# Spark Job Server

Evan Chan and Kelvin Chu

*Date*

# Overview

# Why We Needed a Job Server

- Created at Ooyala in 2013
- Our vision for Spark is as a multi-team big data service
- What gets repeated by every team:
  - Bastion box for running Hadoop/Spark jobs
    - Deploys and process monitoring
  - Tracking and serializing job status, progress, and job results
  - Job validation

# Spark as a Service

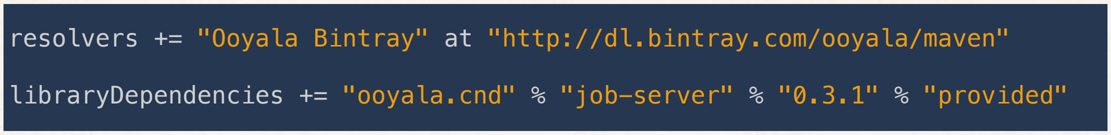
- REST API for Spark jobs and contexts.  Easily operate Spark from any language or environment.
- Runs jobs in their own Contexts or share 1 context amongst jobs
- Great for sharing cached RDDs across jobs and low-latency jobs
- Works for Spark Streaming as well!
- Works with Standalone, Mesos, any Spark config
- Jars, job history and config are persisted via a pluggable API
- Async and sync API, JSON job results

*Open Source!!*

http://github.com/ooyala/spark-jobserver

# Creating a Job Server Project

- In your build.sbt, add this



```
resolvers += "Ooyala Bintray" at "http://dl.bintray.com/ooyala/maven"

libraryDependencies += "ooyala.cnd" % "job-server" % "0.3.1" % "provided"
```

- sbt assembly -> fat jar -> upload to job server
- "provided" is used. Don’t want SBT assembly to include the whole job server jar.
- Java projects should be possible too

# Example Job Server Job

```
/**
* A super-simple Spark job example that implements the SparkJob trait and
* can be submitted to the job server.
*/
object WordCountExample extends SparkJob {
 override def validate(sc: SparkContext, config: Config): SparkJobValidation = {
   Try(config.getString(“input.string”))
     .map(x => SparkJobValid)
     .getOrElse(SparkJobInvalid(“No input.string”))
 }

 override def runJob(sc: SparkContext, config: Config): Any = {
   val dd = sc.parallelize(config.getString(“input.string”).split(" ").toSeq)
   dd.map((_, 1)).reduceByKey(_ + _).collect().toMap
 }
}
```

