TECHNICAL WHITE PAPER

PLAY FRAMEWORK: THE JVM ARCHITECT'S PATH TO SUPER-FAST WEB APPS

THE POWER OF A STATELESS, STREAMING, REACTIVE WEB FRAMEWORK

By Will Sargent | Lightbend, Inc.

Lightbend

Table of Contents

Executive Summary

Not all JVM web frameworks are created equal. When it comes to modern, Reactive systems, the same old technologies that have been powering the last 10-15 years of web applications are simply not designed to efficiently build highly-distributed systems running on multicore, cloud-based environments.

This technical whitepaper describes how Play Framework – the asynchronous, non-blocking, stateless, streaming, open source Reactive web application framework from Lightbend – allows enterprises to achieve a higher level of performance and efficiency with less infrastructure/HW than before, empowering them to meet their customers' needs while saving money on cloud expense and long development cycles.

Play Technical Overview

Play is an **asynchronous, non-blocking, stateless, streaming, Reactive web application framework**. Play is fast. But let's talk about what that really means.

How To Go Fast

Let's say you want to make something go fast — a car, for example.

If you want to build a fast car, you'll need a powerful engine.

However, just because you have that power doesn't mean you have access to all of it. Without the proper engineering, the power in your engine won't make it to the rest of the car. Imagine sticking a fast engine in a car that can't handle it — you'll hear a horrible grinding sound, then the car will lurch forward and you'll smell oil and burning. Something went wrong somewhere — but what?

The engine produces the power, but the transmission takes that power from the engine and delivers it to the tires. So you make a transmission that works with the engine. Now the car goes fast.

But having a fast car isn't enough. You need tires that can handle that power without melting. You need a fueling system that can get fuel to the engine at the rate you need. You need a steering system and brakes that can let you turn and stop the car.

Making something go fast involves a number of small pieces working in perfect synchronicity to deliver a smooth and seamless experience. But there's one final piece that makes a car go fast — the person driving it.

When Jackie Stewart was asked what made a great racing driver, he replied, "You don't have to be an engineer to be be a racing driver, but you do have to have Mechanical Sympathy." Martin Thompson took the phrase "mechanical sympathy" to discuss software that takes advantage of the underlying rhythms and efficiencies of hardware.

Play goes a step further and takes into account "the person driving the car." Play's goal is to provide a seamless general purpose web application framework that works in mechanical sympathy with the CPU, garbage collector, and external data sources.

So that's the goal. Now let's talk about computers and web applications, and discuss where and how Play is fast.

Why It's Important To Be Asynchronous And Non-Blocking

A computer consists of at least one CPU (made up of several cores), a small amount of onboard CPU cache, a much larger amount of RAM, some persistent storage, and a network IO card.

A web application listens on a TCP socket for HTTP requests, and when a request comes in, it typically does the following:

- Accumulates data in the request until request completion, using the CPU
- Processes the action to take on the accumulated request, using the CPU
- Does some lookups of data
  - From cache (fast)
  - From RAM (not so fast)
  - From persistent storage (very slow)
  - From external data sources through the network card (very, very slow)
- Potentially sends updates and commands to external systems (also slow)
- Creates a response using the CPU based on the results of operation
- Streams that response back through the network card

The good news is that today's CPUs — the engine — are massively more powerful than they used to be. The bad news is that the rest of the system hasn't adjusted for the large amounts of power now available. Now let's talk more about CPUs and cores, and point out what Play does to make effective use of CPUs.

The Evolution Of CPU And Cores

A CPU consists of a set of cores. Historically, one CPU meant one core, and there was no parallelism possible. Many of the assumptions behind programming are still based around single cores. But this is no longer the case — it's difficult to get a sense of just how fast modern CPUs are, how many cores are available, and just how much time they spend waiting for work. For example, the Xeon E5 2699 v5 is rumored to have 32 cores available on a single socket.

From **Systems Performance: Enterprise and the Cloud**, we can see the disparity between the time spent processing a full CPU cycle and the time spent waiting for work to get to the CPU:

Operation	Time to Execute	Time scaled to CPU cycle
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access	120 ns	6 min
Solid-state disk I/O	50-150 µs	2-6 days
Rotational disk I/O	1-10 ms	1-12 months
Internet: SF to NYC	40 ms	4 years
Internet: SF to UK	81 ms	8 years
Internet: SF to Australia	183 ms	19 years

So from this chart and from the description we gave above, you can see that processing an HTTP request has many different stages all involving small amounts of work, from accumulating TCP packets into an HTTP request to querying and updating external systems.

Modern CPUs are so fast that they spend a good deal of time waiting for RAM.[1]

Orchestrating Processes, Threads, And Cores

The operating system runs a set of processes. Processes consist of a number of threads, but at the very least every process must have one thread. A thread is defined by Wikipedia as "the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system." Threads are also commonly defined as "units of scheduling and execution," because a thread can only run tasks sequentially.

Every time a thread runs, it runs one of the cores on the CPU at 100%. The number you see when a CPU says it's running at 20% isn't denoting that the CPU is running 20% of the work — it's really saying that 80% of the time, the CPU was idle. Many times, this means you've got a four core CPU with one of the cores pegged at 100% and the other cores standing idle. That's a classic single threaded model — one of the threads is being fed work, but a thread can't run on more than one core at a time.

By definition, a thread is a unit of execution and so can only run a sequential series of tasks. Now, if you had four threads, then you could run work on all four cores at once. And if the threads could steal work from one another, then even if one core was busy running a particularly large task, you could keep all four

1 For more on this, see
http://www.brendangregg.com/blog/2017-05-09/cpu-utilization-is-wrong.html and
http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/

cores occupied with work and your entire CPU could run at 100%. So the gating factor in processing is not the CPU, but in getting enough work to the CPU to keep it busy.

