Reactive Programming

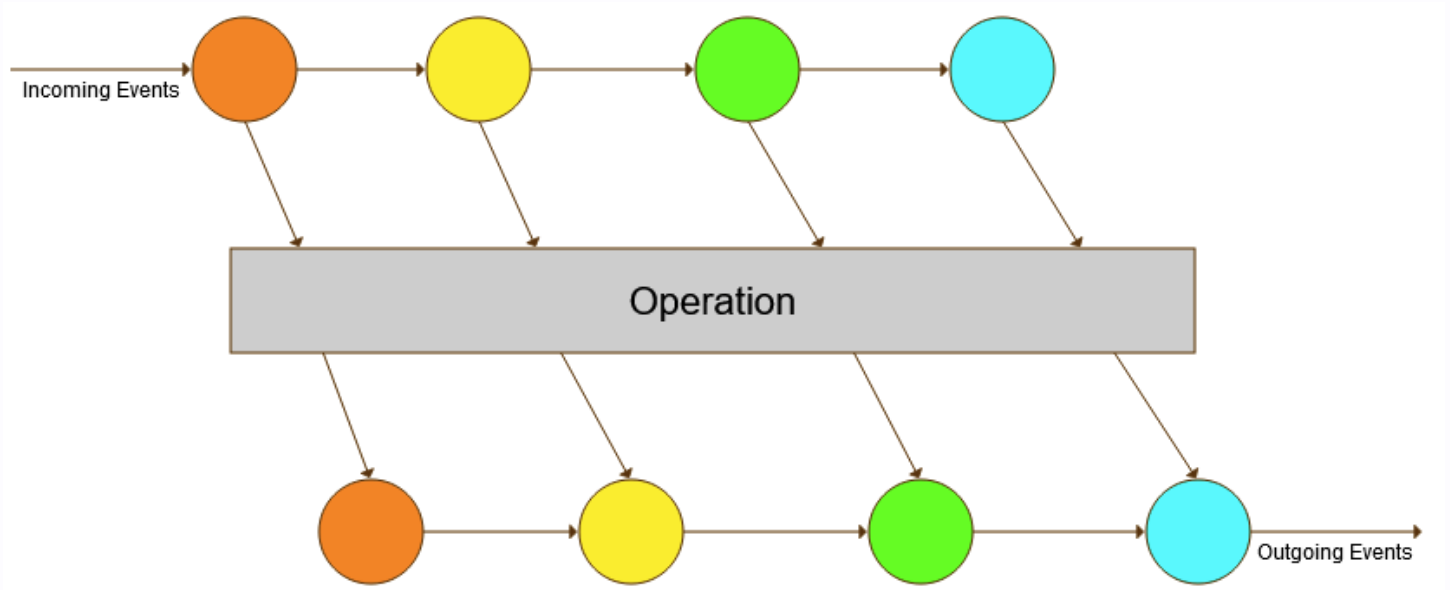
Overview of the topics covered:

- what is reactive programming and key concepts
- show Spring Webflux as an example
- migrate a Spring Web application to Webflux

What is Reactive Programming?

- paradigm that deals with asynchronous data streams and the propagation of change
- data as a flow of events
- can be observed and manipulated over time
- data flows like a stream - processed **reactively** as it arrives.

What is Reactive Programming?



