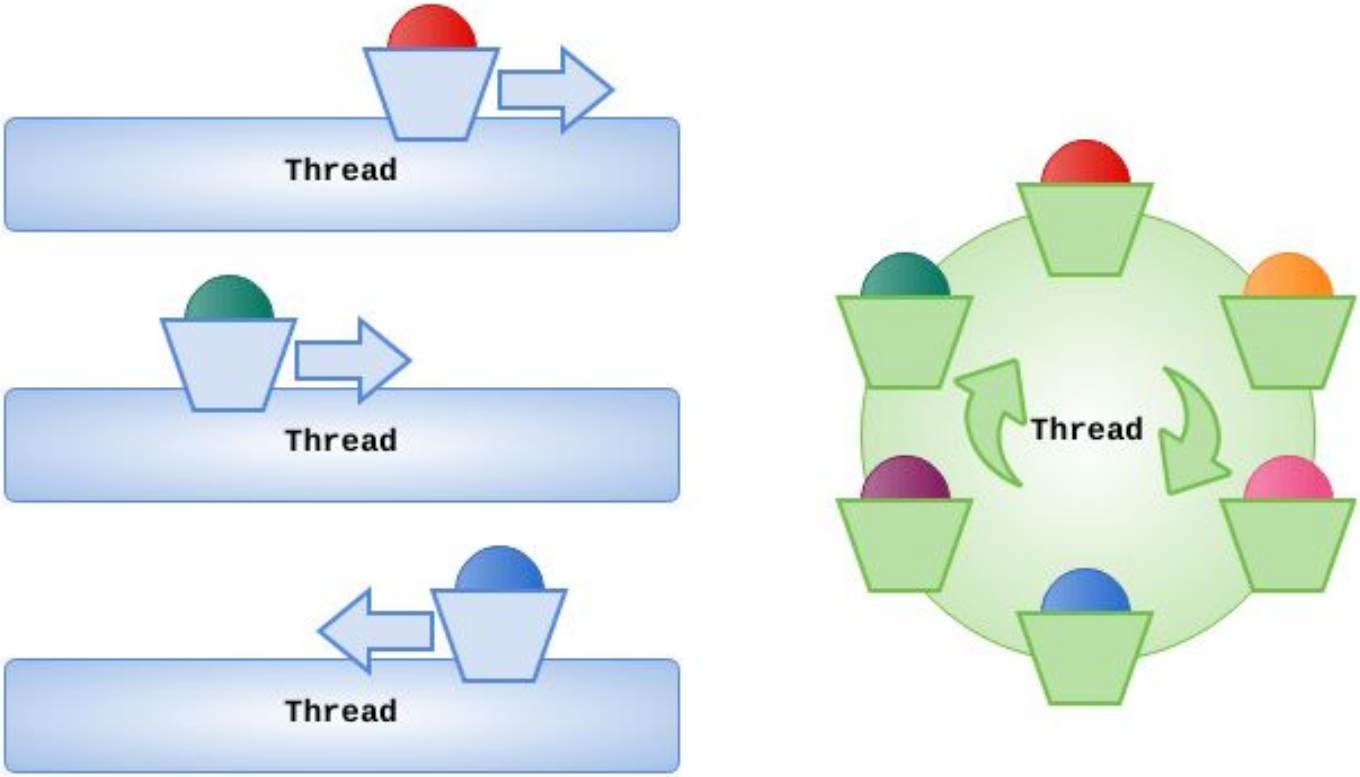
Reactive Systems

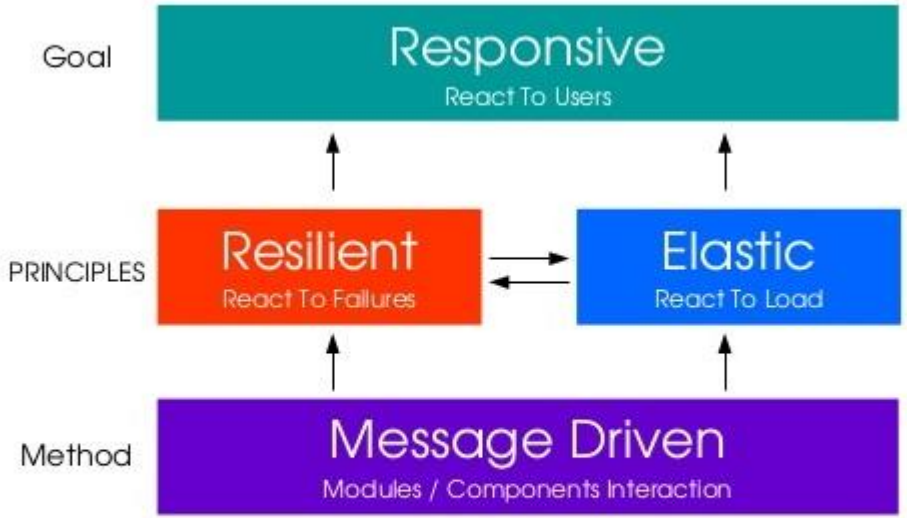
Reactive vs Blocking



When reactive is beneficial?

