Akka Http with Websockets and Reactive Streams

Ordina (http://www.ordina.com) Source (https://github.com/J-Technologies/akka-http-websocket-activator-template#master) July 3, 2015 akka (/activator/templates#filter:akka) akka-persistence (/activator/templates#filter:akka-persistence) akka-http (/activator/templates#filter:akka-http) reactive-streams (/activator/templates#filter:reactive-streams) streams (/activator/templates#filter:streams) websocket (/activator/templates#filter:websocket) websockets (/activator/templates#filter:websockets) reactive-platform (/activator/templates#filter:reactive-platform) scala (/activator/templates#filter:scala)

An Activator template to show how Akka Http can be used with Websockets, Akka Persistence and Reactive Streams. This template consists of two parts, a theoretical and a hands-on part. In the theoretical part you will learn about the techniques mentioned above by means of a fully working example. In the hands-on part you will test-drive your own flow.

How to get "Akka Http with Websockets and Reactive Streams" on your computer

There are several ways to get this template.

Option 1: Choose `akka-http-websocket-reactive-streams` in the Lightbend Activator UI.

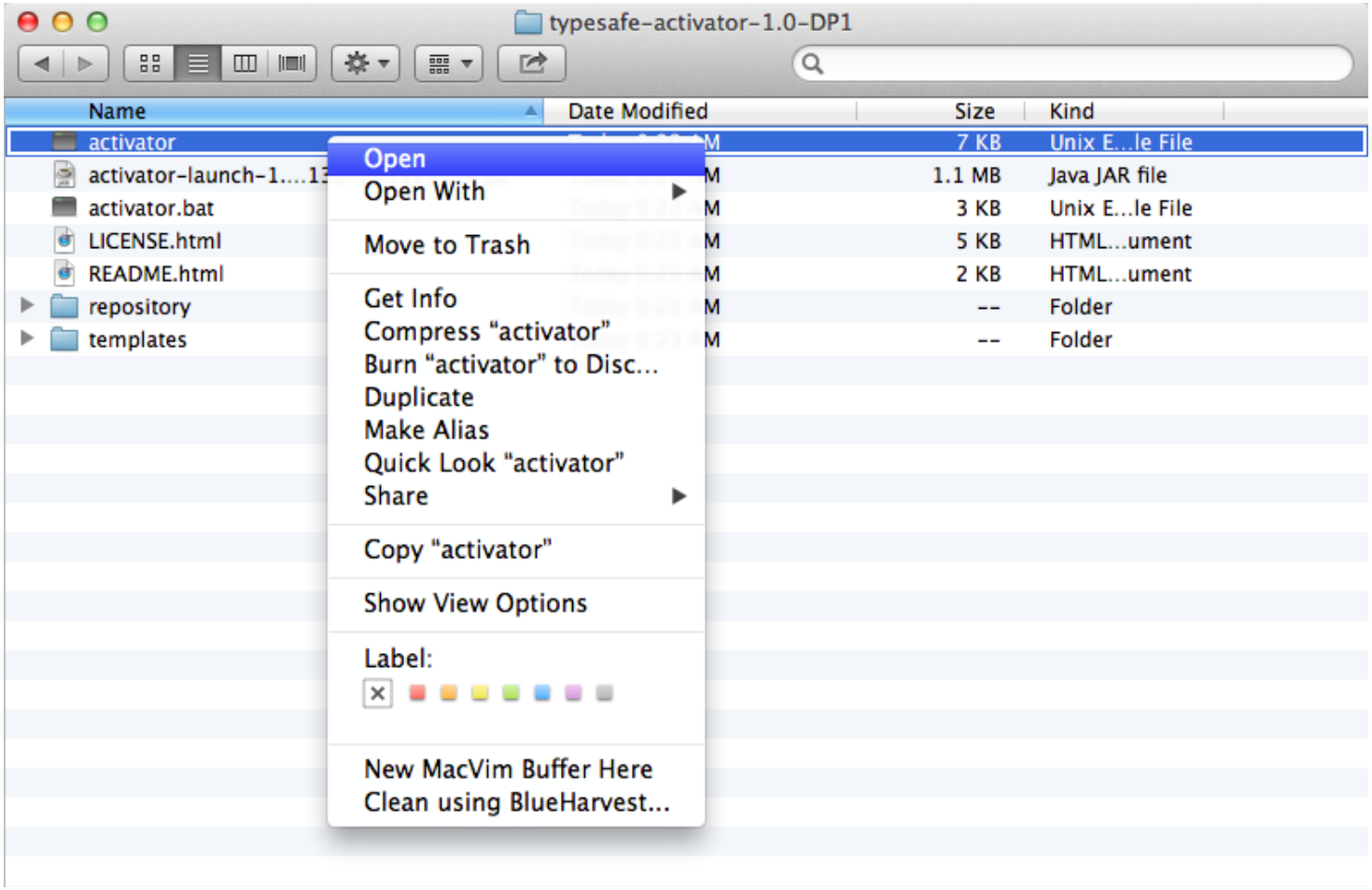
Already have Lightbend Activator (get it here (/activator/download))? Launch the UI then search for **`akka-http-websocket-reactive-streams`** in the list of templates.