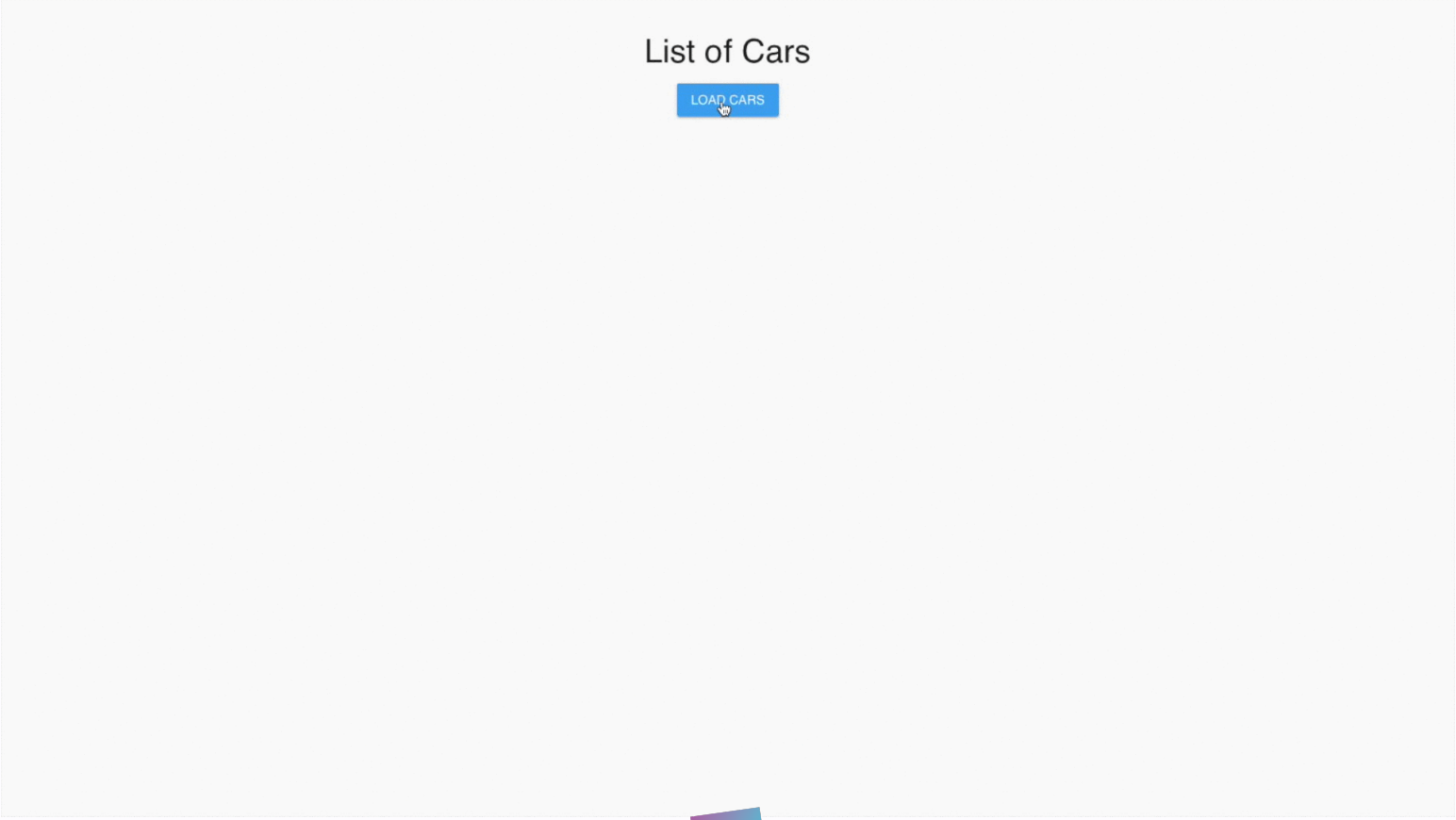
Core Principles

- **Asynchronous execution**: operations independently of the main program flow, continue executing while waiting for other tasks to finish
- **Non-blocking**: no pause while waiting for responses from I/O operations, handle more operations concurrently
- **Event-driven**: data flows are emitted in a sequence, subscribers react to new data

Why Reactive Programming?

- helps with handling concurrency and large-scale data without blocking threads
- improves performance in I/O-bound applications
- useful in microservices architectures where services need to be non-blocking and scalable

Why Reactive Programming?