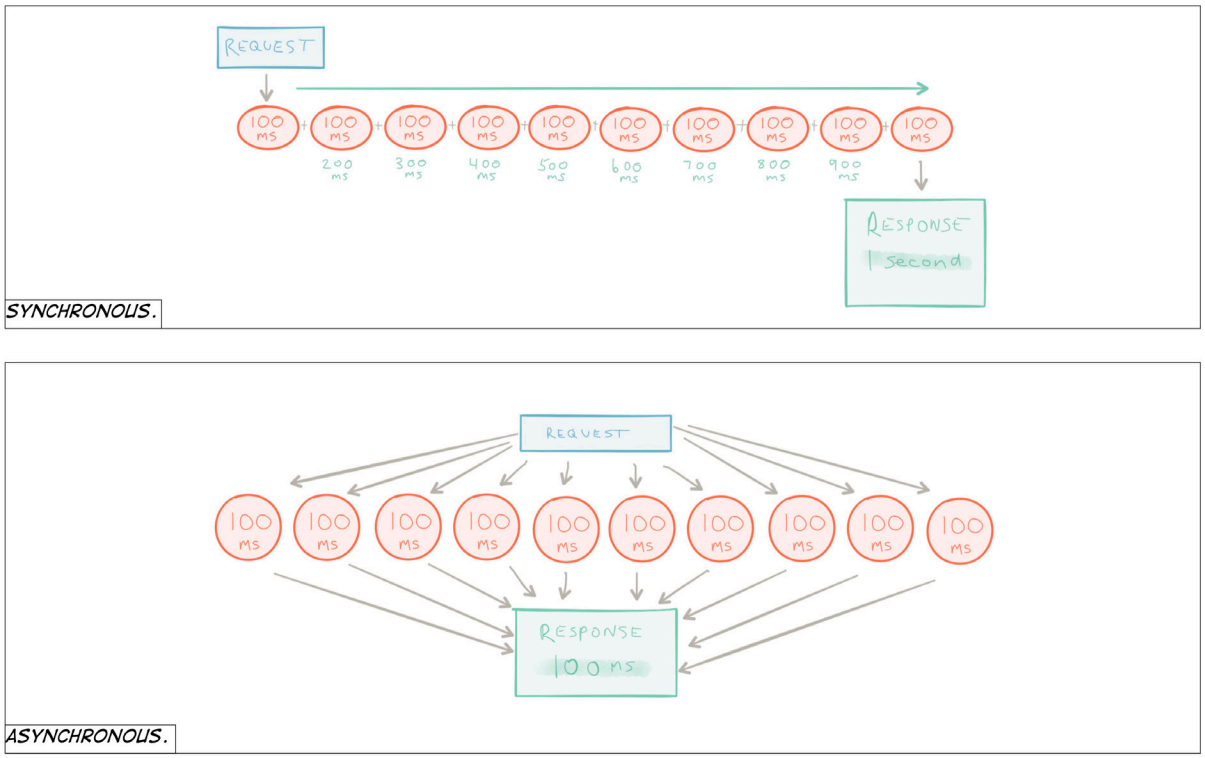

Image courtesy of Kevin Webber (@kvnwbbr)

When there is likely going to be a delay, we want the core to immediately start performing another task and only come back to processing the request when the results are available. When the core is idle and waiting for results to come back, CPU cycles are left on the table that could be put to better use.

Code that calls out to the filesystem, the network, or to any system that is far slower than CPU + RAM is called “blocking” because the thread will remain idle while this is happening. Likewise, code that is written to keep the CPU ready to perform any available work at all times is called “non-blocking”.

(Incidentally, while you may see a four core CPU as having eight cores due to hyperthreading, this is actually the CPU itself doing the same non-blocking trick — squeezing in some more CPU cycles from one thread while another thread is waiting for data from RAM. So hyperthreading does boost performance, but no more than around 30%.)[2]

2 **https://software.intel.com/en-us/articles/how-to-determine-the-effectiveness-of-hyper-threading-technology-with-an-application/**

The old way was to run a process single threaded, and if there were multiple cores or additional processing power was needed, multiple processes would be spawned. This is a common implementation of parallelism for dynamic programming languages, as many interpreted languages use a global interpreter lock that precludes a thread-based implementation.[3]

Java takes the opposite approach: a single JVM can effectively control an entire machine and all its cores, leveraging the strengths of the concurrency support and well-defined memory model added in JDK 1.5 (JSR166).[4] Using threads, the JVM can run multiple tasks in parallel in the same memory space.

The Java way of doing things was to create multiple threads and run them in the same memory space. This proved tricky, as programmers had to work out patterns to ensure memory safety for multithreaded programming. However, multiple threads were useful even on a single core machine, because if one thread was blocked waiting for network or IO, another thread could step in to use the CPU.

It was in this environment that servlets were invented, in 1997. Servlets were written assuming a thread per request model — every request was assigned a thread, and if a thread needed to call out to a database, it would block and another thread would be swapped in. This was a great solution for the time, but as CPUs scaled up, blocking would become more and more of a problem.[5]

The Problem With Threads

Q: *So why is blocking a thread a problem? Why not just use more threads?*

A: Because running large numbers of threads works against mechanical sympathy.

Creating a thread is very expensive from a CPU perspective. Threads are also memory-expensive. And because threads are expensive, the kernel puts hard limits on the number of threads that can be run in a process.

Even past the creation and memory overhead, threads have a pernicious runtime cost — if threads are runnable (i.e. not blocked by IO), then they cause CPU time-slice overhead and unnecessary context switching [6], which in turn causes L2 cache pollution and hence more cache misses.