The key expected benefit of reactive and non-blocking is the ability to *scale with a small, fixed number of threads and less memory*. That makes applications more resilient under load, because they scale in a more predictable way. In order to observe those benefits, however, you need to have some *latency* (including a mix of slow and unpredictable network I/O). That is where the reactive stack begins to show its strengths, and the differences can be dramatic.

Reactive Manifesto



Reactive Manifesto

- Reactive systems are **responsive**, meaning that they respond in a timely manner, in all possible circumstances. They focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service.

- Reactive systems are **resilient**, meaning that they remain responsive in the face of failure. Resilience is achieved by the techniques of replication, containment, isolation, and delegation. By isolating application components from each other, you can contain failures and protect the system as a whole.

- Reactive systems are **elastic**, meaning that they stay responsive under varying workloads. This is achieved by scaling application components elastically to meet the current demand.

- Reactive systems are **message-driven**, meaning that they rely on asynchronous *message* passing between components. This allows you to create loose coupling, isolation, and location transparency.

Reactive technology characteristics

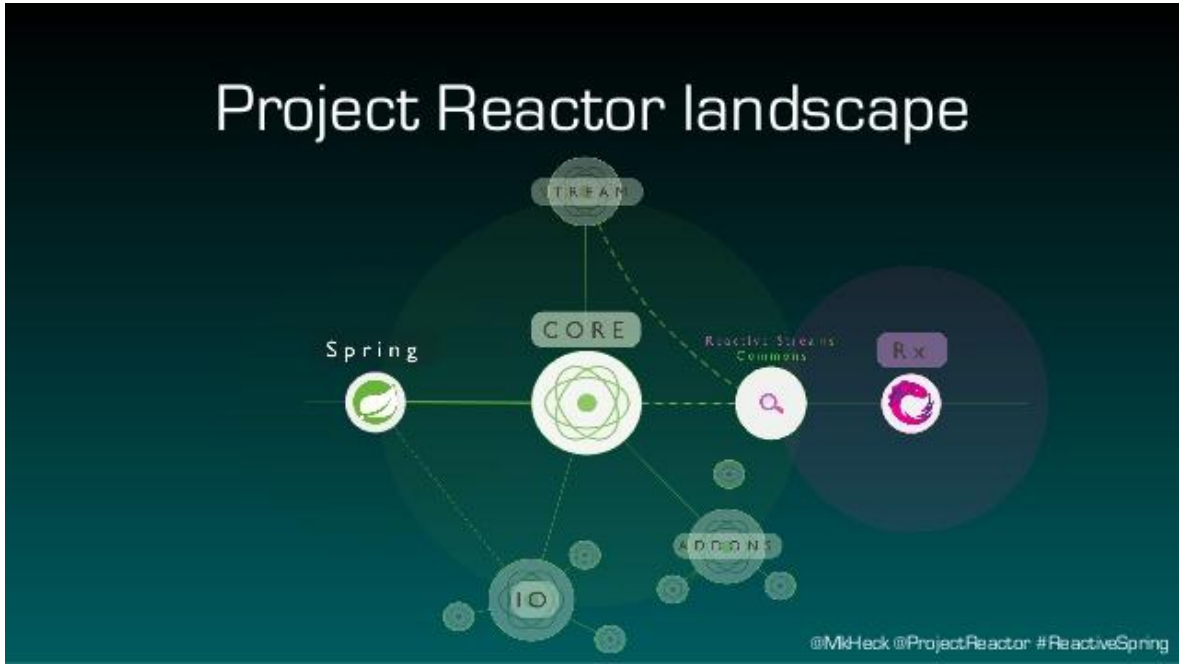
- **Data streams**: A *stream* is a sequence of events ordered in time, such as user interactions, REST service calls, JMS messages, and results from a database.
- **Asynchronous**: Data stream events are captured asynchronously and your code defines what to do when an event is emitted, when an error occurs, and when the stream of events has completed.
- **Non-blocking**: As you process events, your code should not block and perform synchronous calls; instead, it should make asynchronous calls and respond as the results of those calls are returned.
- **Back pressure**: Components control the number of events and how often they are emitted. In reactive terms, your component is referred to as the *subscriber* and events are emitted by a *publisher*. This is important because the *subscriber is in control of how much data it receives* and thus will not overburden itself.
- **Failure messages**: Instead of components throwing exceptions, failures are sent as messages to a handler function. Whereas throwing exceptions breaks the stream, defining a function to handle failures as they occur does not.

