**Reactive web programming**

Reactive web programming is a non-blocking application which manages asynchronous data between the producer and consumer.

Non-blocking/asynchronous application-

Means both the producer and consumer should not have to wait for processing the data. Non-blocking applications are normally implemented with message passing (or events).

Reactive web programming uses the reactor pattern.

Frameworks/tools that are used to implement non-blocking (web) applications are Akka (for Scala and Java) and Node.js

What is a Non-Blocking framework?

<https://softwareengineering.stackexchange.com/questions/144878/understanding-blocking-and-non-blocking-frameworks>

Imagine you want to make a deposit to your bank account. You walk in and notice there is no queue of people waiting line.

The sign over the bank teller says: "Non-blocking teller". You walk up and ask the teller to process your deposit, the teller responds: "I'm busy with another transaction, try again later".

You wait some amount of time, and try again later. You make your own decision if you want to keep trying to get your transaction processed, or not. You try and the 5th time you try, your transaction is processed immediately.

The "Blocking teller" would have told you to "stand in line as a FIFO queue". The non-blocking teller says: "try again later". Which bank teller would you prefer to interact with, and why?

Definition: A non-blocking framework provides a service and returns a result immediately instead of expecting the other programs requesting a resource to wait.

In other words: When a client side program makes a call to the framework, the call will always return immediately with whatever response it has for you without expecting you to sit there waiting for something. This guarantee is nice for programmers who don't want to worry about programming around the situation where the framework expects us to wait for 5 minutes.

A concrete programming example: Suppose your framework wants to provide mutually exclusive access to a file saved on the server side. An example of a "non-blocking call" would be try\_lock. If the client side wants access to the file, the framework responds: "No, try again later", rather than putting you in a queue and expecting you to sit there waiting.

The client side keeps trying for the lock, until it gets it, once it gets it, it does its business and unlocks it. The benefit of this is that whatever you try has an immediate effect.

Drawbacks to non-blocking frameworks: When there is too much work to be done for a non-blocking framework, clients are denied, and fairness is not enforced, only the clients who badger the server the most get access to the service. It's not fair.

Event- Events are messages that are sent from one object to another. The component sending the event (aka firing the event) is the producer, the component receiving the event (aka handling the event) is the consumer.

Reactor pattern- The reactor design pattern is an event handling pattern for handling service requests delivered concurrently to a service handler by one or more inputs. The service handler then demultiplexes the incoming requests and dispatches them synchronously to the associated request handlers

For Example-one thread serves all requests by multiplexing between tasks and never blocks anywhere – whenever something is ready, it gets processed by that thread (or a couple of threads). So, if two requests are made to a web app that reads from the database and writes the response, the framework reads the input from each socket (by getting notified on incoming data, switching between the two sockets), and when it has read everything, passes a “here’s the request” message to the application code. The application code then sends a message to a database access layer, which in turn sends a message to the database (driver), and gets notified whenever reading the data from the database is complete. In the callback it in turn sends a message to the frontend/controller, which in turn writes the data as response, by sending it as message(s). Everything consists of a lot of message passing and possibly callbacks.

Why pattern is required?

<https://dzone.com/articles/understanding-reactor-pattern-thread-based-and-eve>

To handle web requests, there are two competing web architectures: thread-based and event-driven architectures.

Thread-Based Architecture

The most intuitive way to implement a multi-threaded server is to follow the thread-per-connection approach. It is appropriate for sites that need to avoid threading for compatibility with non-thread-safe libraries.

It also uses the best multi-processing modules for isolating each request, so that a problem with a single request will not affect any other.

Processes are too heavyweight, with slower context switching and higher memory consumption. Therefore, the thread-per-connection approach is used for better scalability, though programming with threads is error-prone and hard to debug.

In order to tune the number of threads for the best overall performance and avoid thread-creating/destroying overhead, it is a common practice to put a single dispatcher thread in front of a bounded blocking queue and a thread pool. The dispatcher blocks on the socket for new connections and offers them to the bounded blocking queue. Connections exceeding the limitation of the queue will be dropped, but latencies for accepted connections become predictable. A pool of threads polls the queue for incoming requests, which will then be processed and responded to.