The class Servlet API struggles with the association of threads to requests. Pre Servlet 3.0, Tomasz Nurkiewicz described handling more than 100-200 concurrent connections as “ridiculous”. [7]

3 **https://en.wikipedia.org/wiki/Global_interpreter_lock**
4 **https://jcp.org/en/jsr/detail?id=166**
5 **https://manuel.bernhardt.io/2017/05/15/akka-anti-patterns-blocking/**
6 **http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html**
7 **http://www.nurkiewicz.com/2011/03/tenfold-increase-in-server-throughput.html**

So rather than use a large number of threads, the goal is to work with mechanical sympathy, and make the best use of a small number of threads.

The classic way to do this in Java was to use a programming technique called callbacks. The code would send off a request to a remote system. When the remote system was ready, a callback method would be called to trigger the CPU to do some more processing — something like onCallback(DataReceivedEvent event) with an event containing the results.

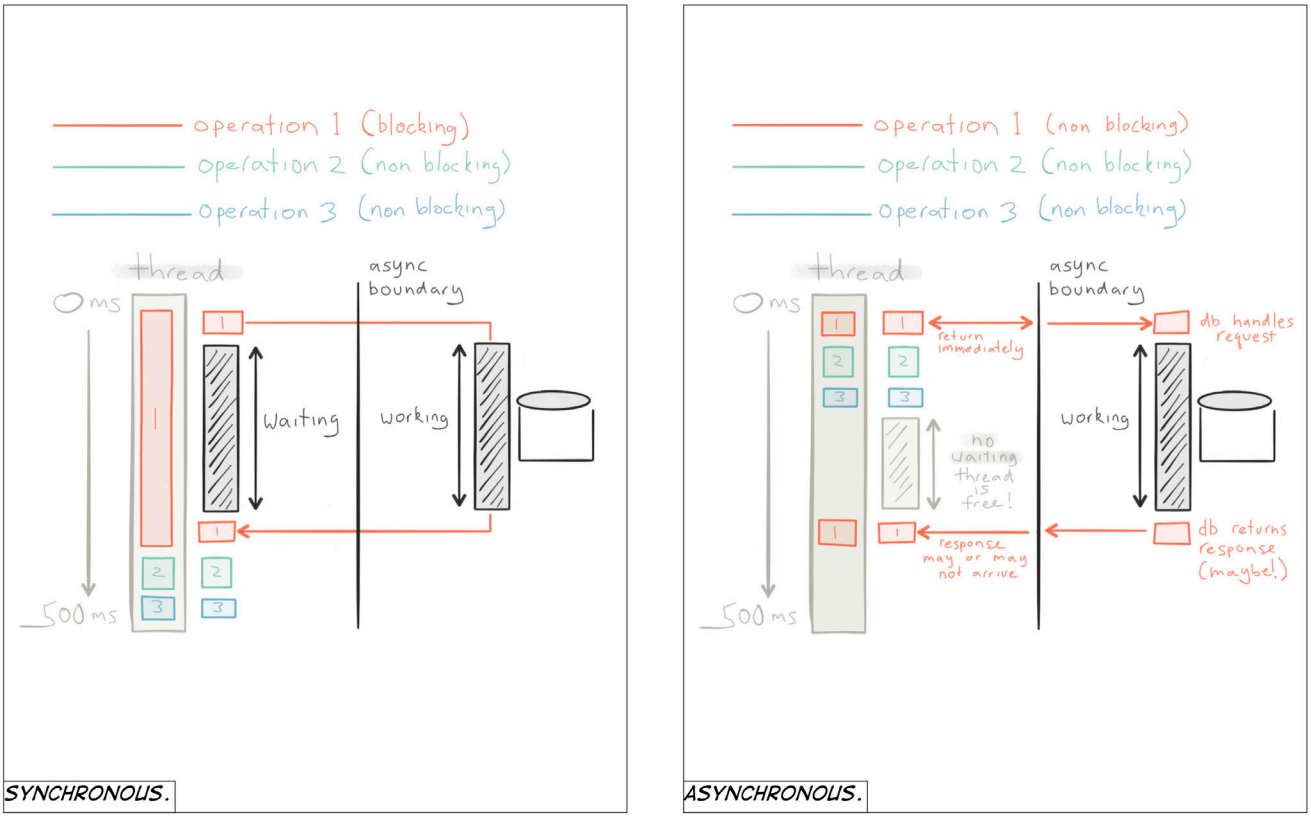

Image courtesy of Kevin Webber (@kvnwbbr)

In the Java model, the callback is invoked after the function returns, and may happen on another thread's stack. Because the callback can be deferred, if you have two callbacks registered to the event, then there is nothing that says those callbacks have to happen in a particular order: callback A can happen before callback B, or callback B can happen before callback A. Systems that do not impose an order in this way are called **asynchronous**.

This is the model that Netty uses under the hood, which is why Netty is described as an **asynchronous, non-blocking, event-driven** framework. Netty uses a small number of threads sized to the CPU, and uses asynchronous IO with Java NIO to provide an architecture capable of running more than 10,000 concurrent connections on a single machine, **solving the "C10K" problem**. (Callbacks are also used in Servlet 3.0, which added an AsyncServlet class to the Servlet API. In 2011, Nurkiewicz reported AsyncServlet increased capacity to 1000-2000 concurrent connections.)

