



Finatra v2

Fast, testable Scala services

Steve Cosenza @scosenza

Finatra & Data API Tech Lead @twitter

What is Finatra?

- Finatra is a framework for easily building API services on top of Twitter's Scala stack (twitter-server, finagle, twitter-util)
- Currently supports building HTTP services
- Plan to support building Thrift services

Project History

- Feb 2012: Finatra project started on Github by two Twitter engineers (Julio Capote and Chris Burnett)
- June 2014: Christopher Coco and Steve Cosenza on the Data API team release Finatra v2.0.0-SNAPSHOT internally at Twitter
- Jan 2015: Finatra v1.6 released for Scala 2.9 and 2.10
- April 2015: Finatra v2.0.0M1 publicly released for Scala 2.10 and 2.11

Highlights

- Production use as Twitter's HTTP framework
- ~50 times faster than v1.6 in several benchmarks
- Powerful feature and integration test support
- JSR-330 Dependency Injection using Google Guice
- Jackson based JSON parsing supporting required fields, default values, and custom validations
- Logback MDC integration for contextual logging across Futures
- Guice request scope integration with Futures

Twitter Inject Libraries

- inject-core
- inject-app
- inject-server
- inject-modules
- inject-thrift-client
- inject-request-scope

Finatra Libraries

- finatra-http
- finatra-jackson
- finatra-logback
- finatra-httpclient
- finatra-utils

Tutorial

Let's create a simple HTTP API for Tweets :-)

Create a FeatureTest

```
class TwitterCloneFeatureTest extends Test {  
  
  val server = new EmbeddedHttpServer(  
    twitterServer = new HttpServer {})  
  
  "Post tweet" in {  
    server.httpPost(  
      path = "/tweet",  
      postBody = ""  
        {  
          "message": "Hello #SFScala",  
          "location": {  
            "lat": "37.7821120598956",  
            "long": "-122.400612831116"  
          },  
          "sensitive": false  
        }""  
      ,  
      andExpect = Created,  
      withLocationHeader = "/tweet/123")  
  }  
}
```


And let's assert the response body

```
"Post tweet" in {
  server.httpPost(
    path = "/tweet",
    postBody = """
      {
        "message": "Hello #SFScala",
        "location": {...}
        "sensitive": false
      }"""
  ,
  andExpect = Created,
  withLocationHeader = "/tweet/123",
  withJsonBody = """
    {
      "id": "123",
      "message": "Hello #SFScala",
      "location": {
        "lat": "37.7821120598956",
        "long": "-122.400612831116"
      },
      "sensitive": false
    }"""
  )
}
```

Test Failure: Route not found

=====

```
HTTP POST /tweet
```

```
[Header] Content-Length -> 204
```

```
[Header] Content-Type -> application/json; charset=utf-8
```

```
[Header] Host -> 127.0.0.1:60102
```

```
{  
  "message" : "Hello #SFScala",  
  "location" : {  
    "lat" : "37.7821120598956",  
    "long" : "-122.400612831116"  
  },  
  "sensitive" : false  
}
```

=====

```
RoutingService Request("POST /tweet", from /127.0.0.1:60103) not found in  
registered routes:
```

```
[Status] 404 Not Found
```

```
[Header] Content-Length -> 0
```

```
*EmptyBody*
```

Create a Controller

```
package finatra.quickstart.controllers

import com.twitter.finagle.http.Request
import com.twitter.finatra.http.Controller

class TweetsController extends Controller {

  post("/tweet") { request: Request =>
    response.created("hello")
  }
}
```

Create a Server

```
import com.twitter.finatra.HttpServer
import com.twitter.finatra.filters.CommonFilters
import com.twitter.finatra.routing.HttpRouter
import finatra.quickstart.controllers.TweetsControllerV1

class TwitterCloneServer extends HttpServer {

  override def configureHttp(router: HttpRouter): Unit = {
    router
      .filter[CommonFilters]
      .add(new TweetsController)
  }
}
```