Option 2: Download the `akka-http-websocket-reactive-streams` project as a zip archive

If you haven't installed Activator, you can get the code by downloading the template bundle for **`akka-http-websocket-reactive-streams`**.

- Download the Template Bundle for "Akka Http with Websockets and Reactive Streams" (/activator/template/bundle/akka-http-websocket-reactive-streams)
- Extract the downloaded zip file to your system
- The bundle includes a small bootstrap script that can start Activator. To start Lightbend Activator's UI:

In Finder, navigate to the directory that the template was extracted to, right-click on the file named "activator", then select "Open", and if prompted with a warning, click to continue:



Or from a command line:

```
typesafe:~/akka-http-websocket-reactive-streams$ ./activator ui
```

This will start Lightbend Activator and open this template in your browser.

Option 3: Create a `akka-http-websocket-reactive-streams` project from the command line

If you have Lightbend Activator, use its command line mode to create a new project from this template. Type **`activator new PROJECTNAME akka-http-websocket-reactive-streams`** on the command line.

Option 4: View the template source

The creator of this template maintains it at https://github.com/J-Technologies/akka-http-websocket-activator-template#master (https://github.com/J-Technologies/akka-http-websocket-activator-template#master).

Option 5: Preview the tutorial below

We've included the text of this template's tutorial below, but it may work better if you view it inside Activator on your computer. Activator tutorials are often designed to be interactive.

Preview the tutorial

Getting started with this template

In this template you will learn some of the basics of

- **Akka Http**
- **Akka Persistence**
- **Reactive Streams**
- **Websockets**

This template consists [illegible] mentioned above by means of a fully working example. In the hands-on part you will test-drive your own flow. So don't mind the failing Activator tests for now, you will fix them soon enough ;)

Next: Activator An Overview

Part 1: An Overview

In this tutorial we will use a simple Twitter like application to demonstrate the different techniques. You can post a tweet as a user and monitor a reactive timeline that changes whenever a new tweet is posted. We simplified the functionality a bit to keep the focus on the essentials.