Reactive Streams initiative

```
package org.reactivestreams;

public interface Publisher<T> {
    void subscribe(Subscriber<? super T> s);
}

package org.reactivestreams;

public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```

```
package org.reactivestreams;

public interface Subscription {
    public void request(long n);
    public void cancel();
}
```

```
package org.reactivestreams;

public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {
}
```



Project Reactor & Spring WebFlux

Main Project Reactor publishers

- **Mono**: Returns 0 or 1 element.
- **Flux**: Returns 0 or more elements. A Flux can be endless, meaning that it can keep emitting elements forever, or it can return a sequence of elements and then send a completion notification when it has returned all of its elements.

Monos and fluxes are conceptually similar to futures, but more powerful. When you invoke a function that returns a mono or a flux, it will return immediately. The results of the function call will be delivered to you through the mono or flux when they become available.

In ***Spring WebFlux***, you will call reactive libraries that return monos and fluxes and your controllers will return monos and fluxes. Because these return immediately, your controllers will effectively give up their threads and allow Reactor to handle responses asynchronously. It is important to note that on*ly by using reactive libraries can your WebFlux services stay reactive*. If you use non-reactive libraries, such as JDBC calls, your code will block and wait for those calls to complete before returning.

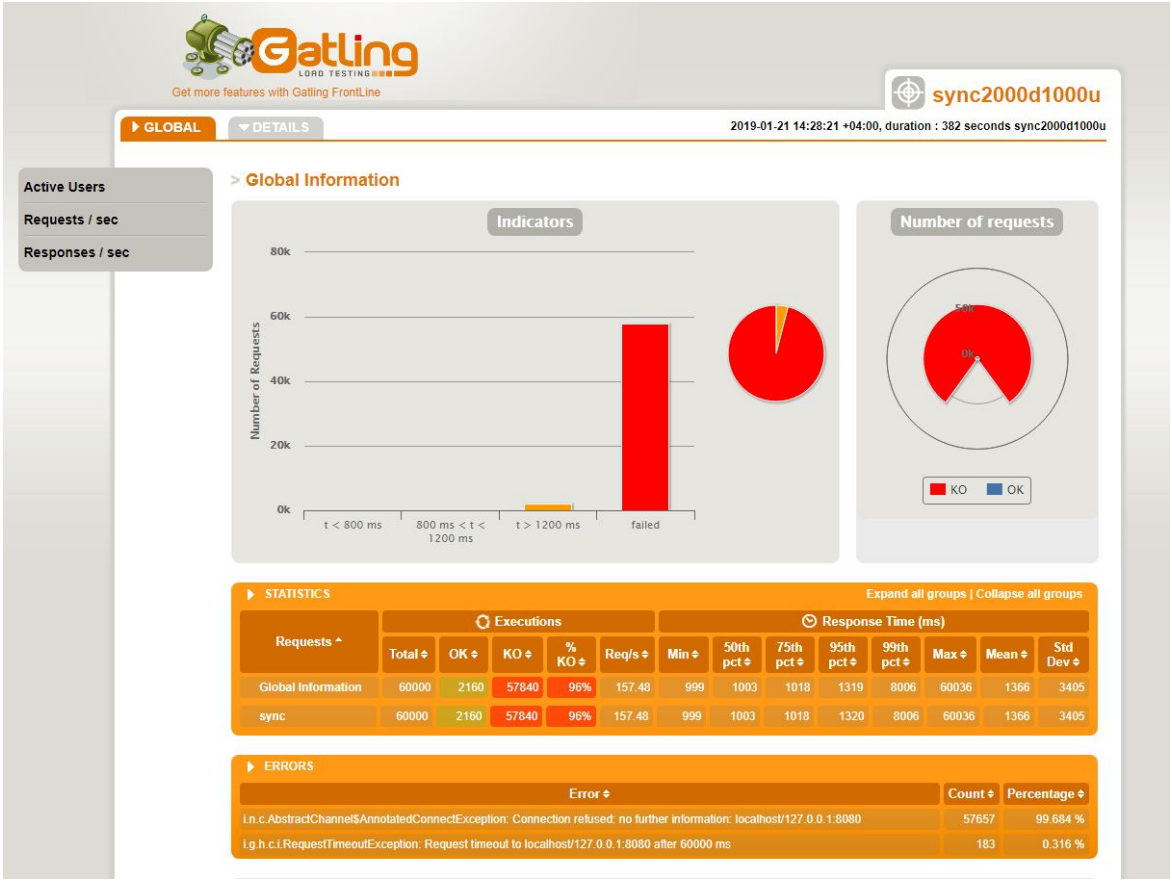
Время кодить!

ReactiveMongoRepository

- Mono<Book> save()
- Flux<Book> saveAll()
- Flux<Book> findById()
- Mono<Boolean> existsById()
- Flux<Book> findAll()
- Flux<Book> findAllById()
- Mono<Long> count()
- Mono<Void> delete()
- Mono<Void> deleteById()
- Mono<Void> deleteAll()
- Flux<Book> insert()

Pure statistics

Synchronous calls 2000d 1000u



Asynchronous calls 2000d 1000u

Further reading

Spring Web Reactive:

https://docs.spring.io/spring/docs/current/spring-framework-reference/web-reactive.html

https://docs.spring.io/spring/docs/5.0.0.M4/spring-framework-reference/html/web-reactive.html

Detailed Spring WebFlux example:

https://developer.okta.com/blog/2018/09/24/reactive-apis-with-spring-webflux

Spring Security 5 for Reactive applications:

https://www.baeldung.com/spring-security-5-reactive