Unfortunately, there is always a one-to-one relationship between connections and threads. Long-living connections like Keep-Alive connections give rise to a large number of worker threads waiting in an idle state, e.g. file system access, network, etc. In addition, hundreds or even thousands of concurrent threads can waste a great deal of stack space in memory.

Event-Driven Architecture

Event-driven approach can separate threads from connections, which only use threads for events on specific callbacks or handlers.

An event-driven architecture consists of event creators (producer) and event consumers. The creator, which is the source of the event, only knows that the event has occurred. Consumers are entities that need to know the event has occurred. They may be involved in processing the event or they may simply be affected by the event.

The Reactor Pattern

The reactor pattern is one implementation technique of event-driven architecture. In simple terms, it uses a single threaded event loop blocking on resource-emitting events and dispatches them to corresponding handlers and callbacks.

There is no need to block on I/O, as long as handlers and callbacks for events are registered to take care of them. Events refer to instances like a new incoming connection, ready for read, ready for write, etc. Those handlers/callbacks may utilize a thread pool in multi-core environments.

This pattern decouples the modular application-level code from reusable reactor implementation.

There are two important participants in the architecture of Reactor Pattern:

1. Reactor

A Reactor runs in a separate thread, and its job is to react to IO events by dispatching the work to the appropriate handler. It’s like a telephone operator in a company who answers calls from clients and transfers the line to the appropriate contact.

2. Handlers

A Handler performs the actual work to be done with an I/O event, similar to the actual officer in the company the client wants to speak to.

A reactor responds to I/O events by dispatching the appropriate handler. Handlers perform non-blocking actions.

The Intent of the Reactor Pattern

The Reactor architectural pattern allows event-driven applications to demultiplex and dispatch service requests that are delivered to an application from one or more clients.

One reactor will keep looking for events and will inform the corresponding event handler to handle it once the event gets triggered.

The Reactor Pattern is a design pattern for synchronous demultiplexing and order of events as they arrive.

It receives messages, requests, and connections coming from multiple concurrent clients and processes these posts sequentially using event handlers. The purpose of the Reactor design pattern is to avoid the common problem of creating a thread for each message, request, and connection. Then it receives events from a set of handlers and distributes them sequentially to the corresponding event handlers.

[Avoid this problem is to avoid the famous problem: C10K.](http://www.kegel.com/c10k.html)

In Summary: Servers have to handle more than 10,000 concurrent clients, and threads cannot scale the connections using Tomcat, Glassfish, JBoss, or HttpClient.

So, the application using the reactor only needs to use a thread to handle simultaneous events.

Basically, the standard Reactor allows a lead application with simultaneous events, while maintaining the simplicity of single threading.

A demultiplexer is a circuit that has an input and more than one output. It is a circuit used when you want to send a signal to one of several devices.

This description sounds similar to the description given to a decoder, but is used to select between many devices, while a demultiplexer is used to send a signal among many devices.

A Reactor allows multiple tasks which block to be processed efficiently using a single thread. The Reactor also manages a set of event handlers. When called to perform a task, it connects with the handler that is available and makes it as active.

The Cycle of Events:

Find all handlers that are active and unlocked or delegates this for a dispatcher implementation.

Execute each of these handlers sequentially until complete, or a point is reached where they are blocked. Completed Handlers are deactivated, allowing the event cycle to continue.

Repeats from Step One (1)

Why does it Matter?

Because the Reactor pattern is used by Node.js, Vert.x, Reactive Extensions, Jetty, Ngnix, and others. So if you like the identify pattern and want to know how things work behind the scenes, it is important to pay attention to this pattern.