# What’s Different?

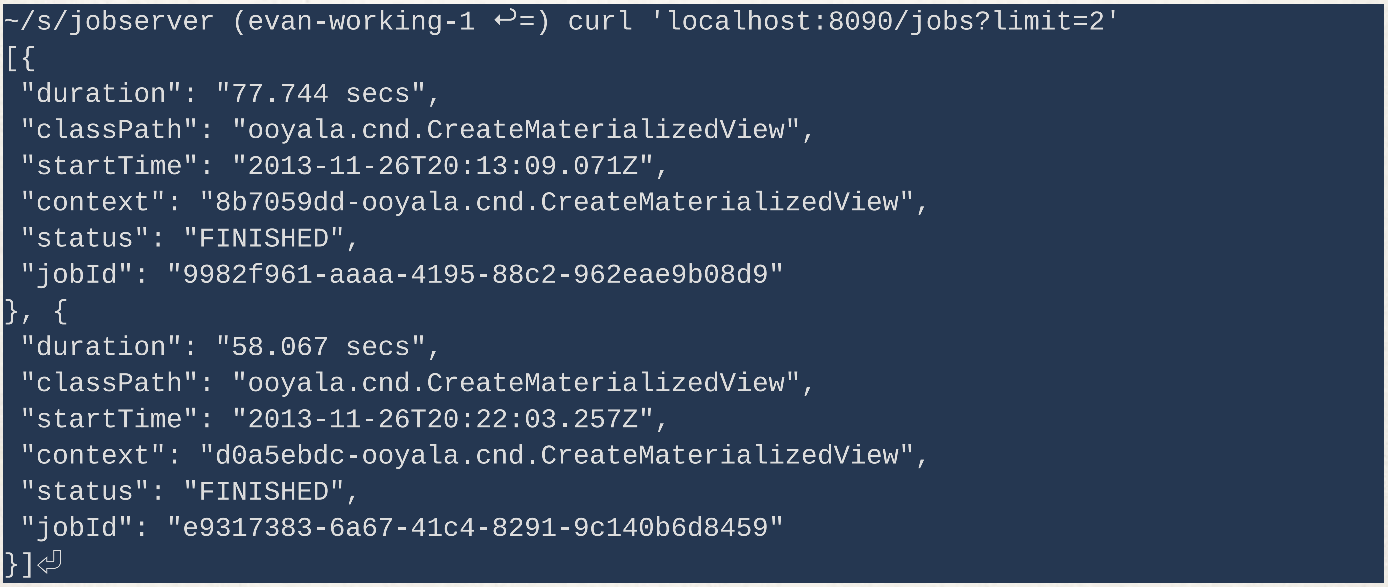
- Job does not create Context, Job Server does
- Decide when I run the job: in own context, or in pre-created context
- Upload new jobs to diagnose your RDD issues:
  - POST /contexts/newContext
  - POST /jobs .... context=newContext
  - Upload a new diagnostic jar... POST /jars/newDiag
  - Run diagnostic jar to dump into on cached RDDs

# Submitting and Running a Job

```
✦  curl --data-binary @../target/mydemo.jar localhost:8090/jars/demo
OK[11:32 PM] ~

✦  curl -d "input.string = A lazy dog jumped mean dog" 'localhost:8090/jobs?
appName=demo&classPath=WordCountExample&sync=true'
{
 "status": "OK",
 "RESULT": {
   "lazy": 1,
   "jumped": 1,
   "A": 1,
   "mean": 1,
   "dog": 2
 }
}
```

