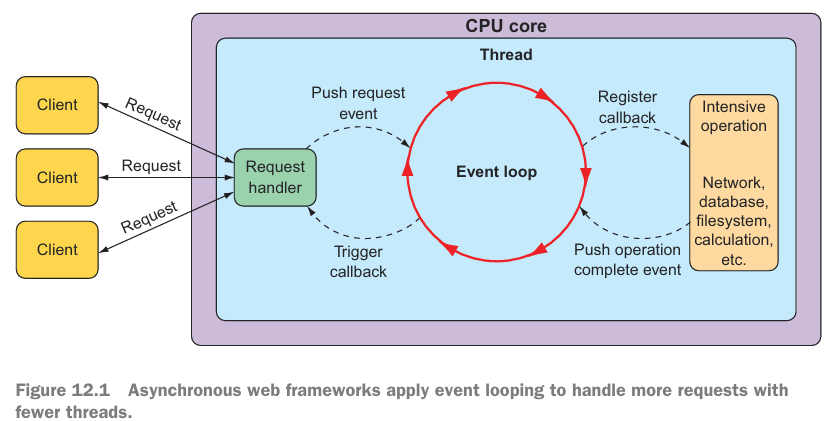
12. Developing reactive APIs

# 12.1 Working with Spring WebFlux

-Typical servlet web framworks (like Spring MVC) are blocking and multithreaded in nature, using a single thread per connection. As requests are handled, a worker thread is pulled from a thread pool to process the request. The request thread is blocked until it’s notified by worker thread that it’s finished.

-Blocking web frameworks won’t scale effectively under heavy request volume. Latency in slow worker threads makes things even worse because it’ll take longer for worker thread to be returned to the pool, ready to handle another request.

-Asynchronous web frameworks achieve higher scalability with fewer threads-generally one thread per CPU core. By applying a technique **event looping**, these frameworks are able to handle many requests per thread, making the per-connection cost more economical

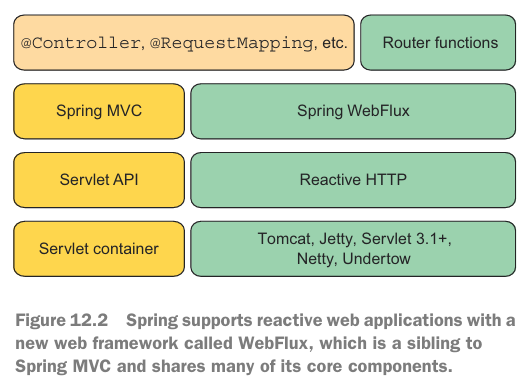


-In an event loop, everything is handled as an event (requests, callbacks from intensive operations like database and network operations). When a costly operation is needed, the event loop registers a callback for that operation to be performed in parallel, while it moves on to handle other events

-When operation is complete, it’s treated as an event by event loop, the same as request. As a result, asynchronous web frameworks are able to scale better under heavy request volume with fewer threads.

## 12.1.1 Introducing Spring WebFlux

-Spring WebFlux: a separate reactive web framework, borrowing as much from Spring MVC as possible



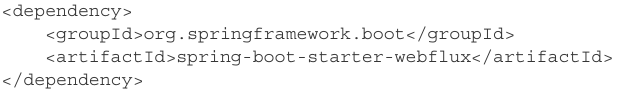
+Spring MVC sits atop Java Servlet API, which requires a servlet container (Tomcat) to execute on.

+Spring WebFlux builds on top of Reactive HTTP API, which is a reactive approximation of the same functionality provided by Servlet API. It doesn’t require a servlet container to run on. It can run on any nonblocking web container (Netty, Undertow, Tomcat, Jetty or any Servlet >3.1)

+Components between Spring MVC and Spring WebFlux: **annotations** used to define controllers.

+Top right corner represents an alternative programming model that defines controllers with a functional programming paradigm instead of annotation.