Reactive Application

Unfortunately, callbacks don't compose very well. The term "**callback hell**" became popularized because dependent data works like this:

```
fooHandler.getFoo(42, new GetFooCallback()
{
    public void onResult(final Foo foo)
    {
        barHandler.getBar(foo.getBarID(), new GetBarCallback()
        {
            public void onResult(final Bar bar)
            {
                zorbHandler.sendZorbToServer(new Zorb(foo, bar), new ZorbResponseHandler()
                {
                    public void onSuccess()
                    {
                        // Keep dancing
                    }
                    public void onFailure()
                    {
                        // Drop the spoon
                    }
                });
            }
        });
    }
});
```

Using a callback model is fiddly and easy to get wrong. It's like a car with a finicky manual transmission that only changes gears if you time the shift just right. You want to feel that the engine is responsive and easy under your hands. In the same way, you want an API that's fluent and feels right.

Doug Lea and the Java team at Oracle put together the java.util.concurrent package and expended significant effort into upgrading Java's concurrency support. In JDK 1.8, the CompletionStage API allowed for true Promise / Future based support, meaning that instead of callbacks, a series of operations could be put together using an underlying executor that connects processing work to threads.

So using the CompletionStage API, you have code that looks like this:

```
CompletionStage<Result> result = CompletableFuture.supplyAsync(() -> {
   return queryDatabase();
}, databaseExecutor).thenApplyAsync(result -> {
   return queryAnotherDatabase(result);
}, databaseExecutor).thenApplyAsync(result -> {
   return doSomeCPUBoundProcessingWork(result);
}, workStealingCpuBoundExecutor);
```

Using CompletionStage means that blocking work is only run on the databaseExecutor (which is sized for a thread pool that can handle it), while CPU-based work uses a small thread pool that is sized to the core and can perform work stealing.

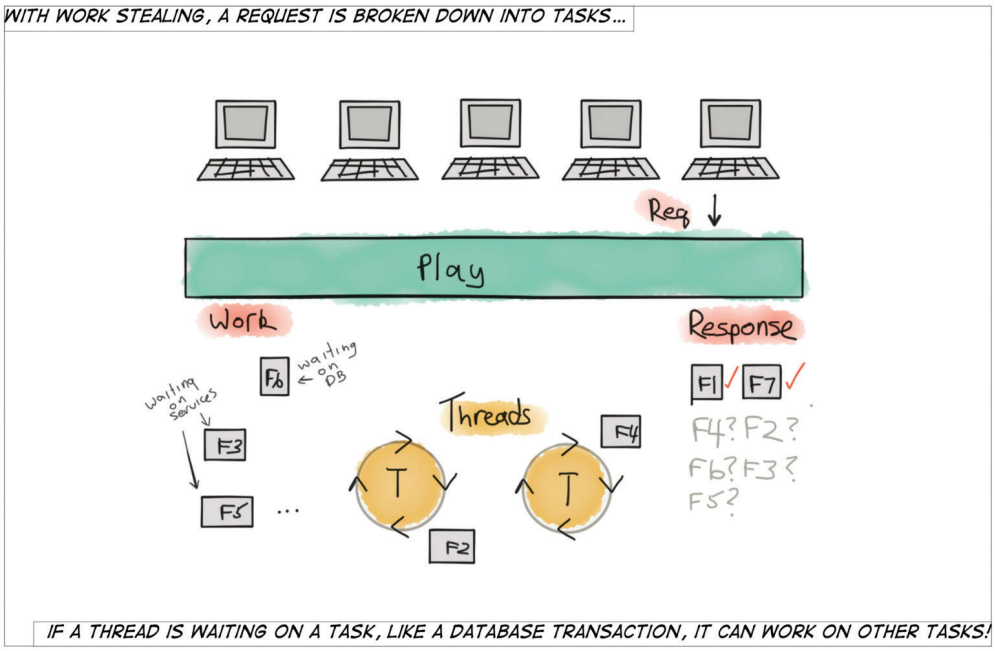

Image courtesy of Kevin Webber (@kvnwbbr)

You'll get some benefit out of using CompletionStage, but it will still be awkward if your framework uses callbacks or a thread per request model under the hood, because the engine doesn't line up with the transmission. You don't want to switch gears.

What you really want to do is return a CompletionStage<Result> to your web framework and be able to specify custom execution contexts, so you can work in mechanical sympathy with the CPU and hardware. This is exactly what Play does.

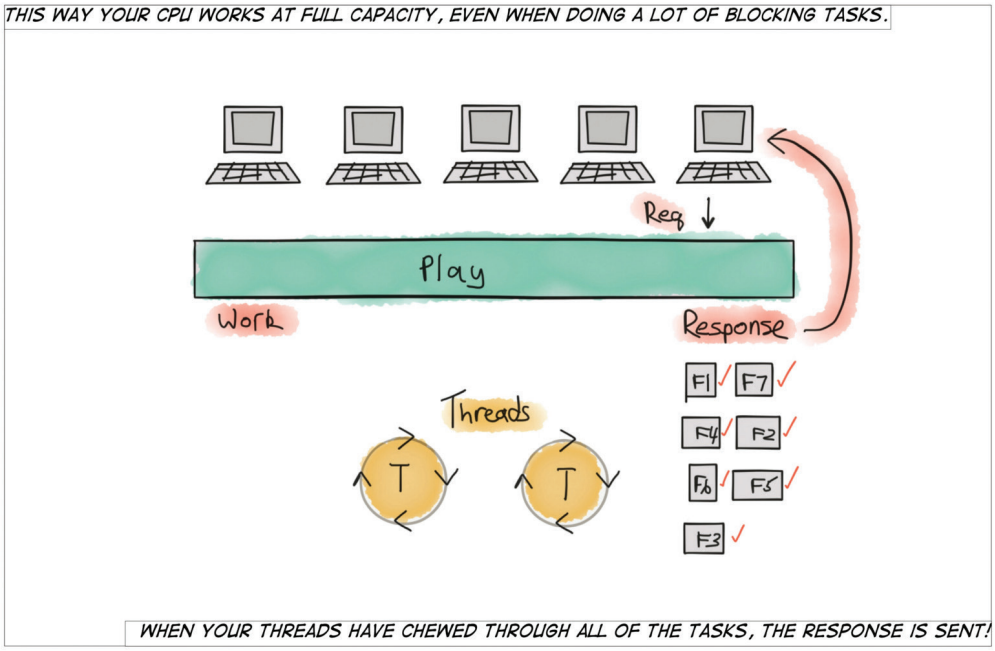

Image courtesy of Kevin Webber (@kvnwbbr)