# Retrieve Job Statuses



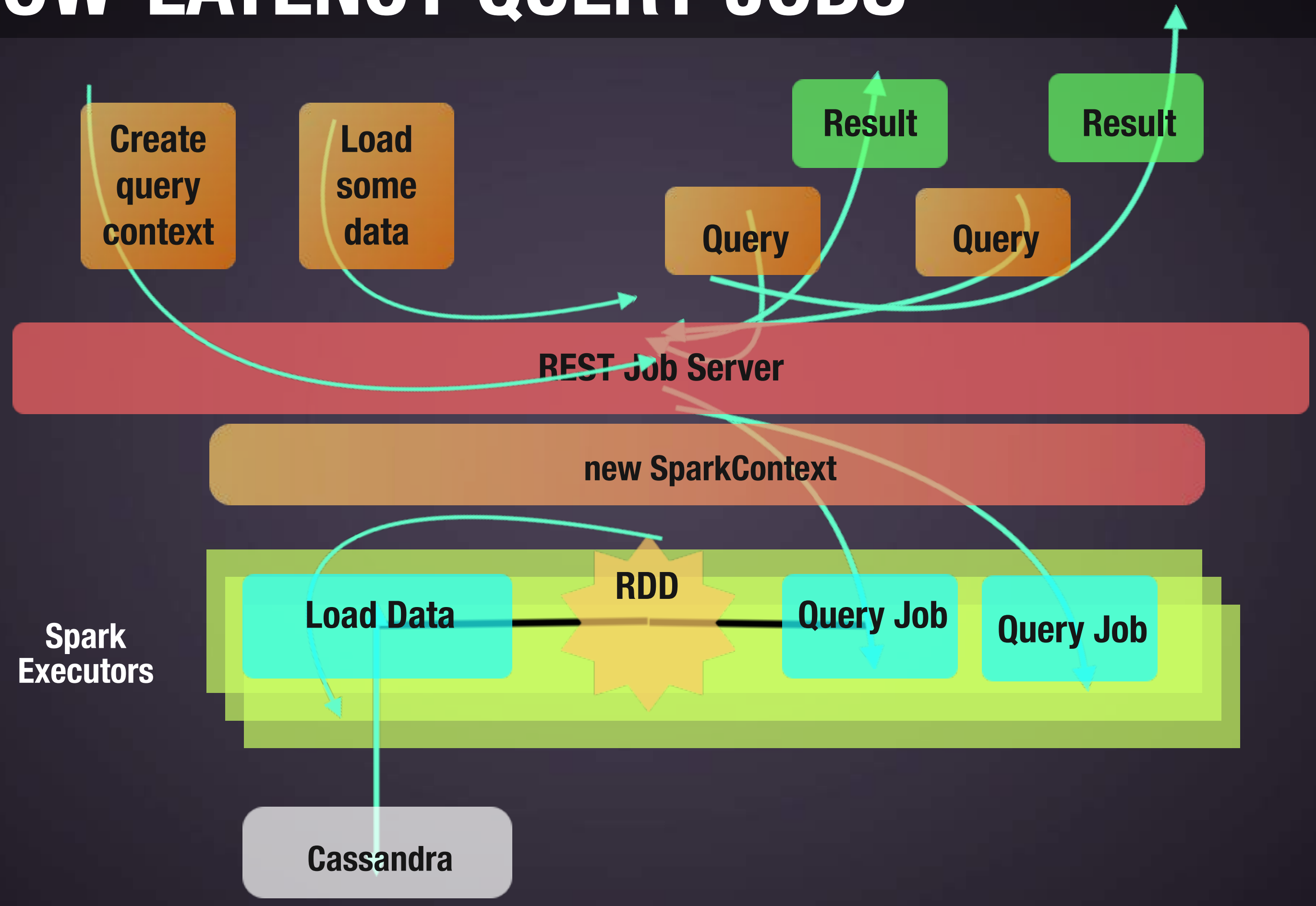
```
~/s/jobserver (evan-working-1 ↩=) curl 'localhost:8090/jobs?limit=2'
[{
 "duration": "77.744 secs",
 "classPath": "ooyala.cnd.CreateMaterializedView",
 "startTime": "2013-11-26T20:13:09.071Z",
 "context": "8b7059dd-ooyala.cnd.CreateMaterializedView",
 "status": "FINISHED",
 "jobId": "9982f961-aaaa-4195-88c2-962eae9b08d9"
}, {
 "duration": "58.067 secs",
 "classPath": "ooyala.cnd.CreateMaterializedView",
 "startTime": "2013-11-26T20:22:03.257Z",
 "context": "d0a5ebdc-ooyala.cnd.CreateMaterializedView",
 "status": "FINISHED",
 "jobId": "e9317383-6a67-41c4-8291-9c140b6d8459"
}]⏎
```