2.[The observer pattern](http://www.vogella.com/tutorials/DesignPatternObserver/article.html#observer)

[1.1. Definition](http://www.vogella.com/tutorials/DesignPatternObserver/article.html#observer_definition)

The observer pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

The object which is being watched is called the subject. The objects which are watching the state changes are called observers or listeners

[1.2. Example](http://www.vogella.com/tutorials/DesignPatternObserver/article.html#observer_example)

The observer pattern is very common in Java. For example, you can define a listener for a button in a user interface. If the button is selected, the listener is notified and performs a certain action.

But the observer pattern is not limited to single user interface components. For example, you could have a part A in your application which displays the current temperature.

Another part B displays a green light if the temperature is above 20 degree celsius. To react to changes in the temperature, part B registers itself as a listener to Part A.

If the temperature in part A is changed, an event is triggered. This event is sent to all registered listeners, as, for example, part B. Part B receives the changed data and can adjust its display.

The following example code shows such a listener implementation for a button.

Button button = new Button(shell, SWT.PUSH);

//register listener for the selection event

button.addSelectionListener(new SelectionAdapter() {

@Override

public void widgetSelected(SelectionEvent e) {

System.out.println("Called!");

}

});

[1.3. Code example](http://www.vogella.com/tutorials/DesignPatternObserver/article.html#observer_code)

In the following example the observer is watching changes in a List of People objects. For this example create a new Java project called com.vogella.java.designpattern.observer and the following classes.

package com.vogella.java.designpattern.observer;

import java.beans.PropertyChangeEvent;

import java.beans.PropertyChangeListener;

import java.util.ArrayList;

import java.util.List;

public class MyModel {

public static final String FIRSTNAME = "firstName";

public static final String LASTNAME = "lastName";

private List<Person> persons = new ArrayList<Person>();

private List<PropertyChangeListener> listener = new ArrayList<PropertyChangeListener>();

public class Person {

private String firstName;

private String lastName;

public Person(String firstName, String lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

public String getFirstName() {

return firstName;

}

public void setFirstName(String firstName) {

notifyListeners(

this,

FIRSTNAME,

this.firstName,

this.firstName = firstName);

}

public String getLastName() {

return lastName;

}

public void setLastName(String lastName) {

notifyListeners(

this,

LASTNAME,

this.lastName,

this.lastName = lastName);

}

}

public List<Person> getPersons() {

return persons;

}

public MyModel() {

// just for testing we hard-code the persons here:

persons.add(new Person("Lars", "Vogel"));

persons.add(new Person("Jim", "Knopf"));

}

private void notifyListeners(Object object, String property, String oldValue, String newValue) {

for (PropertyChangeListener name : listener) {

name.propertyChange(new PropertyChangeEvent(this, property, oldValue, newValue));

}

}

public void addChangeListener(PropertyChangeListener newListener) {

listener.add(newListener);

}

}

package com.vogella.java.designpattern.observer;

import java.beans.PropertyChangeEvent;

import java.beans.PropertyChangeListener;

public class MyObserver implements PropertyChangeListener {

public MyObserver(MyModel model) {

model.addChangeListener(this);

}

@Override

public void propertyChange(PropertyChangeEvent event) {

System.out.println("Changed property: " + event.getPropertyName() + " [old -> "

+ event.getOldValue() + "] | [new -> " + event.getNewValue() +"]");

}

}

package com.vogella.java.designpattern.observer;

import com.vogella.java.designpattern.observer.MyModel.Person;

public class Main {

public static void main(String[] args) {

MyModel model = new MyModel();

MyObserver observer = new MyObserver(model);

// we change the last name of the person, observer will get notified

for (Person person : model.getPersons()) {

person.setLastName(person.getLastName() + "1");

}

// we change the name of the person, observer will get notified

for (Person person : model.getPersons()) {

person.setFirstName(person.getFirstName() + "1");

}

}

}

[1.4. Evaluation](http://www.vogella.com/tutorials/DesignPatternObserver/article.html#observer_s1s2d)

The observer pattern allows for the Open Closed principle. This principle states that a class should be open for extensions without the need to change the class.