Using CompletionStage to provide a Reactive API is how Play fulfills its goal of enabling programmers to work in mechanical sympathy in a non-blocking, asynchronous system. You can define your domain API to return CompletionStage<T> to establish an asynchronous boundary, and you can use your own executors and then map and stage to an Http.Result in Play. There are multiple **examples on the Play Framework website** that demonstrate this paradigm in action.

Roughly speaking, you can consider the core of Play to be a "future-based wrapper" that applies functional programming around a low level HTTP engine such as Netty or Akka-HTTP. Internally, Play receives a series of byte chunks from the incoming HTTP request (either converting from DataReceivedEvent using Netty, or a direct stream of ByteStream elements from Akka-HTTP) and uses an Accumulator to accumulate those chunks into a single Request that can be exposed. Likewise, when a result is being rendered, Play converts the Result or CompletionStage<Result> into a stream of outgoing bytes.[8]

The term for a framework that uses Future-based constructs (such as CompletionStage) instead of callbacks to put together a processing chain of work is a "**reactive application**." The term gets conflated[9] with "web application," so "reactive web application framework" is also commonly used.[10]

8 **https://www.playframework.com/documentation/2.5.x/ScalaEssentialAction**
9 **https://www.oreilly.com/ideas/reactive-programming-vs-reactive-systems**
10 **https://blog.redelastic.com/what-is-reactive-programming-bc9fa7f4a7fc**

Reactive Example

Enough talking—let's look at a real world example. Here's some asynchronous code in Play:

```
public class AsyncController extends Controller {

    public CompletionStage<Result> index() {
        return getFutureMessage().thenApply(Results::ok);
    }

    private CompletionStage<String> getFutureMessage() {
        return CompletableFuture.supplyAsync(() -> “Hello world”);
    }
}
```

If you don't have any asynchronous bits to worry about, then you can always render a Result directly:

```
public class HomeController extends Controller {
    public Result index() {
        return ok(“Hello world”);
    }
}
```

More practically, here's a Java REST API example in Play:

```
public class PostController extends Controller {
    ...
    public CompletionStage<Result> list() {
        return repository.list().thenApplyAsync(resultStream -> {
           // get current request with request() method in this block
           return ok(Json.toJson(resultStream.collect(Collectors.toList())));
        }, httpExecutionContext.current());
    }
}
```

Stateless

HTTP is a stateless application protocol. There is no accumulated state that connects one HTTP request to another — every request is handled and processed individually, with no memory of what happened before. This is great for REST, because REST is literally about transferring state using stateless requests. But this isn't so great for clients that want to use HTTP to render HTML and Javascript.

A client that wants to tie HTTP requests together has to pass in some state, in the form of a cookie or a header with a token value. This token is typically used to establish the identity of the client with the server across multiple HTTP requests. Web application frameworks have two options once there's a session token and they want to keep some extra server-specific information around — they can store additional

state on the client, or they can store state on the server, in memory, or hand off that state to an external service.

Early web application frameworks kept information on the server in memory. The Servlet API contains a HttpSession available from the request, and the Java EE specification contains no mechanism to limit the amount of memory a session uses. Keeping information on the server is convenient, but it carries significant drawbacks.

If there are several servers, either all of them need to have the current session information kept in sync, or all requests on a single session need to be directed to the server containing the server information, thus limiting horizontal scalability and resiliency. In addition, the server must keep a tight lid on the number of concurrent sessions it can manage because server side state takes up memory. If there are too many sessions on one server, it could run low on memory (causing rapid GC cycles) or run out of memory completely.

Play does not keep any kind of server side state between HTTP requests, and so is **stateless** by design. On request, Play will create a PLAY_SESSION cookie which is cryptographically signed so it cannot be modified by the client, allowing it to hold state in the cookie. A Play session cookie is very limited though and should only be used to hold a session token and other small pieces of information that are not security-sensitive, such as the user's timezone and display preferences.

Recommendations On Storing State

To store security-sensitive information, the Play team recommends a combination of encrypted client side data and a backend session service to store the session encryption key. A straight SQL database such as PostgreSQL or MySQL is perfectly adequate in most cases for session data, but you can use Akka Distributed Data or Akka Cluster Sharding to store information across all Play instances (**see this complete example**).

Streaming, HTTP, Akka & Reactive Streams

So that takes care of regular HTTP. Now let's talk about streaming.

Streaming is what happens when a request or response may continue indefinitely, without knowing when it will terminate. Streaming has been around for some time in HTTP with chunked transfer encoding, but it only really got underway with Comet, and then started to hit its stride with Server-Sent Events and finally Websockets. (Interestingly, although HTTP/2 is based around streams internally, there is no Websockets for HTTP/2 — if you want to upgrade to Websocket, it's going to have to be over HTTP 1.1.)

Streaming is used for real time notification and interactive websites. It's a different paradigm from a regular request / response cycle. In the case of HTTP, you must receive the entire request before you can send a response. In the case of Websocket, you have a full duplex connection where input and output can happen simultaneously, and connections can be left hanging open.

