Introduction
to
Reactive Programming

By Sergey Kargopolov

Reactive Programming - Introduction

- Enables developers to build **non-blocking** applications that can handle asynchronous and synchronous operations

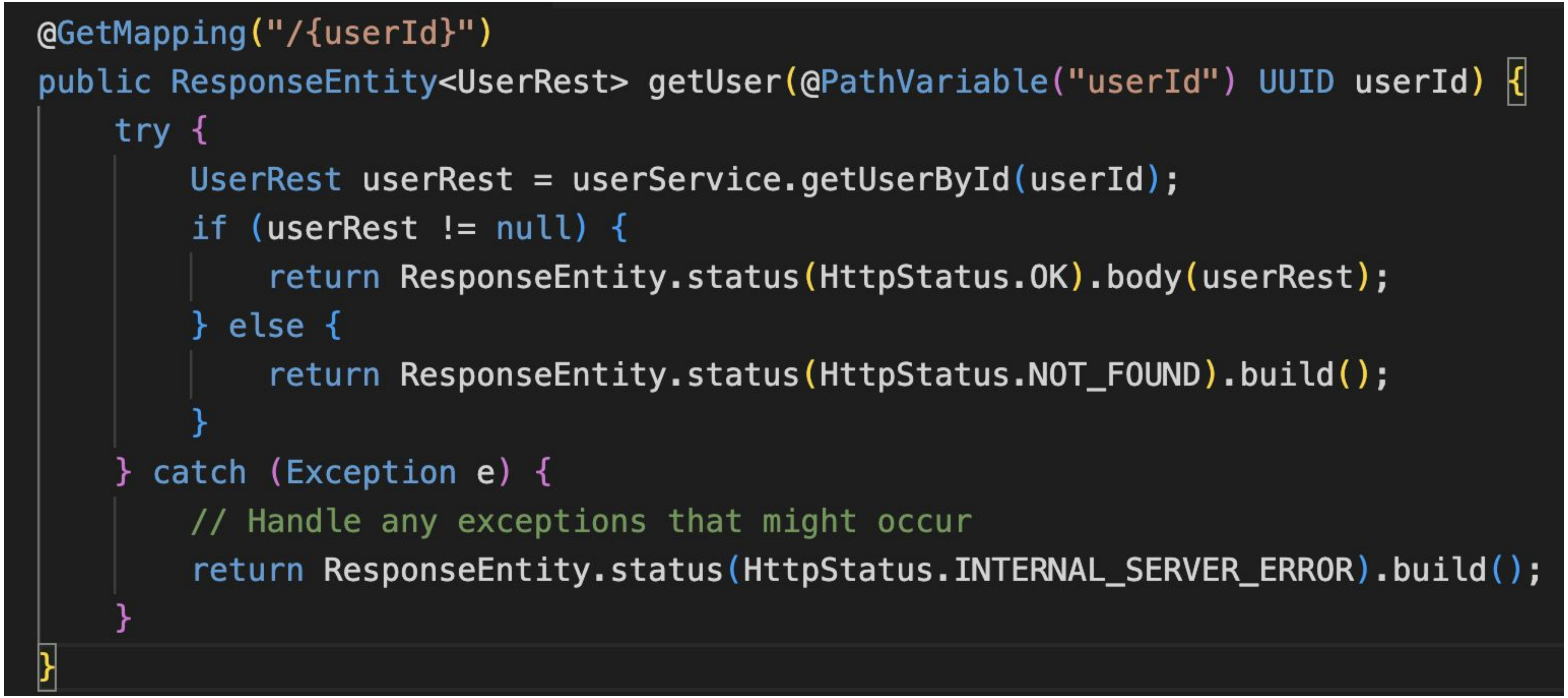
- Focuses on **data streams** and the **propagation of change**.

- Useful for applications that need to handle a **large number of concurrent users** or data streams efficiently.

- Typically employs a **functional programming style** rather than an imperative one.

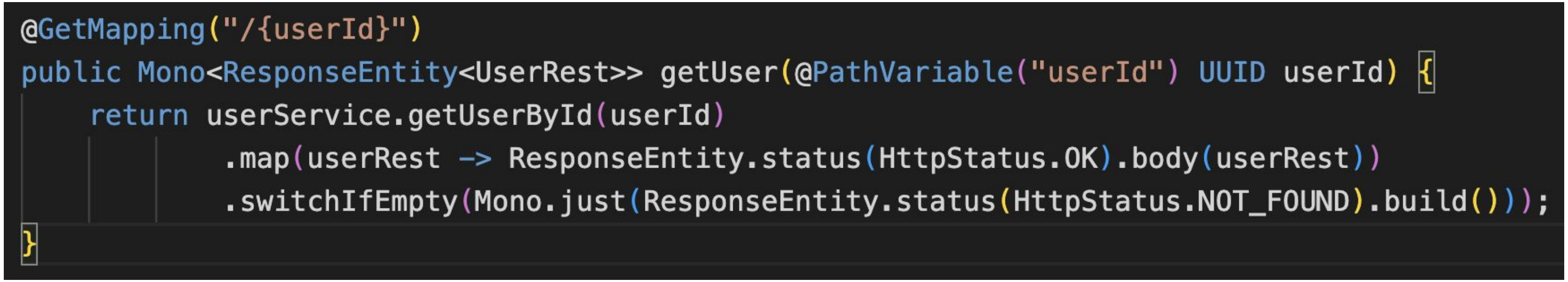
    - Reactive Streams to handle data flow
    - Lambda functions for concise code
    - Operators like map() and filter() to process data.

Imperative programming style



```java
@GetMapping("/{userId}")
public ResponseEntity<UserRest> getUser(@PathVariable("userId") UUID userId) {
    try {
        UserRest userRest = userService.getUserById(userId);
        if (userRest != null) {
            return ResponseEntity.status(HttpStatus.OK).body(userRest);
        } else {
            return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
        }
    } catch (Exception e) {
        // Handle any exceptions that might occur
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
    }
}
```

Functional programming style



```java
@GetMapping("/{userId}")
public Mono<ResponseEntity<UserRest>> getUser(@PathVariable("userId") UUID userId) {
    return userService.getUserById(userId)
            .map(userRest -> ResponseEntity.status(HttpStatus.OK).body(userRest))
            .switchIfEmpty(Mono.just(ResponseEntity.status(HttpStatus.NOT_FOUND).build()));
}
```

By Sergey Kargopolov