Add Server to FeatureTest

```
class TwitterCloneFeatureTest extends Test {  
    val server = new EmbeddedHttpServer(  
        twitterServer = new TwitterCloneServer)  
    ...  
}
```

Test Failure: Unexpected response body

```
=====
HTTP POST /tweet
```

```
[Header] Content-Length -> 204
```

```
[Header] Content-Type -> application/json; charset=utf-8
```

```
[Header] Host -> 127.0.0.1:60118
```

```
...
```

```
=====
127.0.0.1 - - [24/Apr/2015:21:36:25 +0000] "POST /tweet HTTP/1.1" 201 5 9 "-"
-----
```

```
[Status] 201 Created
```

```
[Header] Content-Length -> 5
```

```
hello
```

```
Unrecognized token 'hello': was expecting ('true', 'false' or 'null')
```

```
  at [Source: hello; line: 1, column: 11]
```

```
JSON DIFF FAILED!
```

```
      *
```

```
Received: hello
```

```
Expected: {"id":"123","location":{"lat":"37.7821120598956","long":"-122.4006 ...
```

Let's parse some JSON

```
{  
  "message": "Hello #SFScala",  
  "location": {  
    "lat": "37.7821120598956",  
    "long": "-122.400612831116"  
  },  
  "sensitive": false  
}
```

```
case class PostedTweet(  
  message: String,  
  location: Option[Location],  
  sensitive: Boolean = false) {  
  
  def toStatus(id: StatusId) = { ... }  
}
```

```
case class Location(  
  lat: Double,  
  long: Double)
```

Update Controller to use PostedTweet

```
class TweetsController extends Controller {  
  post("/tweet") { postedTweet: PostedTweet =>  
    response.created(postedTweet)  
  }  
}
```


Test failure: 'id' field in body missing

```
[Status] 201 Created
[Header] Content-Type -> application/json; charset=utf-8
[Header] Content-Length -> 107
{
  "message" : "Hello #SFScala",
  "location" : {
    "lat" : 37.7821120598956,
    "long" : -122.400612831116
  },
  "sensitive" : false
}
```

JSON DIFF FAILED!

*

```
Received: {"location":{"lat":37.7821120598956,"long":-122.400612831116}, ...
Expected: {"id":"123","location":{"lat":"37.7821120598956","long":"-122. ...
```

Let's generate some JSON

```
{  
  "id": 123,  
  "message": "Hello #SFScala",  
  "location": {  
    "lat": "37.7821120598956",  
    "long": "-122.400612831116"  
  },  
  "sensitive": false  
}
```

```
case class SerializedTweet(  
  id: StatusId,  
  message: String,  
  location: Option[SerializedLocation],  
  sensitive: Boolean)
```

```
case class StatusId(  
  id: String)  
extends WrappedValue[String]
```

```
case class SerializedLocation(  
  lat: String,  
  long: String)
```

Update Controller to use RenderableTweet

```
class TweetsController extends Controller {  
  post("/tweet") { postedTweet: PostedTweet =>  
    val statusId = StatusId("123")  
    val status = postedTweet.toDomain(statusId)  
  
    // Save status here  
  
    val renderableTweet = RenderableTweet.fromDomain(status)  
    response  
      .created(renderableTweet)  
      .location(renderableTweet.id)  
  }  
}
```

Feature Test Success!

```
=====
HTTP POST /tweet
```

```
...
```

```
=====
127.0.0.1 - - [24/Apr/2015:22:38:31 +0000] "POST /tweet HTTP/1.1" 201 122 642 "-"
-----
```

```
[Status] 201 Created
```

```
[Header] Content-Type -> application/json; charset=utf-8
```

```
[Header] Location -> http://127.0.0.1:60352/tweet/123
```

```
[Header] Content-Length -> 122
```

```
{
  "id" : "123",
  "message" : "Hello #SFScala",
  "location" : {
    "lat" : "37.7821120598956",
    "long" : "-122.400612831116"
  },
  "sensitive" : false
}
```