# Use Case: Fast Query Jobs

# Spark as a Query Engine

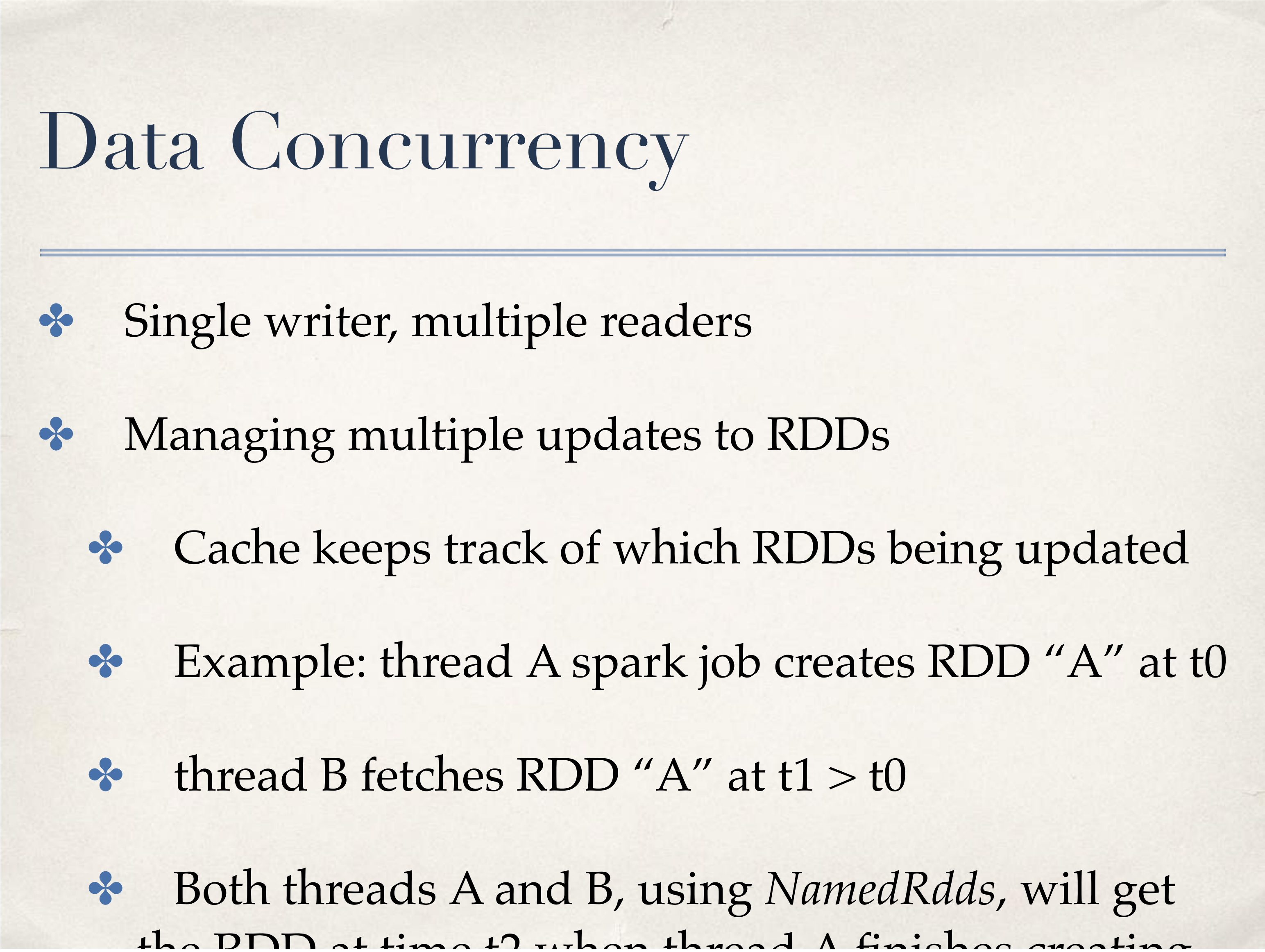
- Goal: spark jobs that run in *under a second* and answers queries on shared RDD data
- Query params passed in as job config
- Need to minimize context creation overhead
  - Thus many jobs sharing the same SparkContext
- On-heap RDD caching means no serialization loss
- Need to consider concurrent jobs (fair scheduling)

# LOW-LATENCY QUERY JOBS



# Sharing Data Between Jobs

- RDD Caching
  - Benefit: no need to serialize data.  Especially useful for indexes etc.
  - Job server provides a *NamedRdds* trait for thread-safe CRUD of cached RDDs by name
    - (Compare to SparkContext's API which uses an integer ID and is not thread safe)



# Data Concurrency

- Single writer, multiple readers
- Managing multiple updates to RDDs
  - Cache keeps track of which RDDs being updated
  - Example: thread A spark job creates RDD “A” at t0
  - thread B fetches RDD “A” at t1 > t0
  - Both threads A and B, using *NamedRdds*, will get the RDD at time t2 when thread A finishes creating

