Reactive Instruments Workshop

An Activator template that can be used during Reactive Application workshops.

Steps

Below are some hints for each step in the tutorial.

Each step has a corresponding git commit for peeps who just want to get to the code without pressing to many keys...

See `git branch` for the available steps.

To jump to a specific step `git checkout <step name>`

Step 0 - Run Play

- Two ways of working with this tutorial (I will show both):
    - 1: Use terminal windows + any IDE of your choice
    - 2: Use Activator UI to develop, test and run the tutorial
- Start the Activator UI:

  `> <reactive-instrument-workshop-location>/activator ui`

- Start your Play application and test endpoint http://localhost:9000
    - In Activator: Run the project by clicking on the “Run” tab and then “Run”
    - In Terminal:

      `> activator run`

- Update “John Doe” in file app/controllers/Application.scala to your name, save the file and refresh the browser

Step 1 - Routes and Controllers

- Create a controller in *app/controllers* named InstrumentController with the method:

  def index : a simple Action that just returns a text `<h1>Instruments</h1>`

- Amend the conf/routes file so that /start points to the above index method
- Start your Play application and test endpoint http://localhost:9000/start
    - In Activator: Run the project by clicking on the “Run” tab and then “Run”
    - In Terminal:

      `> activator run`

Step 2 - Wire WebSocket Communication

- In the `app/controllers/InstrumentController` add the following method:
    - `def prices` that wires up a web socket connection like this:

      `WebSocket.acceptWithActor[String, String] { req => out => … }`

    - The prices method should return this (provided) actor `WebSocketHandler.props(out)`

    - Amend the routes file so that /prices points to the above prices method
    - Start Play application, open a browser console and type the following:

      `> var ws = new WebSocket(“ws://localhost:9000/prices");`

      `> ws.onmessage = function(x){ console.log(“RESULT:", x); }`

      When you’re done watching the output:

      `> ws.close();`

Step 3 - WebSocketHandler Dissection

Dissect the following parts of the existing `WebSocketHandler`

- Actor base classes: `ActorRef`, `Actor`, `ActorLogging`, `Props`
- companion object pattern for `Props`
- actor context
- actor context.system
- Scheduler functionality

Step 4 - Client and InstrumentController

Wire up a new view for prices

- Update method index in `app/controllers/InstrumentController` to use view `app/views/price.scala.html` file
    - hint: Ok(price.view.html())
- Study how the Javascript websocket connection is created in the above view file
    - hint: `<body onload=init()>` is where things are wired up
- Run Play and go to http//localhost:9000/start to see the new view

Step 5 - InstrumentController and WebSocketHandler

In this step we will wire up the communication between the price server and the client via our Play app. A suggested pattern is to send messages to `self`, i.e. the actor you're in, for the different steps involved. Make sure to add any case object/classes in the companion object of the actor class, i.e. `object WebSocketHandler`.

- Update prices in `app/controllers/InstrumentController` to handle `JsValue` instead of `String` (both directions)
    - hint: see the `[String, String]` syntax in there? It should take `JsValue` instead.
- Implement the `app/actors/WebSocketHandler` functionality
    - handle incoming JsValue message and extract instrument id
        - hints:
            - some good imports: `import play.api.libs.json._`
            - the receive method should no longer expect `String` but `JsValue` from the client
            - parsing can be done with `(json \ "x").asOpt[String]` where "x" is the JSON field you want to get (`asOpt[String]` means that the result will be either an `Option[String]` or `None`)
            - `Option` has a couple of useful methods that can be used to extract the data, e.g. `isDefined`, `get` or even better `myOption map { value => ... }`
            - make sure to set the instrument as local state in your actor (a var value in the class in other words)
    - call price server (http://localhost:8080/instrument/id) with the Play WS util lib

        - hints:
            - some good imports: `import play.api.libs.ws.WS`, `import play.api.Play.current`
            - WS usage: `WS.url("http://someurl").withRequestTimeout(5000).get`
                - the above will give you a `Future[WSResponse]` back and the easiest way to handle futures is something like this:

                  ```
                  val myFuture = // some Future[WsResult]
                  myFuture map { wsResult =>
                    val theBodyOfTheResponse = resp.body
                    // do something with the value above
                  } recover {
                    case e: ConnectException => // stop this actor, see
                  context.system.stop(...)
                    case e: TimeoutException => // send error message back to
                  web client
                  }
                  ```

                - pro-tip for the above is to send the `resp.body` as part of a case class to your `self`
    - parse JSON result from WS call
        - hints:
            - some good imports: `import play.api.libs.json._` and `import play.api.libs.functional.syntax._`
            - parsing of JSON can be done like this `val instrument = (json \ "instrument").asOpt[String]`
            - if you want to make something more elaborate you can take a look at Reads, e.g.