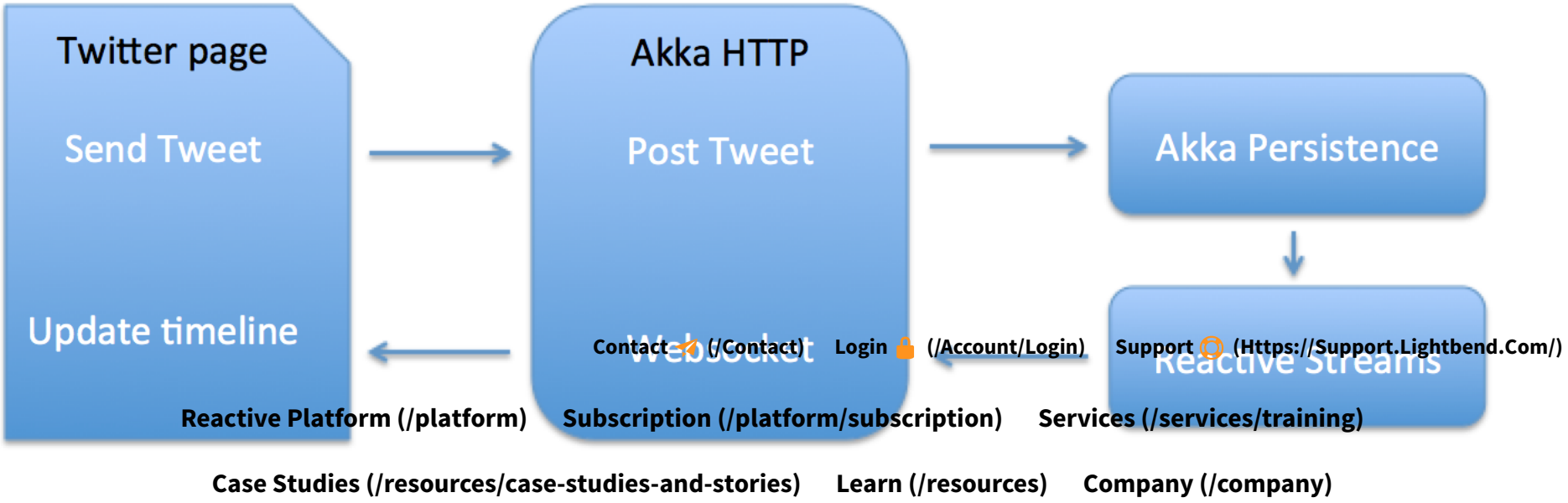
Posting tweets

You can post tweets via a simple REST endpoint. A tweet consists of a username and the message. We use **Akka Persistence** to keep track of the tweets for each user and be able to recover the timeline for a user after a restart of the application. All incoming tweets are put on the event bus that comes with **Akka**, after they are persisted.

Streaming tweets via Websockets

On the other side an actor is subscribed to the event bus. This means the actor will be notified every time a new tweet is put on the event bus. Since **Akka** implements the Reactive Streams (http://www.reactive-streams.org/announce-1.0.0) SPI we can turn the actor into a **Publisher** which we can then turn into a **Source** so we can use it in an **Akka Http Flow**.

The same story but now in a nice diagram



The complete flow from posting a tweet to it being pushed on the websocket
Next: Activator Posting a Tweet

Part 1: Posting a Tweet

We start in Main.scala which defines the routes. The route (in this case a REST endpoint) for posting a tweet is defined in the following method:

```
def addTweet = {
  post {
    entity(as[Tweet]) { tweet =>
      complete {
        (system.actorOf(TweetActorManager.props) ? tweet).map(_ => StatusCodes.NoContent)
      }
    }
  }
}
```

The interesting part resides within the `complete` directive. There we create an actor (TweetActorManager) and send it the tweet. For simplicity we ignore failure and transform a successful completion to an appropriate HTTP status code.

Next: Activator Using Akka Persistence

Part 1: Using Akka Persistence

Forwarding tweets to persistent actors

All posted tweets end up at the TweetActorManager. This actor forwards the tweet to a persistent actor based on the user that posted the tweet. Because we want to keep track of the last tweets of a user we associate each user with a dedicated persistent actor. The job of the TweetActorManager is to ensure the tweets end up at the correct persistent actor.

Persisting and recovering tweets

It is the job of the TweetActor to persist the tweet and put it on the event bus. It also knows how to recover the tweets for a user after a system failure.

```
case tweet: Tweet =>
  persist(tweet) { event =>
    sender() ! Status.Success
    context.system.eventStream.publish(tweet)
  }
```

After successful storage the callback is triggered. We send out a `Success` message and publish the tweet on the event bus.

Next: Activator Reactive Streams

Part 1: Reactive Streams

If we want to stream the tweets via websockets to our users we need to get them from somewhere. That place is the event bus. However, we need to perform some transformations before **Akka Http** is able to use the bus as a source. The first step is hook up an actor to the event bus and expose that actor as a `Publisher`.