# Production Usage

# Persistence

- What gets persisted?
  - Job status (success, error, why it failed)
  - Job Configuration
  - Jars
- JDBC database configuration: spark.sqldao.jdbc.url
  - `jdbc:mysql://dbserver:3306/jobserverdb`

# Deployment and Metrics

- spark-jobserver repo comes with a full suite of tests and deploy scripts:
  - server_deploy.sh for regular server pushes
  - server_package.sh for Mesos and Chronos .tar.gz
- /metricz route for codahale-metrics monitoring
- /healthz route for health check0o

# Challenges and Lessons

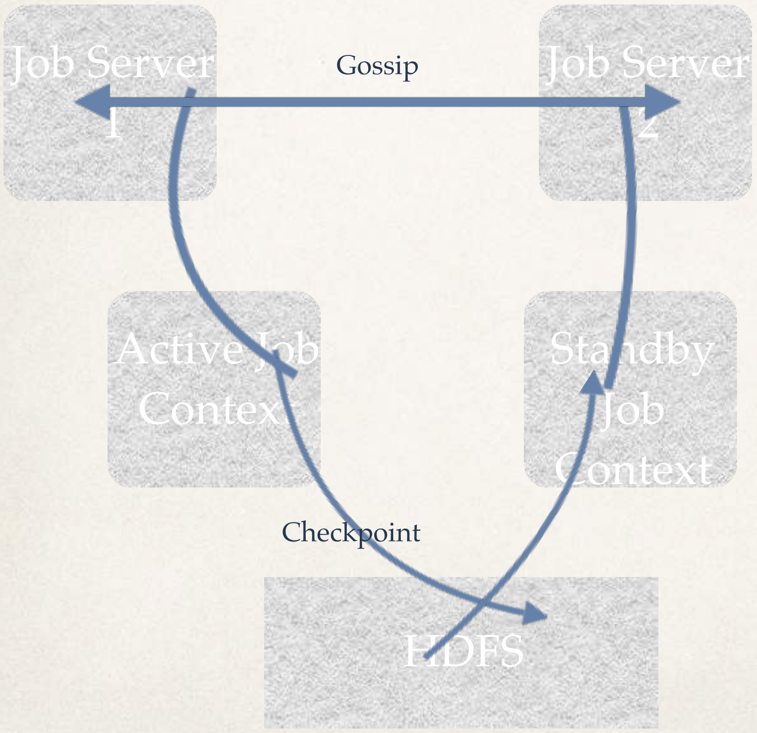
- Spark is based around contexts - we need a Job Server oriented around logical jobs
- Running multiple SparkContexts in the same process
  - Global use of System properties makes it impossible to start multiple contexts at same time (but see pull request...)
  - Have to be careful with SparkEnv
- Dynamic jar and class loading is tricky
- Manage threads carefully - each context uses lots of threads

# Future Work

# Future Plans

- Spark-contrib project list. So this and other projects can gain visibility! (SPARK-1283)
- HA mode using Akka Cluster or Mesos
- HA and Hot Failover for Spark Drivers/Contexts
- REST API for job progress
- Swagger API documentation

# HA and Hot Failover for Jobs



- Job context dies:
  - Job server 2 notices and spins up standby context, restores checkpoint