Key Concepts in Reactive Programming

Mono and Flux

- Mono
  - a single asynchronous value (like Optional or Future)
  - no value at all (empty)
- Flux
  - sequence asynchronous values, like a stream

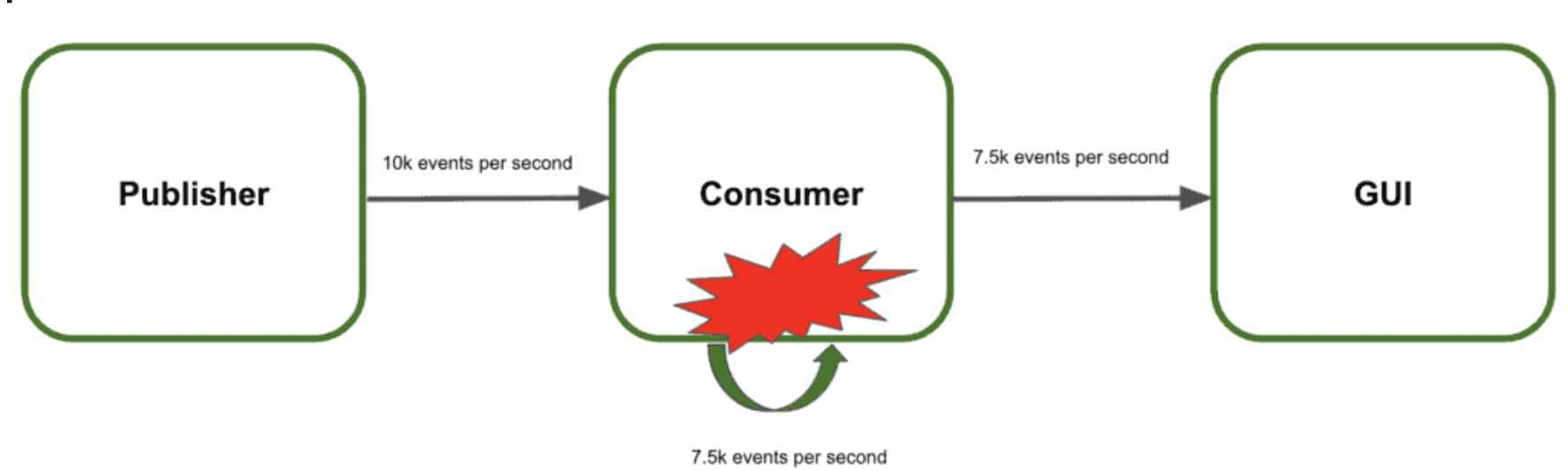
```
Mono<String> mono = Mono.just("Hello, Reactive World!");
Flux<String> flux = Flux.just("Item 1", "Item 2", "Item 3");
```

Processing

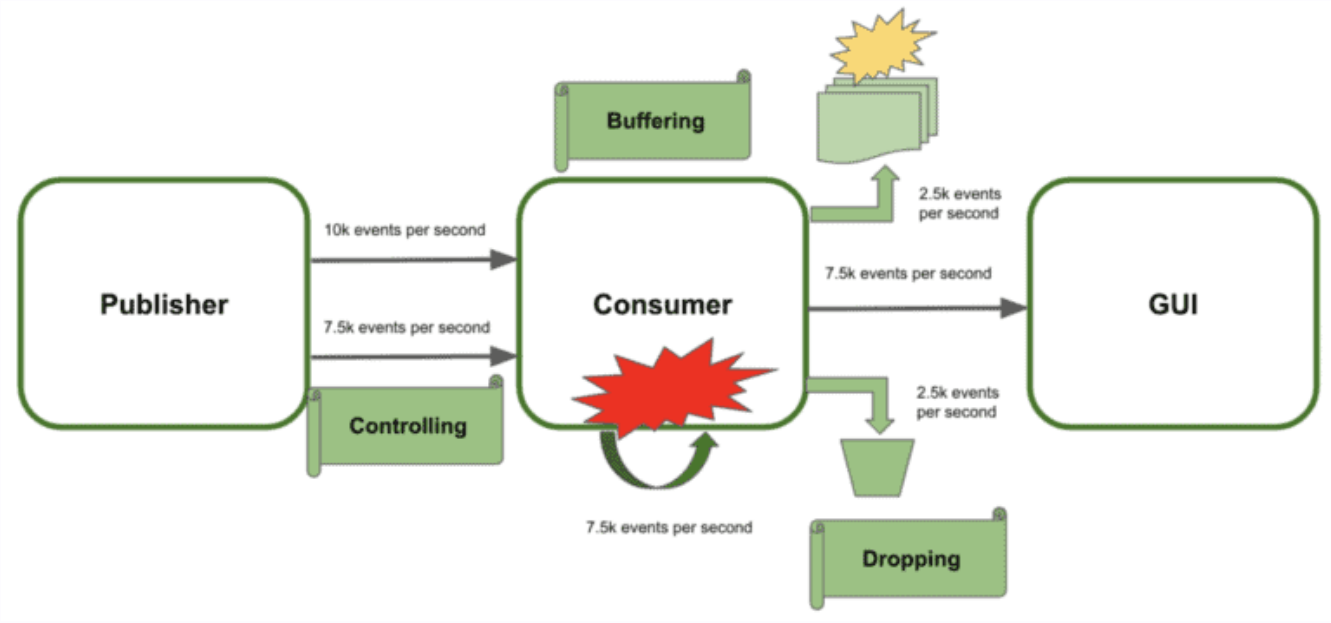
- managing data streams in a declarative way
- key operations
  - filter
  - map
  - reduce
  - concat
  - ...
- operations chainable