This means that application servers must be very careful not to store any information internally when it comes to streaming, because any internal state can quickly balloon with the number of streaming connections and incoming data, at which point the server runs out of memory and shuts down. An application server that does not store any internal state about incoming requests or streams is called a "stateless" server.

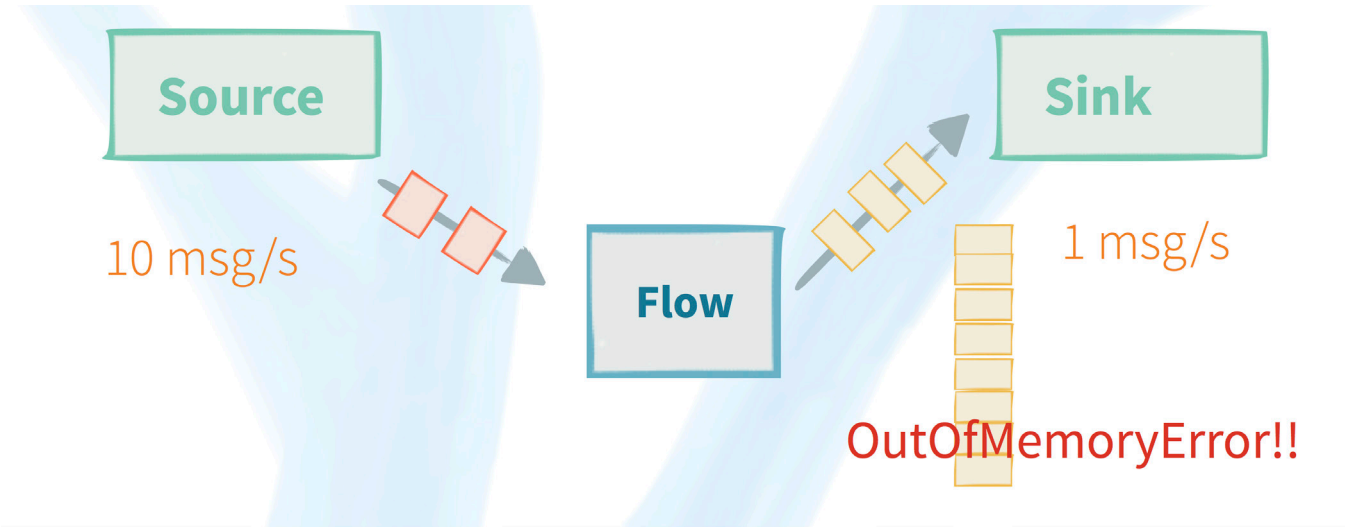

Image courtesy of Konrad Malawski (@ktosopl)

There's also the problem of determining how to "say no" to more data. Standard HTTP has a very simple means of refusing data in the form of rate limiting — it can return HTTP Status code 429 "Too Many Requests," with a Retry-After header indicating when the client can retry the request.

But this doesn't work in a streaming system because there's no status code involved. So for streaming, backpressure is used as a means of flow control.

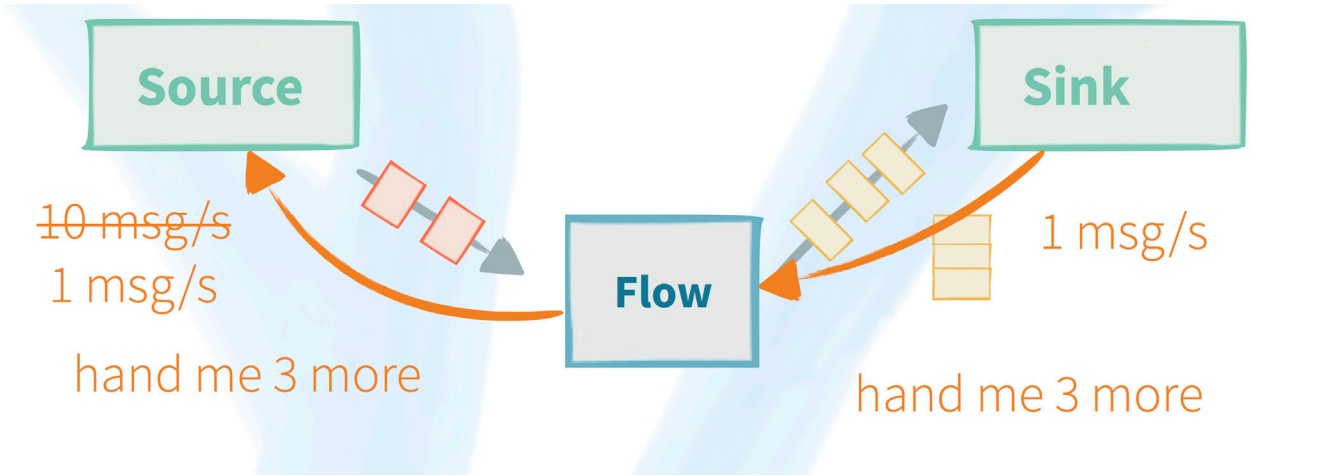

Image courtesy of Konrad Malawski (@ktosopl)

Let's discuss Reactive Streams a bit, because there's some confusion around this area. **Reactive Streams** is an asynchronous, non-blocking, streams-based SPI (service provider interface). Specifically, Reactive Streams is built to handle backpressure — if one part of the system clogs up, Reactive Streams ensures that the rest of the system will work fine.

Akka Streams is the Lightbend implementation of Reactive Streams. Streams consist of a Source, which produces elements, a Sink, which consumes them, and a Flow, which is a template that determines the behavior of a plugged-in Source and Sink.