Let's test that 'message' is required

```
"Missing message field" in {  
  server.httpPost(  
    path = "/tweet",  
    postBody = ""  
    {  
      "location": {  
        "lat": "37.7821120598956",  
        "long": "-122.400612831116"  
      },  
      "sensitive": false  
    }""  
    ,  
    andExpect = BadRequest)  
}
```

Test Success

```
=====
127.0.0.1 - - [24/Apr/2015:22:57:56 +0000] "POST /tweet HTTP/1.1" 400 42 570 "-"
-----
```

```
[Status] 400 Bad Request
```

```
[Header] Content-Type -> application/json; charset=utf-8
```

```
[Header] Content-Length -> 42
```

```
{
  "errors" : [
    "message is a required field"
  ]
}
```

Let's test for invalid values

```
"Invalid fields" in {  
  server.httpPost(  
    path = "/tweet",  
    postBody = ""  
    {  
      "message": "",  
      "location": {  
        "lat": "9999",  
        "long": "-122.400612831116"  
      },  
      "sensitive": false  
    }""  
    ,  
    andExpect = BadRequest)  
}
```

Test Failed (expected 400 Bad Request)

```
=====
127.0.0.1 - - [25/Apr/2015:01:58:27 +0000] "POST /tweet HTTP/1.1" 201 98 742 "-"
=====
```

```
[Status] 201 Created
```

```
[Header] Content-Type -> application/json; charset=utf-8
```

```
[Header] Location -> http://127.0.0.1:49189/tweet/123
```

```
[Header] Content-Length -> 98
```

```
{
  "id" : "123",
  "message" : "",
  "location" : {
    "lat" : "9999.0",
    "long" : "-122.400612831116"
  },
  "sensitive" : false
}
```

```
201 Created did not equal 400 Bad Request
```


Validation Annotations

- CountryCode
- FutureTime
- PastTime
- Max
- Min
- NonEmpty
- OneOf
- Range
- Size
- TimeGranularity
- UUID
- MethodValidation

Let's add some validation annotations

```
case class PostedTweet(  
  @Size(min = 1, max = 140) message: String,  
  location: Option[Location],  
  sensitive: Boolean = false) {  
  
case class Location(  
  @Range(min = -90, max = 90) lat: Double,  
  @Range(min = -180, max = 180) long: Double)
```

Test Success

```
=====
127.0.0.1 - - [24/Apr/2015:23:01:10 +0000] "POST /tweet HTTP/1.1" 400 106 660 "-"
-----
```

```
[Status] 400 Bad Request
```

```
[Header] Content-Type -> application/json; charset=utf-8
```

```
[Header] Content-Length -> 106
```

```
{
  "errors" : [
    "message size [0] is not between 1 and 140",
    "location.lat [9999.0] is not between -90 and 90"
  ]
}
```

Next let's write a class to save a tweet

@Singleton

```
class TweetsService @Inject()(
  idService: IdService,
  firebase: FirebaseClient) {

  def save(postedTweet: PostedTweet): Future[Status] = {
    for {
      id <- idService.getId()
      status = postedTweet.toStatus(id)
      path = s"/statuses/${status.id}.json"
      _ <- firebase.put(path, status)
    } yield status
  }
}
```

Inject TweetsService into Controller

@Singleton

```
class TweetsController @Inject()(
    tweetsService: TweetsService)
extends Controller {

    post("/tweet") { postedTweet: PostedTweet =>
        tweetsService.save(postedTweet) map { status =>
            val renderableTweet = RenderableTweet.fromDomain(status)
            response
                .created(renderableTweet)
                .location(renderableTweet.id)
        }
    }
}
```

Update Server

//Before

```
class TwitterCloneServer extends HttpServer {  
  
  override def configureHttp(router: HttpRouter): Unit = {  
    router  
      .filter[CommonFilters]  
      .add(new TweetsController)  
  }  
}
```