Backpressure

- consumer is overwhelmed by the rate of data being produced



Backpressure Solutions



Disadvantages

- complexity
  - new paradigms to learn
  - concurrent code requires more mental effort
- incompatibilities may introduce blocking
  - some databases do not have native reactive drivers
  - non-reactive legacy services
- not always suitable
  - problems may be solved with other technology
  - simple CRUD app or no real-time data necessary

Making a decision

- reactive programming not a one-size-fits-all solution
- depends on your
  - specific application needs
  - traffic patterns
  - team skillset

Go Reactive if

- I/O-bound application, needs to handle many concurrent requests
- low-latency, high-throughput, and efficient resource usage needed
- requires real-time data processing, event-driven behavior, or handles streaming data
- scalability and fault tolerance under heavy load are key priorities
- team is familiar with reactive principles and libraries

Resources

- https://www.reactivemanifesto.org/

Reactive Programming in Java

- reactive programming libraries
  - Project Reactor 
  - RxJava 
- Spring WebFlux is built on Project Reactor

Spring Webflux

- reactive web framework in Spring
- uses a non-blocking thread model
- supports asynchronous and non-blocking communication
- handle many more requests with fewer threads

Reactive REST API with Spring WebFlux

- known methods like `@RestController`, `@GetMapping` may be used
- return values should be `Flux` or `Mono`

```java
@RestController
public class ReactiveController {

    @GetMapping("/mono")
    public Mono<String> getMono() {
        return Mono.just("Hello, Reactive World!");
    }
}
```

Error Handling

- errors as signals
  - part of the stream
  - terminating the stream unless explicitly handled
- exceptions are wrapped in Mono or Flux
- not thrown

```
Flux.error(new RuntimeException("Something went wrong!"))
```

Error Handling

- different ways to react on errors
  - **onErrorResume**: switch to alternative stream
  - **onErrorReturn**: provide default fallback value
  - **onErrorMap**: transform error into different exception
  - **doOnError**: perform side effect (logging, metrics, ...)
  - many more

Error Handling - Example

- return a default value on error

```
Mono.error(new RuntimeException("Something went wrong!"))
                .onErrorReturn("Default");
```

Spring WebFlux WebClient

- non-blocking, reactive HTTP client
- asynchronous HTTP requests
- consuming third-party APIs in a reactive Spring application
- fluent, builder-based API for constructing HTTP requests

Spring WebFlux WebClient - Example

```
WebClient webClient = WebClient.create("http://localhost:8080");

Flux<String> response = webClient.get()
        .uri("/api/items")
        .retrieve()
        .bodyToFlux(String.class);
```

Spring Data R2DBC

- Reactive Relational Database Connectivity
- perform reactive database operations
- non-blocking databse calls
- reactive drivers that return Mono and Flux
- Spring Data provides a repository layer
  - use familiar patterns like `@Repository`

Spring Data R2DBC - Example

```
public class Book {
  @Id
  private int id;
  private String title;
  private String author;
  ...
}
```

```
@Repository
public interface BookRepository extends R2dbcRepository<Book, Integer>{
  Flux<Book> findByTitleContaining(String title);
}
```

Resources

- https://docs.spring.io/spring-framework/reference/web/webflux.html
- https://spring.io/projects/spring-data-r2dbc