Using the observer pattern a subject can register an unlimited number of observers. If a new listener wants to register with the subject, no code change in the subject is necessary.

Using the listener pattern decouples the subject from its observers. Only the observers have direct knowledge about the subject.

Spring documentation

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

23.1.1 What is Reactive Programming?

In plain terms reactive programming is about non-blocking applications that are asynchronous and event-driven and require a small number of threads to scale vertically (i.e. within the JVM) rather than horizontally (i.e. through clustering).

A key aspect of reactive applications is the concept of backpressure which is a mechanism to ensure producers don’t overwhelm consumers. For example in a pipeline of reactive components extending from the database to the HTTP response when the HTTP connection is too slow the data repository can also slow down or stop completely until network capacity frees up.

Reactive programming also leads to a major shift from imperative to declarative async composition of logic. It is comparable to writing blocking code vs using theCompletableFuture from Java 8 to compose follow-up actions via lambda expressions.

For a longer introduction check the blog series ["Notes on Reactive Programming"](https://spring.io/blog/2016/06/07/notes-on-reactive-programming-part-i-the-reactive-landscape) by Dave Syer.

23.1.2 Reactive API and Building Blocks

Spring Framework 5 embraces [Reactive Streams](https://github.com/reactive-streams/reactive-streams-jvm#reactive-streams) as the contract for communicating backpressure across async components and libraries. Reactive Streams is a specification created through industry collaboration that has also been adopted in Java 9 as java.util.concurrent.Flow.

The Spring Framework uses [Reactor](https://projectreactor.io/) internally for its own reactive support. Reactor is a Reactive Streams implementation that further extends the basic Reactive Streams Publisher contract with the Flux and Mono composable API types to provide declarative operations on data sequences of 0..N and 0..1.

The Spring Framework exposes Flux and Mono in many of its own reactive APIs. At the application level however, as always, Spring provides choice and fully supports the use of RxJava. For more on reactive types check the post ["Understanding Reactive Types"](https://spring.io/blog/2016/04/19/understanding-reactive-types) by Sebastien Deleuze.

23.2 Spring Web Reactive Module

Spring Framework 5 includes a new spring-web-reactive module. The module contains support for reactive HTTP and WebSocket clients as well as for reactive server web applications including REST, HTML browser, and WebSocket style interactions.