//After

```
class TwitterCloneServer extends HttpServer {  
  
  override def modules = Seq(FirebaseHttpClientModule)  
  
  override def configureHttp(router: HttpRouter): Unit = {  
    router  
      .filter[CommonFilters]  
      .add[TweetsController]  
  }  
}
```

FirebaseHttpClientModule

```
object FirebaseHttpClientModule extends HttpClientModule {  
  
    override val dest = "flag!firebase"  
  
    override def retryPolicy = Some(exponentialRetry(  
        start = 10.millis,  
        multiplier = 2,  
        numRetries = 3,  
        shouldRetry = Http4xx0r5xxResponses))  
}
```

Rerun 'Post tweet' Test

```
"Post tweet" in {  
  server.httpPost(  
    path = "/tweet",  
    postBody = ...,  
    andExpect = Created,  
    withLocationHeader = "/tweet/123",  
    withJsonBody = ""  
      {  
        "id": "123",  
        "message": "Hello #SFScala",  
        "location": {  
          "lat": "37.7821120598956",  
          "long": "-122.400612831116"  
        },  
        "sensitive": false  
      }""")  
}
```


Test Fails: “no hosts are available”

```
=====
HTTP POST /tweet
```

```
{
  "message" : "Hello #SFScala",
  ...
}
```

```
=====
Request("PUT /statuses/374ea2dd-846a-45a5-a296-bcf43e5830c3.json")
```

```
service unavailable com.twitter.finagle.NoBrokersAvailableException:
```

```
No hosts are available for flag!firebase
```

```
127.0.0.1 - - [25/Apr/2015:02:37:01 +0000] "POST /tweet HTTP/1.1" 503 34 584 "-"
```

```
-----
[Status] 503 Service Unavailable
```

```
[Header] Content-Type -> application/json; charset=utf-8
```

```
[Header] Content-Length -> 34
```

```
{
  "errors" : [
    "service unavailable"
  ]
}
```

Let's mock our services

// Before

```
class TwitterCloneFeatureTest extends Test {  
    val server = new EmbeddedHttpServer(  
        twitterServer = new TwitterCloneServer)  
  
    "Post tweet" in { ... }  
}
```

// After

```
class TwitterCloneFeatureTest extends FeatureTest with Mockito {  
  
    override val server = new EmbeddedHttpServer(  
        twitterServer = new TwitterCloneServer {  
            override val overrideModules = Seq(integrationTestModule)  
        })  
  
    @Bind val firebaseClient = smartMock[FirebaseClient]  
  
    @Bind val idService = smartMock[IdService]  
  
    "Post tweet" in { ... }  
}
```

Let's mock our services

```
class TwitterCloneFeatureTest extends FeatureTest with Mockito {

  override val server = new EmbeddedHttpServer(
    twitterServer = new TwitterCloneServer {
      override val overrideModules = Seq(integrationTestModule)
    })

  @Bind val firebaseClient = smartMock[FirebaseClient]

  @Bind val idService = smartMock[IdService]

  "Post tweet" in {
    val statusId = StatusId("123")
    idService.getId returns Future(statusId)

    val putStatus = Status(id = statusId, ...)
    firebaseClient.put("/statuses/123.json", putStatus) returns Future.Unit

    val result = server.httpPost(
      path = "/tweet",
      ...)
```

Feature Test Success!

```
=====
HTTP POST /tweet
```

```
...
```

```
=====
127.0.0.1 - - [24/Apr/2015:22:38:31 +0000] "POST /tweet HTTP/1.1" 201 122 642 "-"
-----
```

```
[Status] 201 Created
```

```
[Header] Content-Type -> application/json; charset=utf-8
```

```
[Header] Location -> http://127.0.0.1:60352/tweet/123
```

```
[Header] Content-Length -> 122
```

```
{
  "id" : "123",
  "message" : "Hello #SFScala",
  "location" : {
    "lat" : "37.7821120598956",
    "long" : "-122.400612831116"
  },
  "sensitive" : false
}
```

