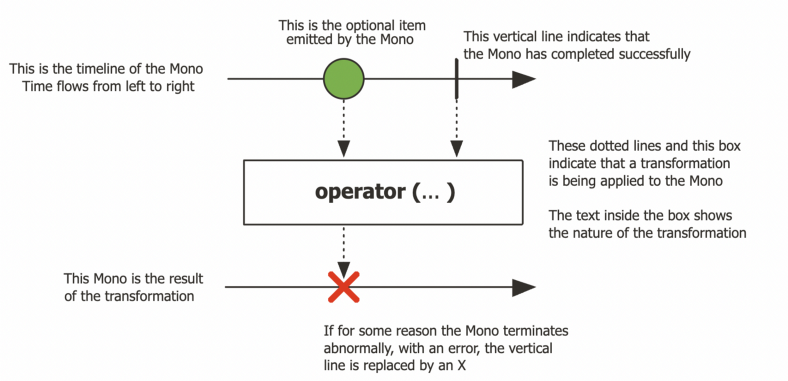
WebFlux is **a Spring reactive-stack web framework**. It was added to Spring 5. It is fully non-blocking, supports reactive streams back pressure, and runs on servers such as Netty, Undertow, and Servlet 3.1+ containers as the inbuilt server to run reactive applications. Spring WebFlux is an alternative to the traditional Spring MVC. If you are familiar with the Spring MVC programming style, you can easily work on webflux also.

Spring Webflux uses two publishers;

**Mono**: Returns 0 or 1 element.

*A Reactive Streams*[*Publisher*](https://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html?is-external=true)*with basic rx operators that emits at most one item*via*the onNext signal then terminates with an onComplete signal (successful Mono, with or without value), or only emits a single onError signal (failed Mono).*

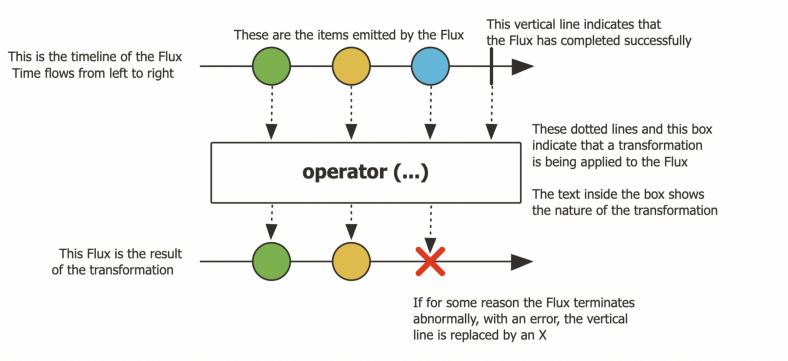
public abstract class **Mono<T>**  
extends [Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html?is-external=true)  
implements [CorePublisher](https://projectreactor.io/docs/core/release/api/reactor/core/CorePublisher.html)<T>



Mono Operator Work Flow

**Flux:**A Reactive Streams [Publisher](https://www.reactive-streams.org/reactive-streams-1.0.3-javadoc/org/reactivestreams/Publisher.html?is-external=true) with rx operators that emits 0 to N elements, and then completes (successfully or with an error).

public abstract class **Flux<T>**  
extends [Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html?is-external=true)  
implements [CorePublisher](https://projectreactor.io/docs/core/release/api/reactor/core/CorePublisher.html)<T>



In Spring WebFlux, we call reactive APIs/functions that return monos and fluxes, and your controllers will return monos and fluxes. When you invoke an API that returns a mono or a flux, it will return immediately. The function call results will be delivered to you through the mono or flux when they become available.

To build a truly non-blocking application, we must aim to create/use all of its components as non-blocking i.e. client, controller, middle services and even the database. If one of them is blocking the requests, our aim will be defeated.

**Spring Boot WebFlux Example**

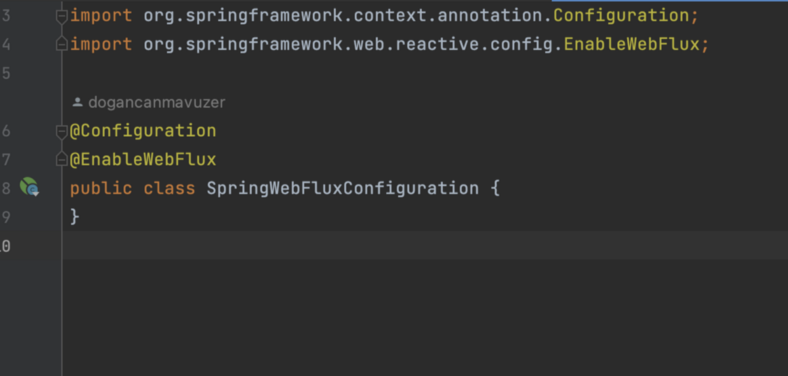
In this Spring Boot application, we will create user and role management system. To make it fully non-blocking, I am using *MongoDB* as back-end database.