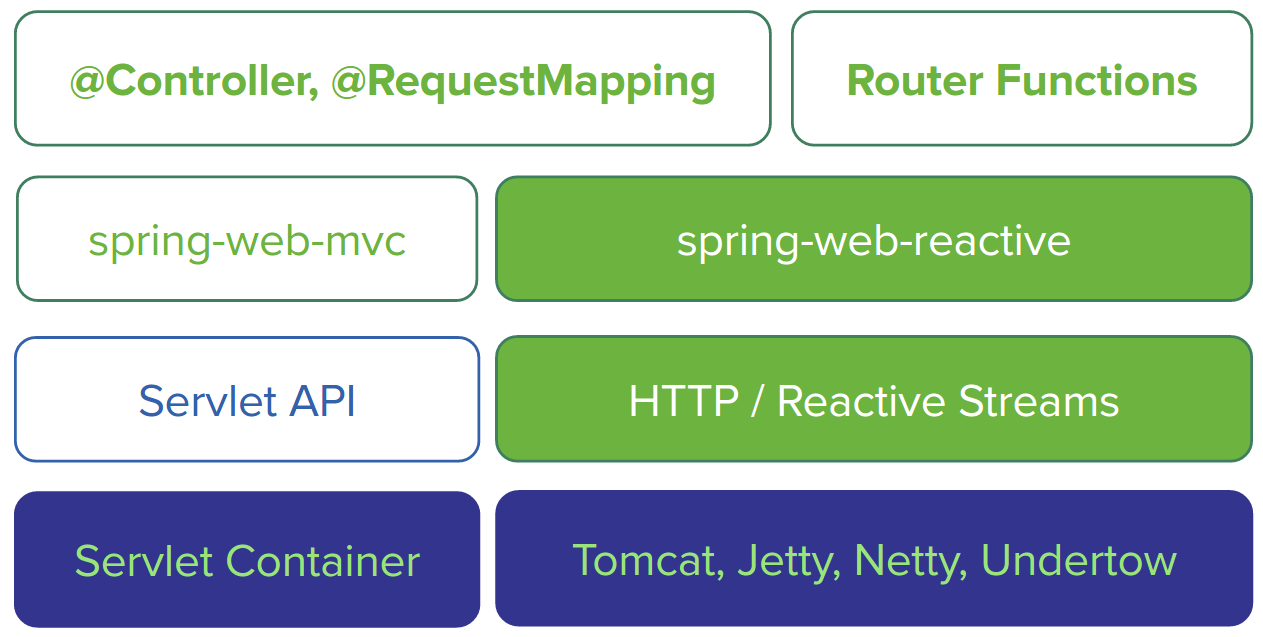
23.2.1 Server Side

On the server-side the new reactive module supports 2 distinct programming models:

Annotation-based with @Controller and the other annotations supported also with Spring MVC

Functional, Java 8 lambda style routing and handling

Both programming models are executed on the same reactive foundation that adapts non-blocking HTTP runtimes to the Reactive Streams API. The diagram below shows the server-side stack including traditional, Servlet-based Spring MVC on the left from the spring-web-mvc module and also the reactive stack on the right from the spring-web-reactive module.



The new reactive stack can run on Servlet containers with support for the Servlet 3.1 Non-Blocking IO API as well as on other async runtimes such as Netty and Undertow. Each runtime is adapted to a reactive ServerHttpRequest and ServerHttpResponse exposing the body of the request and response as Flux<DataBuffer>, rather than InputStream and OutputStream, with reactive backpressure. REST-style JSON and XML serialization and deserialization is supported on top as a Flux<Object>, and so is HTML view rendering and Server-Sent Events.

Annotation-based Programming Model

The same @Controller programming model and the same annotations used in Spring MVC are also supported on the reactive side. The main difference is that the framework contracts underneath — i.e. HandlerMapping, HandlerAdapter, are non-blocking and operate on the reactive ServerHttpRequest and ServerHttpResponse rather than on the HttpServletRequest and HttpServletResponse. Below is an example with a reactive controller:

@RestController

public class PersonController {

private final PersonRepository repository;

public PersonController(PersonRepository repository) {

this.repository = repository;

}

@PostMapping("/person")

Mono<Void> create(@RequestBody Publisher<Person> personStream) {

return this.repository.save(personStream).then();

}

@GetMapping("/person")

Flux<Person> list() {

return this.repository.findAll();

}

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

Mono<Person> findById(@PathVariable String id) {

return this.repository.findOne(id);

}

}

Functional Programming Model

The functional programming model uses Java 8 lambda style routing and request handling instead of annotations. The main API contracts are functional interfaces named RouterFunction and HandlerFunction. They are simple but powerful building blocks for creating web applications. Below is an example of functional request handling:

PersonRepository repository = ...

RouterFunctions

.route(GET("/person/{id}").and(accept(APPLICATION\_JSON)), request -> {

int personId = Integer.valueOf(request.pathVariable("id"));

Mono<ServerResponse> notFound = ServerResponse.notFound().build();

return repository.findOne(personId)

.then(person -> ServerResponse.ok().body(Mono.just(person), Person.class))

.otherwiseIfEmpty(notFound);

})

.andRoute(GET("/person").and(accept(APPLICATION\_JSON)), request ->

ServerResponse.ok().body(repository.findAll(), Person.class))

.andRoute(POST("/person").and(contentType(APPLICATION\_JSON)), request ->

ServerResponse.ok().build(repository.save(request.bodyToMono(Person.class))));

For more on the functional programming model see the [M3 release blog post](https://spring.io/blog/2016/09/22/new-in-spring-5-functional-web-framework).