The Reactive Streams Specification (https://github.com/reactive-streams/reactive-streams-jvm/blob/v1.0.0/README.md#specification) defines a Publisher as

> *a provider of a potentially unbounded number of sequenced elements, publishing them according to the demand received from its Subscriber(s).*

Akka implements the Reactive Streams SPI and fortunately the code to turn an actor into a Publisher is pretty straightforward: we just have to extend `ActorPublisher` which exposes (among others) the `onNext` which we can use to push a new tweet to our subscribers when one arrives. Only thing left is to subscribe to the event bus when the actor is started. The code can be found in TweetPublisher.scala and should look like this:

```
class TweetPublisher extends ActorPublisher[Tweet] {

  override def preStart = {
    context.system.eventStream.subscribe(self, classOf[Tweet])
  }

  override def receive = {
    case tweet: Tweet => onNext(tweet)
      //We do not send tweets if a client is not reading from the stream fast enough.
      if (isActive && totalDemand > 0)
        onNext(tweet)
  }

}
```

note: You should not call onNext if the stream is not active or there is no demand.

Next: Activator Running the Flow

Part 1: Running the Flow

In the previous section we described how to define an actor as a Reactive Streams Publisher. To start streaming we still need to instantiate the actor and - in one go - transform it into a Source. This is done in TweetFlow.scala.

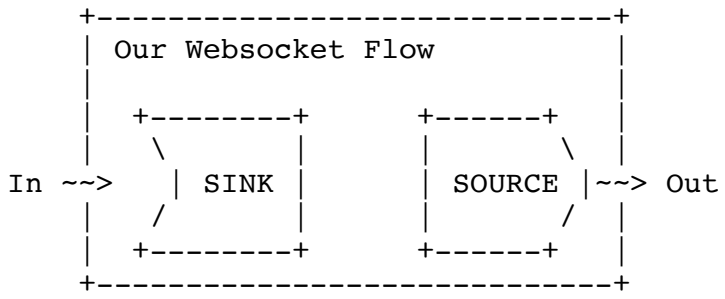
```
private val tweetSource: Source[Tweet, ActorRef] =
        Source.actorPublisher[Tweet](TweetPublisher.props)
```

Now we've got our hands on a proper `Source` we can use it to construct a `Flow`:

```
def tweetFlow(tweetFilter: TweetFilter): Flow[Message, Message, Unit] =
            Flow.wrap(Sink.ignore, tweetSource filter tweetFilter map toMessage)(Keep.none)
```

Note: ignore the tweetFilter for now, this is just a convenience parameter to be able to construct specific streams, like the stream of tweets for a specific user, or the stream of tweets that contain a specific hash tag.

We are only interested in one way communication with our Websocket. We push new tweets to the client on their arrival. Theoretically we could also receive messages over the same channel from the client. We just ignore those. So what does this look like in terms of a Flow?



The important thing to note is that the inner parts of the Flow (the Sink and the Source) have no connection with each other. Incoming messages from the Websocket are ignored by routing them to the Ignore Sink while on the other hand the outgoing channel will be serviced by our transformed actor that is listening to the event bus:

```
Flow.wrap(Sink.ignore, tweetSource filter tweetFilter map toMessage)(Keep.none)
```

Next: Activator The Websocket

Part 1: The Websocket

So we have a nice Flow, the next step is plugging it into Akka Http.
The most notable part is the `handleWebsocketMessages` directive, which comes out of the box with Akka Http.

```
def tweetsOfUserSocket = (pathPrefix("users") & path(Segment)) { userName =>
  handleWebsocketMessages(tweetFlowOfUser(userName))
}
```

The part below describes the path at which the Websocket for this Flow can be found: `/ws/tweets/users/[userName]`.

```
pathPrefix("ws") {
  pathPrefix("tweets") {
    get {
      tweetsOfUserSocket
    }
  }
}
```

Now we have explained all the pieces you can Activator run the application and see everything in action.

In the next part you will extend the application with a flow containing only tweets with a certain hash tag.

Next: Activator Building your own Flow

Part 2: Building your own Flow

Implementing a Websocket for tweets with a certain hash tag

It's time to get your hands dirty. Tweets for a user are nice to have, but having a different channel to only follow tweets with a certain hash tag would be even nicer, don't you agree?

In this second part we will create this feature step by step. Each step will have a failing test and an unimplemented method to help you on the way.