**1- Gradle Dependencies**

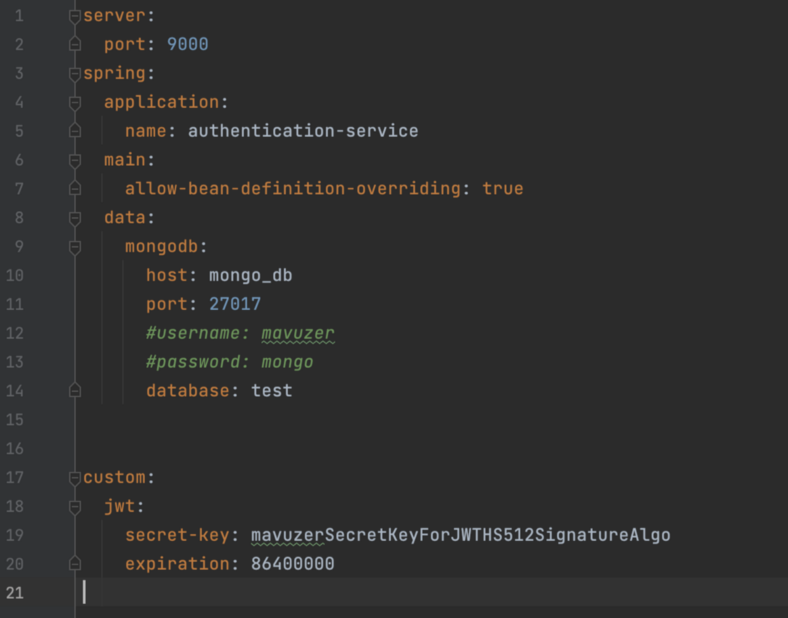


Gradle Dependencies

**2- Configurations**



Custom WebFlux Configuration



application.yml file — database connection

**Models**

User Model:

package com.mavuzer.authentication.users.model;  
  
import com.mavuzer.authentication.role.model.Role;  
import lombok.\*;  
import org.springframework.data.annotation.Id;  
import org.springframework.data.mongodb.core.index.Indexed;  
import org.springframework.data.mongodb.core.mapping.DBRef;  
import org.springframework.data.mongodb.core.mapping.Document;  
  
  
import javax.validation.constraints.Email;  
import javax.validation.constraints.NotBlank;  
import javax.validation.constraints.NotEmpty;  
import javax.validation.constraints.Size;  
import java.util.Set;  
  
@Document(collection = "users")  
@AllArgsConstructor  
@Data  
@Builder  
public class User {  
  
 @Id  
 private String id;  
  
 @NotEmpty(message = "User name is mandatory.")  
 @Size(min = 5, max = 10, message = "User name should have min 5 max 10 characters.")  
 @Indexed(unique = true)  
 private String userName;  
  
 @NotBlank(message = "Firstname is mandatory field!")  
 @Size(max = 50, message = "Firstname should be smaller than {max} characters")  
 private String firstName;  
  
 @NotBlank(message = "Lastname is mandatory field!")  
 @Size(max = 50, message = "Firstname should be smaller than {max} characters")  
 private String lastName;  
  
 @NotBlank(message = "E-mail is mandatory field!")  
 @Size(max = 50, message = "E-mail should be smaller than {max} characters")  
 @Email  
 @Indexed(unique = true)  
 private String email;  
  
 @NotBlank(message = "Password is mandatory field!")  
 @Size(max = 200, message = "Password should be smaller than {max} characters")  
 private String password;  
  
 @DBRef  
 private Set<Role> roles;  
  
  
 @Builder.Default  
 private boolean accountNonExpired = false;  
  
 @Builder.Default  
 private boolean isEnabled = true;  
  
  
}

Role Model:

@Document(collection = "role")  
@Data  
@Builder  
@NoArgsConstructor  
@AllArgsConstructor  
public class Role {  
  
 @Id  
 private String id;  
  
 @NotBlank(message = "Role name is a mandatory field!")  
 @Size(min = 1, max = 50, message = "Role name should between {min} and {max} characters")  
 private String roleName;  
  
 public Role(String roleName) {  
 this.roleName = roleName;  
 }  
}

**Repositories**

User Repository:

package com.mavuzer.authentication.users.repository;  
  
import com.mavuzer.authentication.users.model.User;  
import org.reactivestreams.Publisher;  
import org.springframework.data.mongodb.repository.ReactiveMongoRepository;  
import org.springframework.stereotype.Repository;  
import reactor.core.publisher.Mono;  
  