Server Sent Events Example

```
public class JavaEventSourceController extends Controller implements JavaTicker {
    public Result streamClock() {
        final Source<EventSource.Event, ?> eventSource = getStringSource().map(Event-
Source.Event::event);
        return ok().chunked(eventSource.via(EventSource.flow())).as(Http.MimeTypes.EVENT_
STREAM);
    }
}
```

https://github.com/playframework/play-java-streaming-example

Websockets Example

```
public class HomeController extends Controller {
    public WebSocket ws() {
        return WebSocket.Json.acceptOrResult(request ->
                wsFutureFlow(request).thenApplyAsync(Either::Right)
.exceptionally(this::logException);
        );
    }

 public CompletionStage<Flow<JsonNode, JsonNode, NotUsed>> wsFutureFlow(Http.Request-
Header request) {
...
    }
}
```

https://github.com/playframework/play-java-websocket-example

Complex Streams Example

```
public class HomeController extends Controller {
    private final Flow<String, String, NotUsed> userFlow;

    @Inject
    public HomeController(Materializer mat)
    {
Source<String, Sink<String, NotUsed>> source = MergeHub.of(String.class)
                .recoverWithRetries(-1, new PFBuilder().match(Throwable.class, e ->
Source.empty()).build());
        Sink<String, Source<String, NotUsed>> sink = BroadcastHub.of(String.class);

        Pair<Sink<String, NotUsed>, Source<String, NotUsed>> sinkSourcePair = source.
toMat(sink, Keep.both()).run(mat);
        Sink<String, NotUsed> chatSink = sinkSourcePair.first();
        Source<String, NotUsed> chatSource = sinkSourcePair.second();
        this.userFlow = Flow.fromSinkAndSource(chatSink, chatSource);
    }

    public WebSocket chat() {
        return WebSocket.Text.acceptOrResult(request -> {
            return CompletableFuture.completedFuture(F.Either.Right(userFlow));
        });
    }
}
```

https://github.com/playframework/play-java-chatroom-example

What It Means To Be A "Framework" vs "Library"

The framework piece is what distinguishes Play. Play comes with modules and defaults that let you start building a correctly-configured web application right away. A library has to be configured by hand.

Q: Should I Use Play Or Akka HTTP?

If you are using Play 2.6, then you are using Akka-HTTP. The difference between them is that Play is a framework, and Akka-HTTP is a library. This means that Play comes with a set of default filters and configuration options, whereas Akka-HTTP only has the options you give it. In addition, Play gives you a full development experience with instant reload, with full resource management bound to Play's application lifecycle and built in management of components through dependency injection and modules.

Q: *Is Play a microframework?*

Yes. The term "microframework" is nebulous, but is commonly used to refer to minimalistic web application frameworks. Play's core is lightweight and minimal. It's a Reactive application wrapper around an HTTP engine with a series of optional modules.

Here's the overall impact of Play — it has 35 dependencies in total:

https://app.updateimpact.com/treeof/com.typesafe.play/play_2.11/2.5.14
https://app.updateimpact.com/treeof/com.typesafe.play/play_2.12/2.6.0-M5

You don't need a dependency injection framework because Play lets you use interfaces:

```
public class MyComponents extends BuiltInComponentsFromContext implements HttpFilter-
sComponents, BodyParserComponents {

    public MyComponents(ApplicationLoader.Context context) {
         super(context);
    }

    @Override
    public play.routing.Router router() {
        RoutingDsl routingDsl = new RoutingDsl(scalaParsers(), javaContextCompo-
nents());
        return routingDsl.GET("/").routeTo(() -> ok("Hello")).build();
    }
}
```

https://www.playframework.com/documentation/2.6.x/ScalaEmbeddingPlayAkkaHttp

Q: *Is Play a REST API framework?*

Yes. See **https://developer.lightbend.com/guides/play-rest-api/index.html** for a guide. LinkedIn has thousands of REST API services running on Play.

The guide on the tech hub is in Scala, but you can see the Java version on Github at **https://github.com/playframework/play-java-rest-api-example**

Q: *Is Play a microservices framework?*

No, but Play is frequently part of a microservices framework, which is what Lightbend's customer Verizon does with their go90 platform (read the **case study**).

Microservices own their own persistence; Play is persistence agnostic. You can run Play without any database backend at all. There are hooks in Play to tie persistence frameworks into its application lifecycle, but Play doesn't manage persistence directly.

We recommend **Lagom**, which runs on Play and uses Akka Cluster with a backend persistence solution, to simplify development of microservices.

A single microservice does no good if you're not also thinking about the interaction (e.g. backpressure, partition) between microservices. This is what distinguishes a Reactive system from a domino system where bringing down one microservice takes out all of them.[11]

Q: *Is Play a secure framework?*

Security is baked into Play at every level. Play contains out-of-the-box protection against common XML and HTML attacks like XXE and XSS. For rendering and processing content, Play 2.6.x comes with a set of default filters protecting against CSRF, security headers, and DNS rebinding, with SLF4J marker logging for SIEM monitoring. Play uses JSON Web Tokens with SHA256 signatures for cookie encoding, and supports SameSite and cookies prefixes.

For HTTPS, you can run Play internally on JSSE or behind a TLS aware reverse proxy like nginx:

https://github.com/playframework/play-scala-tls-example

And for running an HTTPS client, Play WS comes with a number of HTTPS options:

https://www.playframework.com/documentation/latest/WsSSL

If you are wondering about authentication or authorization solutions, then refer to Silhouette or Play-Authenticate for authentication and Deadbolt 2 for authorization.

All Together Now

So, Play is an **asynchronous, non-blocking, stateless, streaming, Reactive web application framework**.

11 The Calculus of Service Availability **https://queue.acm.org/detail.cfm?id=3096459**

How Fast Is Play?

Pretty Fast!