23.2.2 Client Side

Spring Framework 5 includes a functional, reactive WebClient that offers a fully non-blocking and reactive alternative to the RestTemplate. It exposes network input and output as a reactive ClientHttpRequest and ClientHttpRespones where the body of the request and response is a Flux<DataBuffer> rather than anInputStream and OutputStream. In addition it supports the same reactive JSON, XML, and SSE serialization mechanism as on the server side so you can work with typed objects. Below is an example of using the WebClient which requires a ClientHttpConnector implementation to plug in a specific HTTP client such as Reactor Netty:

WebClient client = WebClient.create(new ReactorClientHttpConnector());

ClientRequest<Void> request = ClientRequest

.GET("http://example.com/accounts/{id}", 1L)

.accept(APPLICATION\_JSON)

.build();

Mono<Account> account = client

.exchange(request)

.then(response -> response.bodyToMono(Account.class));

|  |
| --- |
|  |
| The AsyncRestTemplate also supports non-blocking interactions. The main difference is it can’t support non-blocking streaming, like for example [Twitter one](https://dev.twitter.com/streaming/overview), because fundamentally it’s still based and relies on InputStream and OutputStream. |

23.2.3 Request and Response Body Conversion

The spring-core module provides reactive Encoder and Decoder contracts that enable the serialization of a Flux of bytes to and from typed objects. The spring-web module adds JSON (Jackson) and XML (JAXB) implementations for use in web applications as well as others for SSE streaming and zero-copy file transfer.

For example the request body can be one of the following way and it will be decoded automatically in both the annotation and the functional programming models:

Account account — the account is deserialized without blocking before the controller is invoked.

Mono<Account> account — the controller can use the Mono to declare logic to be executed after the account is deserialized.

Single<Account> account — same as with Mono but using RxJava

Flux<Account> accounts — input streaming scenario.

Observable<Account> accounts — input streaming with RxJava.

The response body can be one of the following:

Mono<Account> — serialize without blocking the given Account when the Mono completes.

Single<Account> — same but using RxJava.

Flux<Account> — streaming scenario, possibly SSE depending on the requested content type.

Observable<Account> — same but using RxJava Observable type.

Flowable<Account> — same but using RxJava 2 Flowable type.

Flux<ServerSentEvent> — SSE streaming.

Mono<Void> — request handling completes when the Mono completes.

Account — serialize without blocking the given Account; implies a synchronous, non-blocking controller method.

void — specific to the annotation-based programming model, request handling completes when the method returns; implies a synchronous, non-blocking controller method.

23.2.4 Reactive WebSocket Support

The Spring Framework 5 spring-web-reactive module includes reactive WebSocket client and server support. Both client and server are supported on the Java WebSocket API (JSR-356), Jetty, Undertow, Reactor Netty, and RxNetty.

On the server side, declare a WebSocketHandlerAdapter and then simply add mappings to WebSocketHandler-based endpoints:

@Bean

public HandlerMapping webSocketMapping() {

Map<String, WebSocketHandler> map = new HashMap<>();

map.put("/foo", new FooWebSocketHandler());

map.put("/bar", new BarWebSocketHandler());

SimpleUrlHandlerMapping mapping = new SimpleUrlHandlerMapping();

mapping.setUrlMap(map);

return mapping;

}

@Bean

public WebSocketHandlerAdapter handlerAdapter() {

return new WebSocketHandlerAdapter();

}

On the client side create a WebSocketClient for one of the supported libraries listed above:

WebSocketClient client = new ReactorNettyWebSocketClient();

client.execute("ws://localhost:8080/echo"), session -> {... }).blockMillis(5000);

Implementation—

<https://springframework.guru/spring-web-reactive/>

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

<https://dzone.com/articles/reactive-programming-with-spring-5>

<https://dzone.com/articles/whats-new-in-spring-framework-5>

<https://dzone.com/articles/reactive-spring-5-and-application-design-impact>