@Repository  
public interface UserRepository extends ReactiveMongoRepository<User, String> {  
  
 Mono<User> findById(String id);  
 Mono<User> findByUserName(String userName);  
 Mono<Boolean> existsById(String firstName);  
 Mono<Boolean> existsByEmail(String email);  
  
 Mono<Boolean> existsById(Publisher<String> id);  
}

Role Repository:

package com.mavuzer.authentication.role.repository;  
  
import com.mavuzer.authentication.role.model.Role;  
import org.springframework.data.mongodb.repository.ReactiveMongoRepository;  
import reactor.core.publisher.Mono;  
  
import java.util.Optional;  
  
public interface RoleRepository extends ReactiveMongoRepository<Role,String> {  
 Mono<Role> findRoleByRoleName(String roleName);  
}

**Services**

User Service:

package com.mavuzer.authentication.users.service;  
  
import com.mavuzer.authentication.auth.provider.PassEncoder;  
import com.mavuzer.authentication.exception.UserNotFoundException;  
import com.mavuzer.authentication.role.repository.RoleRepository;  
import com.mavuzer.authentication.users.dto.UserUpdateDto;  
import com.mavuzer.authentication.users.model.User;  
import com.mavuzer.authentication.users.repository.UserRepository;  
import lombok.RequiredArgsConstructor;  
import lombok.extern.slf4j.Slf4j;  
import org.springframework.security.crypto.password.PasswordEncoder;  
import org.springframework.stereotype.Service;  
import org.springframework.web.reactive.function.server.ServerResponse;  
import reactor.core.publisher.Flux;  
import reactor.core.publisher.Mono;  
  
@Service  
@RequiredArgsConstructor  
@Slf4j  
public class UserServiceImpl implements UserService{  
  
 @PassEncoder  
 private final PasswordEncoder passwordEncoder;  
 private final UserRepository userRepository;  
 private final RoleRepository roleRepository;  
  
 @Override  
 public Mono<User> findByUserName(String userName) {  
 *log*.debug("Finding user by {} ",userName);  
 return userRepository.findByUserName(userName);  
 }  
  
 @Override  
 public Mono<User> save(User user) {  
 *log*.debug("Recording user object to database!");  
 return userRepository.save(user);  
 }  
  
 @Override  
 public Mono<ServerResponse> deleteById(String id){  
  
 *log*.debug("Deleting the record - id : {}",id);  
  
 return userRepository.deleteById(id).then(ServerResponse.*ok*().build());  
  
 }  
  
 @Override  
 public Mono<User> update(String id, UserUpdateDto updateDto) throws UserNotFoundException {  
  
 *log*.debug("Updating user - id: {}",id);  
  
 return userRepository.findById(id).flatMap(oldUser -> {  
 oldUser.setUserName(oldUser.getUserName() != null ? oldUser.getUserName() : updateDto.getUserName());  
 oldUser.setFirstName(oldUser.getFirstName() != null ? oldUser.getFirstName() : updateDto.getFirstName());  
 oldUser.setUserName(oldUser.getLastName() != null ? oldUser.getLastName() : updateDto.getLastName());  
 oldUser.setEmail(oldUser.getEmail() != null ? oldUser.getEmail() : updateDto.getEmail());  
 oldUser.setPassword(passwordEncoder.matches(updateDto.getPassword(),oldUser.getPassword()) ? oldUser.getPassword() : passwordEncoder.encode(updateDto.getPassword()) );  
 return userRepository.save(oldUser);  
 });  
 }  
  
 @Override  
 public Mono<User> findById(String id) {  
  
 return userRepository.findById(id);  
 }  
  
 @Override  
 public Flux<User> findAll() {  
 return userRepository.findAll();  
 }  
  
}

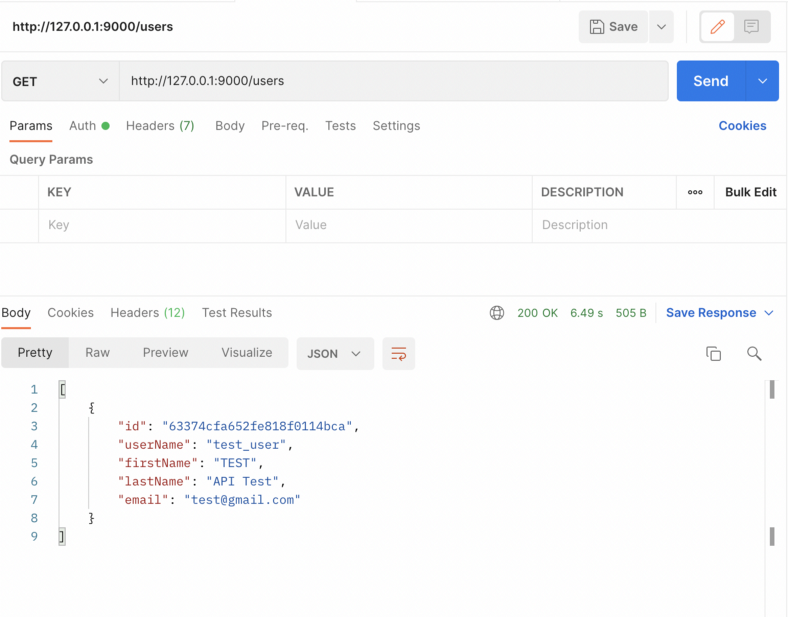
Role Service:

package com.mavuzer.authentication.role.service;  
  
import com.mavuzer.authentication.role.dto.RoleDto;  
import com.mavuzer.authentication.role.model.Role;  
import com.mavuzer.authentication.role.repository.RoleRepository;  
import lombok.RequiredArgsConstructor;  
import lombok.extern.slf4j.Slf4j;  
import org.springframework.stereotype.Service;  
import org.springframework.web.reactive.function.server.ServerResponse;  
import reactor.core.publisher.Flux;  
import reactor.core.publisher.Mono;  
  
@Service  
@RequiredArgsConstructor  
@Slf4j  
public class RoleServiceImpl implements RoleService{  
  
 private final RoleRepository roleRepository;  
 @Override  
 public Mono<Role> findRoleById(String id) {  
 return roleRepository.findById(id);  
 }  
  
 @Override  
 public Mono<Role> findRoleByName(String roleName) {  
 return roleRepository.findRoleByRoleName(roleName);  
 }  
  
 @Override  
 public Flux<Role> findAll() {  
 return roleRepository.findAll();  
 }  
  
 @Override  
 public Mono<Role> updateRole(String id, RoleDto roleDto) {  
 *log*.debug("Updating role - id: {}",id);  
  
 return roleRepository.findById(id).flatMap(oldRole -> {  
 oldRole.setRoleName(oldRole.getRoleName() != null ? oldRole.getRoleName() : roleDto.getRoleName());  
 return roleRepository.save(oldRole);  
 });  
 }  
  
 @Override  
 public Mono<ServerResponse> deleteById(String id) {  
 return roleRepository.deleteById(id).then(ServerResponse.*ok*().build());  
 }  
  
 @Override  
 public Mono<Role> save(Role role) {  
 return roleRepository.save(role);  
 }  
  
 @Override  
 public Mono<Role> update(String id, RoleDto role) {  
 return roleRepository.findById(id).flatMap(rl -> {  
 rl.setRoleName(role.getRoleName() != null ? role.getRoleName() : rl.getRoleName());  
 return roleRepository.save(rl);  
 });  
 }  
}

**Main Class**

package com.mavuzer.authentication;  
  
import com.mavuzer.authentication.users.model.User;  
import com.mavuzer.authentication.users.service.UserServiceImpl;  
import lombok.extern.slf4j.Slf4j;  
import org.springframework.boot.CommandLineRunner;  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.security.crypto.password.PasswordEncoder;  
import reactor.core.publisher.Mono;  
  
import java.util.Collections;  
import java.util.Optional;  
  
@SpringBootApplication()  
@ComponentScan(value = "com.mavuzer")  
@Slf4j  
public class AuthenticationServiceApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(AuthenticationServiceApplication.class, args);  
 }  
  
 @Bean  
 public CommandLineRunner runner(UserServiceImpl userService, PasswordEncoder passwordEncoder) {  
 return args -> {  
 User user = User.*builder*().userName("test\_user")  
 .firstName("TEST")  
 .lastName("API Test")  
 .password(passwordEncoder.encode("password"))  
 .email("test@gmail.com")  
 .roles(Collections.*EMPTY\_SET*)  
 .build();  
 Optional<User> existUser =Optional.*ofNullable*(userService.findByUserName(user.getUserName()).block());  
  
 if (existUser.isEmpty()){  
 Mono<User> userMono = userService.save(user);  
 *log*.debug("User Mono {}", userMono);  
 userMono.subscribe();  
 }  
  
 };  
 }  
  
  
}

**Postman:**



Postman API Call

**Conclusion**

Spring MVC and Spring WebFlux are both client-server architectures. But there is a key difference between them which is concurrency and blocking nature and threads of request.

With reactive model, we dont expect that our application would be work faster. The main benefit of it that it makes applications more resilient under load because they scale in a more predictable manner.

All the code used is available in the following Git Repository

<https://github.com/doganmavuzer/authentication-service>

**Bibliography**

* <https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html>
* <https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html>
* <https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html>