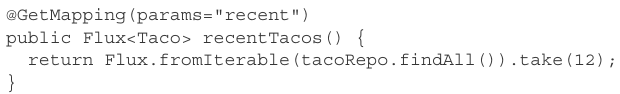
-Add SB WebFlux starter dependency:



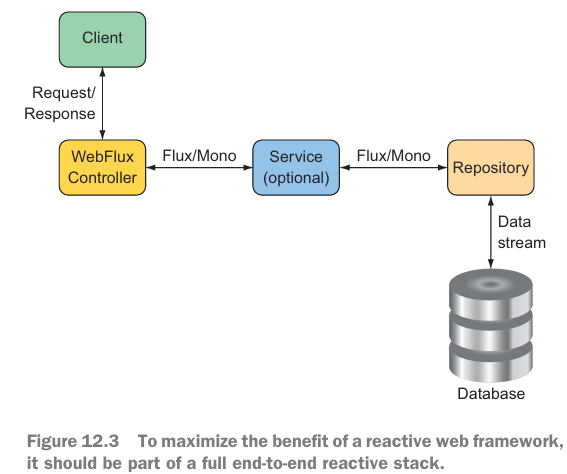
+The default embedded server for WebFlux is Netty: asynchronous, event-driven servers and is a natural fit for reactive web framework.

+**Spring WebFlux controller** methods usually accept and return **reactive types** like Mono and Flux.

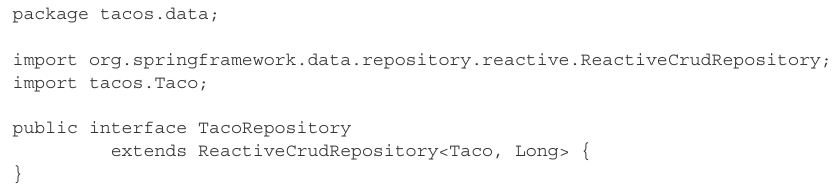
## 12.1.2 Writing reactive controllers



-A reactive controller will be the tip of a stack that’s reactive end to end, including controllers, repositories, database and any service between:

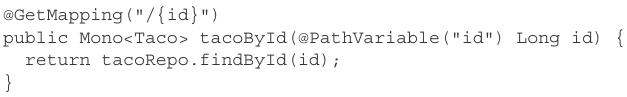


+An end-to-end stack requires that the repository be written to return Flux instead Iterable. It may look like:



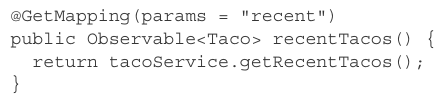
+Although getting Flux<Taco> back from repository, you can return it without calling subscribe(). The framework will call subscribe() for you. This means when request for /api/tacos?recent is handled, recentTacos() will be called and will return before data is event fetched from database

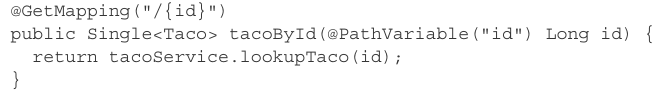
-Returning single values:



-Working with RxJava types:

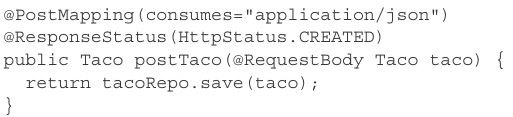
+Suppose there’s a service sitting between TacoController and backend repository that deals in terms of RxJava types:





+Spring WebFlux controller methods can also return RxJava’s Completable (Mono<Void> in Reactor), Flowable as an alternative to Observable

-Handling input reactively:



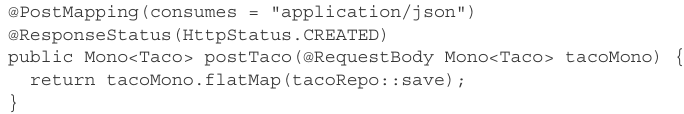
+postTaco() can’t invoked until the request payload has been resolved and used to instantiate Taco object -> postTaco() can’t return until blocking call to save() returns ->The request is blocked twice: enters postTaco()and inside postTaco()



+saveAll(): accept any implementation of Reactive Streams Publisher (Mono + Flux). next() to obtain a Mono<Taco> from Flux<Taco>.

+The method is invoked immediately without waiting for Taco to be resolved from request body. The repository is also reactive, it’ll accept Mono and immediately return Flux<Taco>, next() and return Mono<Taco>… all before the request is even processed.

+Alternatively:



Taco contained within tacoMono is handed to save() via flatMap(), resulting in a new Mono<Taco>

# 12.2 Defining functional request handlers

# 12.3 Testing reactive controllers

## 12.3.1 Testing GET requests

## 12.3.2 Testing POST requests

## 12.3.3 Testing with a live server

# 12.4 Consuming REST APIs reactively

## 12.4.1 GETting resources

## 12.4.2 Sending resources

## 12.4.3 Deleting resources

## 12.4.4 Handling errors

## 12.4.5 Exchanging requests

# 12.5 Securing reactive web APIs

## 12.5.1 Configuring reactive web security

## 12.5.2 Configuring a reactive user details service

# Summary