Next: Activator Creating a New Flow

Part 2: Creating a New Flow

Before we can expose a stream of filtered tweets on hash tag to the outside world, we'll first need something to actually expose. This basically comes down to the fact that we need an extra flow, besides the flow for all tweets and the flow for tweets of a user. Let's start by looking at the failing tests in TweetFlowSpec.scala.

```
"The flow for tweets with hash tag" should "only forward tweets with matching hash tag" in {
  val hashTag = "shouldMatch"
  val sut = tweetFlowWithHashTag(hashTag).runWith(TestSource.probe[Message], TestSink.probe[Message])
  val (_, mockSink) = sut

  val tweet = Tweet(User("test"), s"Hello World! #${hashTag}")
  system.eventStream.publish(tweet)

  mockSink.request(1)
  mockSink.expectNext()
  mockSink.expectNoMsg(noMessageTimeout)
}
```

We mock out the `Source` and `Sink` to be able to control the input and output for a flow, which makes testing a lot easier. We expect, given a flow that filters on hash tag X and putting a tweet on the event bus with that same tag X, that we will actually receive that tweet in our mock sink. Because we only put one tweet on the bus, we expect no more message after our first request.

There is another test to test the opposite scenario (putting a tweet on the bus with a non matching tag and expecting not to see it arrive), but we'll not discuss since it is analogous to the test described above. Open TweetFlow.scala and add an implementation on the spot marked with `???`. Of course your implementation should satisfy both failing tests in TweetFlowSpec.scala.

Next: Activator The Solution for this challenge (spoiler alert!)

Part 2: Solution New Flow

Drumroll...

`def tweetFlowWithHashTag(hashTag: String) = tweetFlow(_.text contains hashTag)`
Your method should look something like this. Yes, it was that easy. Sorry.

Now we have a `Flow` we can expose it via a new Websocket route.

Next: Activator Hash Tag Websocket

Part 2: Hash Tag Websocket

Let's take a look at the failing test in MainRoutingSpec.scala first.

```
it should "handle websocket requests for hash tags" in {
  Get("/ws/tweets/hashtag/test") ~> Upgrade(List(UpgradeProtocol("websocket"))) ~> emulateHttpCore ~> Main.mainFlow ~>
check {
    status shouldEqual SwitchingProtocols
  }
}
```

The test should make clear on which path it expects the stream to be exposed. You can implement the functionality in Main.scala. Look for the `???` and comments with `TODO`.

Next: Activator The Solution for this challenge (spoiler alert!)

Part 2: Solution Websocket

```
def tweetsWithHashTagSocket = (pathPrefix("hashtag") & path(Segment)) { hashTag =>
  handleWebsocketMessages(tweetFlowWithHashTag(hashTag))
}
```

We can then use this method in our main routing declaration:

```
// Websocket endpoints
pathPrefix("ws") {
  pathPrefix("tweets") {
    get {
      allTweetsSocket ~
      tweetsOfUserSocket ~
      tweetsWithHashTagSocket
    }
  }
}
```

Et voilà! We just extended our application with a nice flow to monitor tweet streams for certain hash tags. And all that with only a few lines of extra code.
Activator Run the application and enjoy the fruits of your hard labour.

Next: Activator Time to wrap it all up

Conclusion

In this Activator template we explored the following techniques:

- **Akka Http**
- **Akka Persistence**
- **Reactive Streams**
- **Websockets**

We hope you enjoyed this tutorial. Feedback and suggestions are always appreciated.
So if you want to get in touch contact us @ Github (https://github.com/J-Technologies/akka-http-websocket-activator-template).

Next: Activator Not tired yet?

Further Explorations

So, still not tired? Try implementing some (or all) of the following functionality:

- **Exposing a combined stream of tweets for a user with a certain hash tag**
- **Distribute the application using Akka Cluster**
- **Perform real time analytics with Spark**
- **[Insert your own creative insight here]**

LIGHTBEND ACTIVATOR

Overview (/community/core-tools/activator-and-sbt)

Download Activator (/activator/download)