Play is so fast that you may have to turn off machines so that the rest of your architecture can keep up. With Play, your web application framework is no longer the bottleneck in your architecture. If you deploy Play with the same infrastructure that you were using for other web frameworks, you can effectively stage a denial of service attack against your own database.

For example, **Hootsuite** was able to reduce its number of servers by 80% by switching to Play and Akka.

We've discussed streaming above: using Akka Streams, you can stream video through Play Websockets. **See the Youtube Video from two years ago**.

Play 2.6.x comes with the latest Akka 2.5.x, and it comes out-of-the-box with Akka HTTP, which provides HTTP/2 support.[12]

Why Is It Fast?

Under the hood, Play works with Netty or Akka-HTTP to process requests as efficiently as possible. It exposes asynchronous work through the CompletionStage API so that developers can write non-blocking asynchronous code easily.

Play optimizes for the case where many requests will involve some amount of flow context switching off the main thread pool, and assumes that CPU bound work is "gappy" and will come in short bursts. Play uses a small number of threads in a fork/join thread pool that ensures a constant and balanced stream of work to the cores, optimizing for concurrency.

A Single Server Can Go This Fast

The Play team runs nightly load tests against Prune to monitor and measure improvements and catch regressions.

https://github.com/playframework/prune

12 See **http://www.httpvshttps.com/**, **https://www.troyhunt.com/i-wanna-go-fast-https-massive-speed-advantage/** and listen to this podcast: **https://www.lightbend.com/blog/lightbend-podcast-play-2-6-is-coming-and-its-faster-than-ever** for more

Prune runs on Vegemite, a **Xeon E5-2430L v2 2.4GHz** (6 cores, 12 threads at 2.5GHz, from 2014) running with a JVM 1.8.0 JVM with 1 GB heap.

In a recent load test, Play runs around 60K requests a second with a 95% response time of 0.83 milliseconds.

And when servicing Websocket connections, an AWS c3.2xlarge instance running Play 2.4 and Netty is **able to serve 100k simultaneous connections**.

But Here's Why You Shouldn't Go That Fast

Now that we've given you some load tests numbers, let's explain why you shouldn't use load tests as your only metric in production.

The first problem is that load tests are inherently unrealistic. The ideal load test happens on a machine on the local network, executing as many requests as possible over the wire repeatedly with no variation or delays from looking up external data from IO or network. All the gaps are squeezed out of the system. This is almost the exact opposite of a real application.

The second caveat is that a load test gives an incomplete understanding of latency. Using CPU utilization as a rough metric, when you start getting up to around 70% utilization you'll see a difference between the initial latency (the time spent "on the wire") and the total response time as the CPU work queue starts accumulating. This means there is more room for jitter and delay in the system, and that adds up to a "long tail" of latency.

Run a load test, and Play will happily work the CPU up to 100%. Play will give excellent throughput even at maximum capacity, handling as many requests as possible. However, Play cannot ensure that the response time of a request is within a certain range under high service utilization.

This means that out of 100 people, 95 of them will have a nice experience with a system under 100% load. However, the 5 people who are in the long tail could see delays longer than a second.

Additional Resources:

- **http://www.javidjamae.com/2005/04/07/response-time-vs-latency/**
- **https://blog.acolyer.org/2015/01/15/the-tail-at-scale/**
- **http://bravenewgeek.com/everything-you-know-about-latency-is-wrong/**
- **https://nirajrules.wordpress.com/2009/09/17/measuring-performance-response-vs-latency-vs-throughput-vs-load-vs-scalability-vs-stress-vs-robustness/**
- **https://robharrop.github.io/maths/performance/2016/02/20/service-latency-and-utilisation.html**
- **http://www.hpts.ws/papers/2007/Cockcroft_HPTS-Useless.pdf**

Want To Scale Up? Here's What We Recommend

Buy a subscription to Lightbend and get Lightbend Enterprise Suite. Lightbend Enterprise Suite includes a range of out-of-the-box features, including Application Management, Intelligent Monitoring, Enterprise Integrations, and Advanced Tooling.

The Application Management features are packaged and delivered by a tool called ConductR. Set this up, then using ConductR set up a couple of load balancers for redundancy, and then set up frontend HTTP servers as reverse proxies to the Play instances.

Lightbend Enterprise Suite also includes Intelligent Monitoring features, which provide expert instrumentation, automated discovery, visualization, and machine-driven correlation. Set this up and you'll be able to see exactly what Play is doing internally. When one of the instances hits 70% on a regular basis, then start adding capacity. This gives you lots of breathing room in the event of peak load and ensures consistent latency.

Where Next?

Play's roadmap for 3.0 is all about developer experience. The richer programming constructs available in JDK 1.8 and Scala 2.12 have opened up the opportunity to provide a richer API that uses lambdas in the Java API and can make better use of implicits in the Scala API to pass context through asynchronous boundaries. In addition, the Play team will continue to expand outreach for Play, ensuring a smooth experience for developers transitioning off legacy systems.

New To Play Framework?

Download everything you need to start a new Play project in Java or Scala on Lightbend Tech Hub:

RAPID PROJECT STARTER

Using Play Framework In Your Team?

Check out Lightbend Enterprise Suite, a set of commercial add-ons for building, deploying, monitoring and managing your Reactive applications:

LEARN ABOUT ENTERPRISE SUITE

Lightbend

Lightbend (Twitter: @Lightbend) provides the leading Reactive application development platform for building distributed systems. Based on a message-driven runtime, these distributed systems, which include microservices and fast data applications, can scale effortlessly on multi-core and cloud computing architectures. Many of the most admired brands around the globe are transforming their businesses with our platform, engaging billions of users every day through software that is changing the world.

Lightbend, Inc. 625 Market Street, 10th Floor, San Francisco, CA 94105 | www.lightbend.com