Startup Test

```
class TwitterCloneStartupTest extends Test {  
  
    val server = new EmbeddedHttpServer(  
        stage = PRODUCTION,  
        twitterServer = new TwitterCloneServer,  
        clientFlags = Map(  
            "com.twitter.server.resolverMap" -> "firebase=nil!")  
    )  
  
    "server" in {  
        server.assertHealthy()  
    }  
}
```

Guice with Startup Tests

- Guice provides significant modularity and testing benefits
- Startup tests mostly mitigate lack of compile time safety
- We've found the combination to be a good compromise

Additional v2 Features

- Message body readers and writers
- Declarative request parsing
- Injectable flags
- Mustache Templates
- Multi-Server Tests
- Server Warmup

Message body readers and writers

```
class StatusMessageBodyWriter extends MessageBodyWriter[Status] {  
    override def write(status: Status): WriterResponse = {  
        WriterResponse(  
            MediaType.JSON_UTF_8,  
            RenderableTweet.fromDomain(status))  
        }  
    }  
}
```


Declarative request parsing

```
case class GetTweetRequest(  
  @RouteParam id: StatusId,  
  @QueryParam expand: Boolean = false)
```

```
case class GetTweetsRequest(  
  @Range(min = 1, max = 100) @QueryParam count: Int = 50,  
  @QueryParam expand: Boolean = false)
```

```
@Singleton
```

```
class TweetsController extends Controller {
```

```
  get("/tweet/:id") { request: GetTweetRequest =>  
    ...  
  }
```

```
  get("/tweets/?") { request: GetTweetsRequest =>  
    ...  
  }
```

```
}
```

Injectable flags

```
@Singleton
class TweetsService @Inject()(
    @Flag("max.count") maxCount: Int,
    idService: IdService,
    firebase: FirebaseClient) {

    ...
}
```

Mustache templates

```
// user.mustache  
id:{{id}}  
name:{{name}}
```

```
@Mustache("user")  
case class UserView(  
    id: Int,  
    name: String)
```

```
get("/") { request: Request =>  
    UserView(  
        123,  
        "Bob")  
}
```

Server Warmup

```
class TwitterCloneServer extends HttpServer {  
  override val resolveFinagleClientsOnStartup = true  
  
  override def warmup() {  
    run[TwitterCloneWarmup]()  
  }  
  
  ...  
}
```

Multi-Server tests

```
class EchoHttpServerFeatureTest extends Test {  
  
  val thriftServer = new EmbeddedThriftServer(  
    twitterServer = new EchoThriftServer)  
  
  val httpServer = new EmbeddedHttpServer(  
    twitterServer = new EchoHttpServer,  
    clientFlags = Map(  
      resolverMap("flag!echo-service" -> thriftServer.thriftExternalPort)))  
  
  "EchoHttpServer" should {  
    "Echo msg" in {  
      httpServer.httpGet(  
        path = "/echo?msg=Bob",  
        andExpect = Ok,  
        withBody = "Bob")  
    }  
  }  
}
```

Finatra Team

- Steve Cosenza
- Christopher Coco
- Jason Carey
- Eugene Ma

Questions?

```
class TwitterCloneServer extends HttpServer {  
  override val resolveFinagleClientsOnStartup = true  
  override def modules = Seq(FirebaseHttpClientModule)  
  override def warmup { run[TwitterCloneWarmup]() }  
  override def configureHttp(router: HttpRouter): Unit = {  
    router  
      .register[StatusMessageBodyWriter]  
      .filter[CommonFilters]  
      .add[TweetsController]  
  }  
}
```

@Singleton

```
class TweetsController @Inject()(  
  tweetsService: TweetsService)  
  extends Controller {  
  post("/tweet") { postedTweet: PostedTweet =>  
    tweetsService.save(postedTweet) map { status =>  
      response  
        .created(status)  
        .location(status.id)  
    }  
  }  
}
```