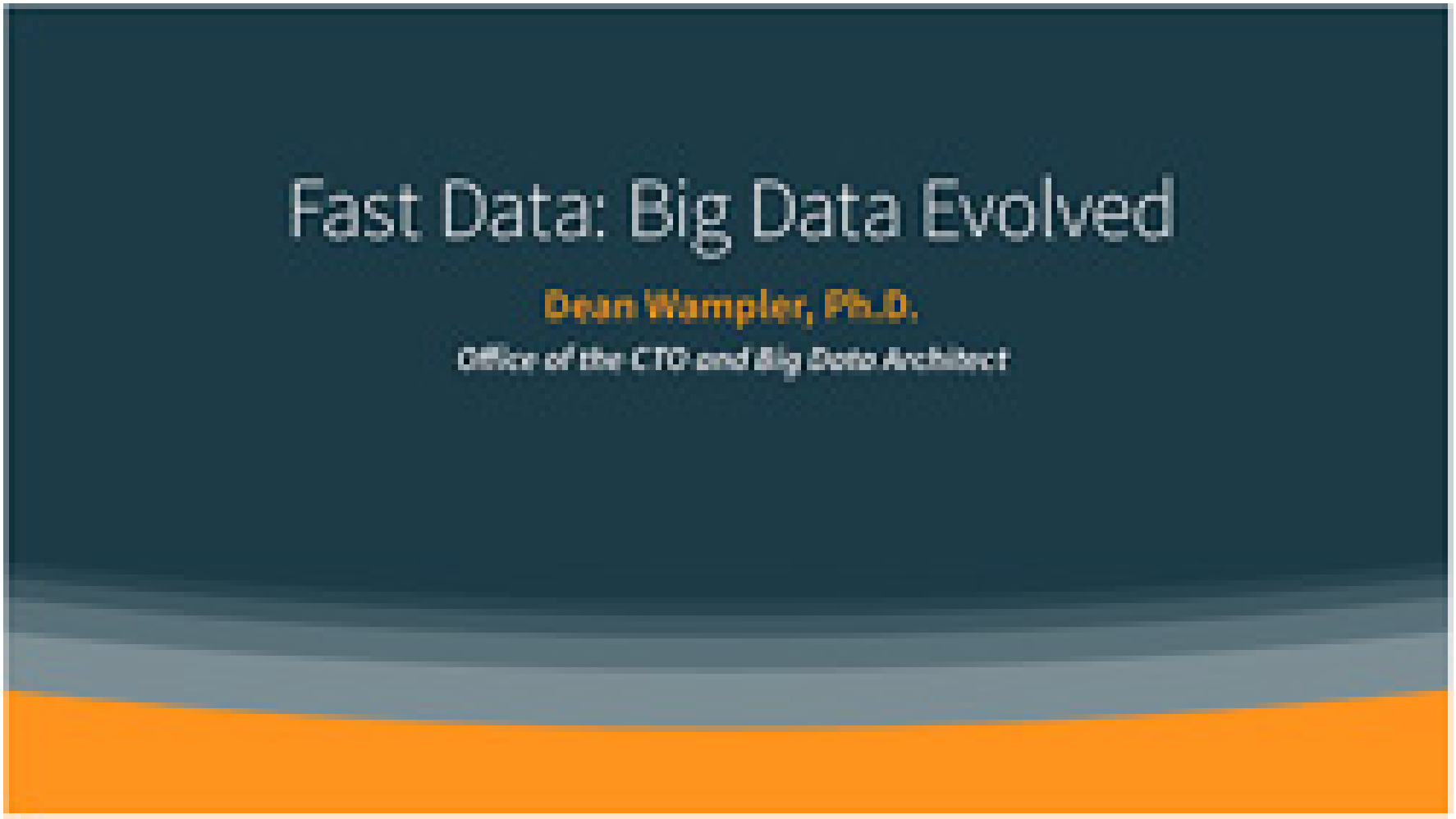
Documentation (/activator/docs)

Browse Templates (/activator/templates)

Contribute Template (/activator/template/contribute)