# Thanks for your contributions!

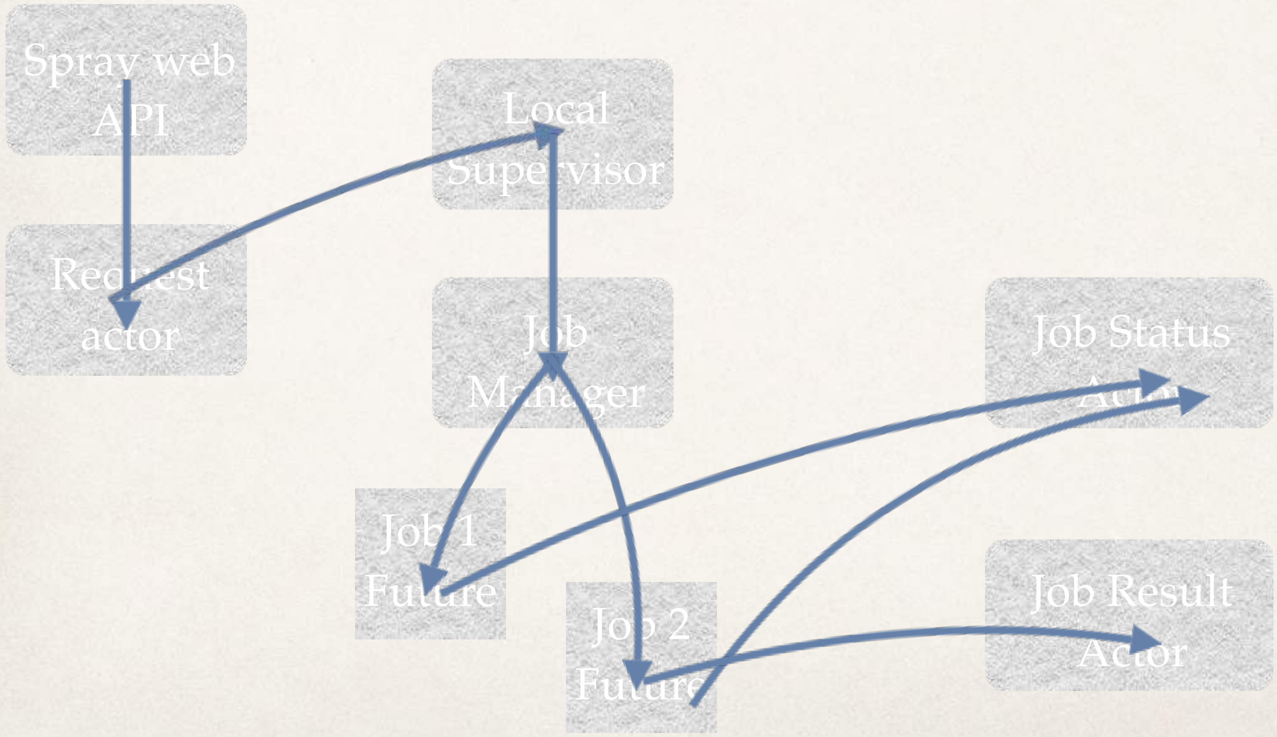
- All of these were community contributed:
  - index.html main page
  - saving and retrieving job configuration
- Your contributions are very welcome on Github!

# Architecture

# Completely Async Design

- http://spray.io - probably the fastest JVM HTTP microframework
- Akka Actor based, non blocking
- Futures used to manage individual jobs. (Note that Spark is using Scala futures to manage job stages now)
- Single JVM for now, but easy to distribute later via remote Actors / Akka Cluster