              ```
                case class PriceInfo(instrument: String, price: Int, timestamp:
              Long)
                implicit val priceInfoReads: Reads[PriceInfo] = (
                  (JsPath \ "instrument").read[String] and
                    (JsPath \ "price").read[Int] and
                    (JsPath \ "timestamp").read[Long]
                  )(PriceInfo.apply _)
              ```

    - calculate % fluctuation from last price update
        - hint: actors can have state! Keep the last price around to be able to calculate the fluctuation.
    - create JSON response and send to client via websocket
        - hints:
            - `Json.object("x" -> x, "y" -> y)` is your friend
            - Payload should contain: instrument, price, timestamp and fluctuation
    - schedule fetching new price in 2 seconds
        - hint: look at the Akka docs and use `context.system.schedule` to schedule a new retrieval of price in 2 seconds from the result being sent back to the web client.

Done! Now test the UI by starting the Price server:

```
> activator "runMain priceserver.frontend.RestMain"
```

The Play server:

```
> activator run
```

Browse to http://localhost:9000/prices and enter a instrument id (any random string will do).
Try this for a couple of different instrument ids. Do you notice that the initial response slows down? What happens if you enter more than five instrument ids? What's the reason for this?

Step 6 - Clustering of Price Service

As you could tell in Step 5 the solution we have created so far cannot cope with a lot of requests. This is due to a simulated slowness built into the class `priceserver.backend.PriceController`. In this step we will make sure that the application can handle more requests by using the Akka cluster feature. Instead of just having one price service node we will make sure that it is possible to use the number of nodes required to handle the load.

- Start off by creating a new main class used to run each node instance called `priceserver.backend.NodeMain`
    - Use `ClusterActorRefProvider` and set remote configuration in `application.conf`:

      ```
      akka {
        actor {
          provider = "akka.cluster.ClusterActorRefProvider"
        }
        remote {
          log-remote-lifecycle-events = off
          netty.tcp {
            hostname = "127.0.0.1"
            port = 0
          }
        }
      ```

    - Ensure that the main class can take a port number as input
- Create a `priceserver.backend.PriceService` to manage all incoming requests to the cluster
    - Make sure to register the `PriceService` actor with `ClusterReceptionistExtension`
    - Create a router with `FromConfig.props` (setting for this is done in the next step)
- Go back to `NodeMain` and set up `ClusterSingletonManager` and `ClusterSingletonProxy` for `PriceService` to ensure that there is only one `PriceService` instance running in the cluster at any point in time
    - hint: `ClusterSingletonManager` code:

      ```
      system.actorOf(ClusterSingletonManager.props(
        singletonProps = PriceService.props,
        singletonName = "priceService",
        terminationMessage = PoisonPill,
        role = None),
        name = "singletonManager")
      ```

    - hint: `ClusterSingletonProxy` code:

      ```
       system.actorOf(ClusterSingletonProxy.props(
         singletonPath = "/user/singleton/priceService",
         role = None))
      ```

    - hint: Make `PriceService` cluster aware using the *consistent-hashing-pool* router:

      ```
      akka.actor.deployment {
       /singletonManager/priceService/instrumentActor {
         router = consistent-hashing-pool
         nr-of-instances = 10
         cluster {
           enabled = on
           max-nr-of-instances-per-node = 2
           allow-local-routees = on
         }
       }
      }
      ```

    - hint: you must also configure some cluster seed nodes and add the cluster receptionist extension in the config file:

      ```
      akka
        cluster {
          seed-nodes = [
            "akka.tcp://ClusterSystem@127.0.0.1:2551",
            "akka.tcp://ClusterSystem@127.0.0.1:2552"]
        }

        extensions = ["akka.contrib.pattern.ClusterReceptionistExtension"]
      }
      ```

- Final step for the backend is to handle incoming messages in `PriceService` and to forward them with the help of the router
    - Forward a message of type `ConsistentHashableEnvelope`
- Now update the `priceserver.frontend.RestMain` to use `ClusterClient` to communicate with the actors in the cluster
    - Create a cluster client actor
        - hint: use `Set` this as initial contacts:

          ```
          val initialContacts = Set(

          system.actorSelection("akka.tcp://ClusterSystem@127.0.0.1:2551/user/rec
          eptionist"),

          system.actorSelection("akka.tcp://ClusterSystem@127.0.0.1:2552/user/rec
          eptionist"))
          ```

    - Use the above cluster client when communicating by invoking an ask (`?`) with the message `ClusterCliend.Send`

        - the actor address should be */user/singletonManager/priceService* (see the code for `ClusterSingletonManager` above)

To run you should first start a couple of nodes in the cluster. Let's start with the seed nodes:

```
> sbt "runMain priceserver.backend.NodeMain 2551"
```

and

```
> sbt "runMain priceserver.backend.NodeMain 2552"
```

Once the seed nodes are up and running you can start a couple of more nodes if you like:

```
> sbt "runMain priceserver.backend.NodeMain"
```

The next step is to start the price server front-end:

```
> activator "runMain priceserver.frontend.RestMain"
```

Finally run the Play server:

```
> activator run
```

Browse to http://localhost:9000/prices and enter a instrument id (any random string will do). Although we cheated in this step, by not using the `PriceController` which contains the slowdown code, I hope you do get that this solution can handle load much better. The more requests you have the more nodes you can start and the requests will be routed automatically to any new nodes in the cluster.

Credits

Background image in application from:

- http://www.zingerbug.com/background.php?MyFile=mountain_landscape_with_lake_oregon_cascades_background_1800x1600.php&ID=C766.php