WHITE PAPER

Fast Data: Big Data Evolved

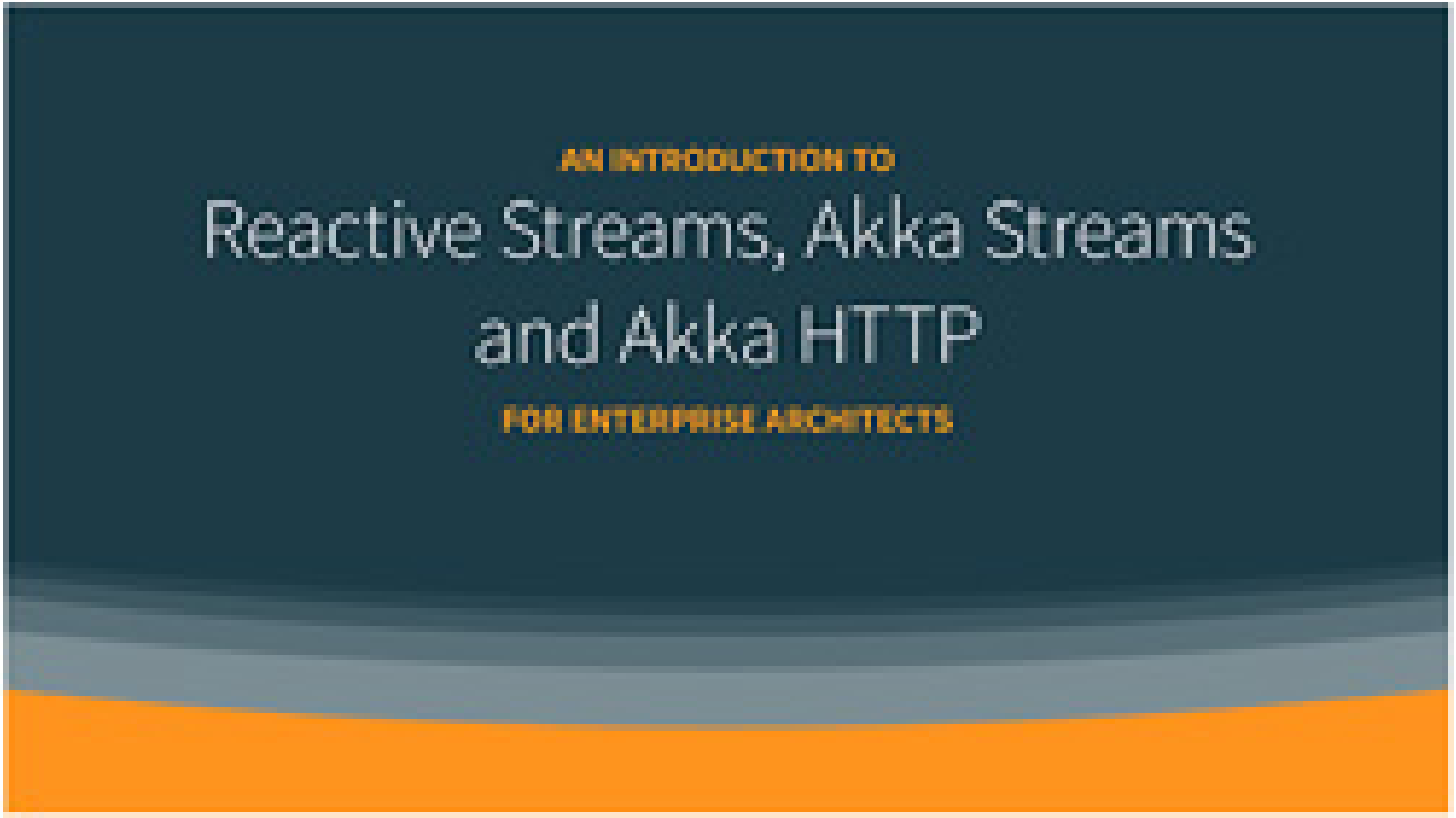


(https://info.lightbend.com/COLL-20XX-Fast-Data-Big-Data-Evolved-WP_LP.html?lst=WS&lsd=COLL-20XX-Fast-Data-Big-Data-Evolved-WP)

GET WHITE PAPER (HTTPS://INFO.LIGHTBEND.COM/COLL-20XX-FAST-DATA-BIG-DATA-EVOLVED-WP_LP.HTML?LST=WS&LSD=COLL-20XX-FAST-DATA-BIG-DATA-EVOLVED-WP)

WHITE PAPER

An Introduction to Reactive Streams, Akka Streams and Akka HTTP



(https://info.lightbend.com/COLL-20XX-Enterprise-Architect-Akka-Streaming-Guide_LP.html?lst=WS&lsd=COLL-20XX-Enterprise-Architect-Akka-Streaming-Guide)