# Async Actor Flow



Spray web API

Request actor

Local Supervisor

Job Manager

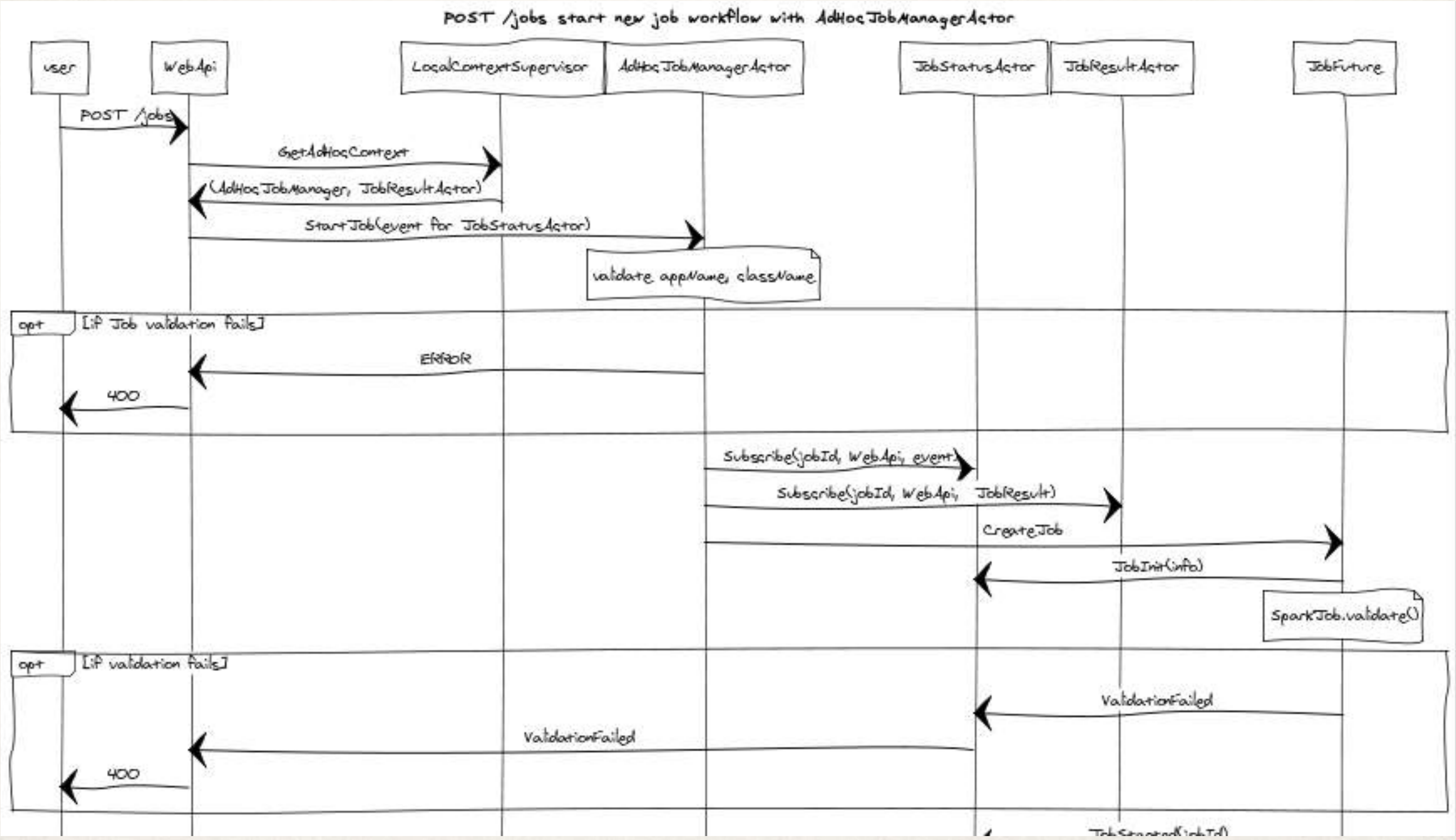
Job 1 Future

Job 2 Future

Job Status Actor

Job Result Actor

# Message flow fully documented


POST /jobs start new job workflow with AdHocJobManagerActor
user
WebApi
LocalContextSupervisor
AdHocJobManagerActor
JobStatusActor
JobResultActor
JobFuture
POST /jobs
GetAdHocContext
(AdHocJobManager, JobResultActor)
StartJob(event for JobStatusActor)
validate appName, className
opt
[if Job validation fails]
ERROR
400
Subscribe(jobId, WebApi, event)
Subscribe(jobId, WebApi, JobResult)
CreateJob
JobInit(info)
SparkJob.validate()
opt
[if validation fails]
ValidationFailed
ValidationFailed
400
JobStarted(jobId)

# Thank you!

And Everybody is Hiring!!

# Using Tachyon

| Pros | Cons |
|---|---|
| Off-heap storage: No GC | ByteBuffer API - need to pay deserialization cost |
| Can be shared across multiple processes | |
| Data can survive process loss | |
| Backed by HDFS | Does not support random access writes |
| | |