GET WHITE PAPER (HTTPS://INFO.LIGHTBEND.COM/COLL-20XX-ENTERPRISE-ARCHITECT-AKKA-STREAMING-GUIDE_LP.HTML?LST=WS&LSD=COLL-20XX-ENTERPRISE-ARCHITECT-AKKA-STREAMING-GUIDE)

REACTIVE PLATFORM (/PLATFORM)

For Development (/platform/development)

- Akka (/platform/development/akka)
- Play Framework (/platform/development/play-framework)
- Lagom Framework (/platform/development/lagom-framework)
- Apache Spark (/platform/development/spark)
- Scala and Java (/platform/development/scala-and-java)

For Production (/platform/production)

- Service Orchestration (/platform/production/service-orchestration)
- Application Monitoring (/platform/production/application-monitoring)
- Application Resilience (/platform/production/application-resilience)
- Enhanced Availability (/platform/production/enhanced-availability)

For Business (/platform/enterprise)

SUBSCRIPTION (/PLATFORM/SUBSCRIPTION)

Overview (/platform/subscription)

SERVICES (/SERVICES/EXPERT-SUPPORT)

Training (/services/training)

Consulting (/services/consulting)

CASE STUDIES (/RESOURCES/CASE-STUDIES-AND-STORIES)

Case Studies & Stories (/resources/case-studies-and-stories)

SWAG (HTTP://SHOP.LIGHTBEND.COM)

Online Store (http://shop.lightbend.com)

ACTIVATOR (/ACTIVATOR/DOWNLOAD)

Download Activator (/activator/download)

Activator Templates (/activator/templates)

BLOG (/BLOG)

Blog (/blog)

Blog RSS (/blog/rss.xml)

LEARN (/RESOURCES)

Blog (/blog)

eBooks (/resources/e-books)

Podcasts (https://soundcloud.com/lightbend)

Training (/services/training)

Industry Analysts (/industry-analysts)

White Papers & Reports (/resources/collateral)

Videos & Webinars (/resources/videos#filter:webinar)

Upcoming Events (/community)

COMPANY (/COMPANY)

About (/company)

Leadership (/company/leadership)

News & Press Releases (/company/news)

Careers with Lightbend (/company/careers)

Careers with Customers (/customers/our-customers-are-hiring)

Contact (/contact)

Open Source Position Statement (/open-source-position-statement)

PARTNERS (/PARTNERS/OVERVIEW)

Overview (/partners/overview)

Resellers (/partners/resellerpartners)

Technology (/partners/technology-partners)

System Integrators (/partners/systemintegrator-partners)

Global System Integrators (/partners/globalsystemintegrator-partners)

Training (/partners/training-partners)

Become a Partner (/partners/become-a-partner)

LIGHTBEND ACCOUNT (/ACCOUNT)

Login (/account/login)

Sign up (/account/register)

CUSTOMER PORTAL (HTTPS://PORTAL.LIGHTBEND.COM/ACCOUNT/LOGIN)

Login (https://portal.lightbend.com/account/login)

SUPPORT (HTTPS://SUPPORT.LIGHTBEND.COM/)

Support Login (https://support.lightbend.com/)

ACTIVATOR (/ACTIVATOR/DOWNLOAD)

Download Activator (/activator/download)

Activator Templates (/activator/templates)

BLOG (/BLOG)

Blog (/blog)

Blog RSS (/blog/rss.xml)

LIGHTBEND ACCOUNT (/ACCOUNT)

Login (/account/login)

Sign up (/account/register)

CUSTOMER PORTAL (HTTPS://PORTAL.LIGHTBEND.COM/ACCOUNT/LOGIN)

Login (https://portal.lightbend.com/account/login)

SUPPORT (HTTPS://SUPPORT.LIGHTBEND.COM/)

Support Login (https://support.lightbend.com/)

Follow

(http://www.facebook.com/typesafe) (http://twitter.com/intent/follow?source=followbutton&variant=1.0&screen_name=lightbend)

(https://plus.google.com/+Typesafe/videos) (http://www.linkedin.com/company/typesafe)

(http://www.youtube.com/channel/UCmbAzITOKm4KAhV8FBECfMA?sub_confirmation=1) (https://www.typesafe.com/blog/rss.xml)