Softwarearchitektur - Schach

Jörg Stenger

Functional Style

```
case class GridImpl @Inject()(cells: Matrix, specialSquares: SpecialSquares, turnStatus: TurnStatus) extends GridInterface {
  val BOARD_SIZE: Int = 8
  val injector = Guice.createInjector(new ChessModule)
  val pieceFactory: PieceFactory = injector.getInstance(classOf[PieceFactory])

  override def createNewGrid: GridImpl = setUpPieces.setTurnStatus(TurnStatus(Turn.P1))

  override def getSquare(square: (Int, Int)): Square = cells.getCell(square._1, square._2)

  override def replacePiece(row: Int, col: Int, value: Option[Piece]): GridImpl = copy(cells.replaceValue(row, col, value))
  override def replacePiece(square: (Int, Int), value: Option[Piece]): GridImpl = copy(cells.replaceValue(square._1, square._2, value))
```

```
def selectSquare(row: Int, col: Int): Option[GridImpl] = {
  if (!grid.squareIsSelected && turnColorMatches(row, col)) {
    Option(grid.replaceSelectedSquare((row, col)).highlightSquares(getPossibleMoves(grid.getSquare((row, col)).value.get)))
  } else {
    None
  }
}
```

Functional Style

```
override def undo(): Unit = {
  undoManager.undoMove match {
    case Success(resultGrid) => executeAndPublish(() => grid =resultGrid)
    case Failure(_) => println("Couldn't undo move.")
  }
}

override def redo(): Unit = {
  undoManager.redoMove match {
    case Success(resultGrid) => executeAndPublish(() => grid =resultGrid)
    case Failure(_) => println("Couldn't redo move.")
  }
}
```

```
def getSetCells: List[Square] = {
  for {
    row <- List.range(0, 8)
    col <- List.range(0, 8)
    if getCell(row, col).isSet
  } yield getCell(row, col)
}
```

```
override def highlightSquares(squares: List[Square]): GridImpl = {
  if (squares.isEmpty) this else highlightSquare(squares.head.row, squares.head.col).highlightSquares(squares.tail)
}
def highlightSquare(square: (Int, Int)): GridImpl = copy(cells.highlight(square._1, square._2))
```

Functional Style

```
override def createNewGrid: Unit = applyMoveResult(grid.createNewGrid)
override def selectSquare(row: Int, col: Int): Unit = applyMoveResult(grid.executeMove(row, col))

def applyMoveResult(grid: GridImpl): Unit = executeAndPublish(() => this.grid = grid)

def executeAndPublish(callback: () => Unit): Unit = {
  callback.apply()
  publish(new CellChanged)
}
```

```
def executeMove(row: Int, col: Int): GridImpl = {
  moveToSquare(row, col)
    .orElse(selectSquare(row, col))
    .getOrElse(grid.resetSelectedSquare().unhighlightAll())
}
```

```
override def save(): Unit = Future(fileIO.save(grid)).onComplete {
  case Success(_) => log.info("Grid saved successfully to file.")
  case Failure(_) => log.info("Grid could not be saved to file.")
}

override def load(): Unit = Future(fileIO.load).onComplete {
  case Success(result) => executeAndPublish(() => grid = result)
  case Failure(_) => log.info("Grid could not be loaded from file.")
}
```

REST

```
class WebView(controller: ControllerInterface) {
  implicit val system = ActorSystem()
  implicit val materializer = ActorMaterializer()
  implicit val executionContext = system.dispatcher

  val mainServiceHost: String = ConfigFactory.load().getString( path = "mainServiceHost")
  val mainServicePort: Int = ConfigFactory.load().getInt( path = "mainServicePort")
  val controllerActor = system.actorOf(Props(new ControllerActor(controller)), name = "controllerActor")

  val route = {
    path( pm = "chess" / "view") {
      get {
        implicit val timeout: Timeout = 5.seconds
        val grid: Future[String] = (controllerActor ? GetGrid).mapTo[String]
        complete(HttpEntity(ContentTypes.`text/html(UTF-8)`, "<pre>" + Await.result(grid, Duration.Inf) +"</pre>"))
      }
    }~
    path ( pm = "chess" / "select") {
      get {
        parameter("row".as[Int], "col".as[Int]) { (row, col) =>
          controllerActor ! (row, col)
          complete((StatusCodes.Accepted, "<pre>" + controller.gridString +"</pre>"))
        }
      }
    }~
```

```
val bindingFuture = Http().bindAndHandle(route, mainServiceHost, mainServicePort)

def unbind : Unit = {
  bindingFuture
    .flatMap(_.unbind())
    .onComplete(_ => system.terminate())
}
```

Actor Model

```
class ControllerActor(controller: ControllerInterface) extends Actor with ActorLogging with Reactor {
  listenTo(controller)

  def receive : PartialFunction[Any, Unit] = {
    case (row, col) => controller.selectSquare(row.asInstanceOf[Int], col.asInstanceOf[Int])
    case GetGrid => sender() ! controller.gridString
    case NewGrid => sender() ! controller.createNewGrid
    case _ => log.info("Invalid message")
  }
}

case object GetGrid
case object NewGrid
```

Slick

```
class DAOSlick extends DAOInterface {
  val grids = TableQuery[Grids]
  val database = Database.forConfig( path = "h2DB")
  database.run(DBIO.seq(grids.schema.create))

  override def create(gridJson: String): Unit = database.run(grids += (getSavedGridCount + 1, gridJson))

  override def read(id: Int): String = Await.result(database.run(grids.filter(_.id === id).result.headOption), Duration.Inf).get._2

  override def update(id: Int, gridJson: String): Boolean = Await.result(database.run(grids.filter(_.id === id).
    update(id, gridJson)) map { _ >= 0}, Duration.Inf)

  override def delete(id: Int): Boolean = Await.result(database.run(grids.filter(_.id === id).delete) map { _ >= 0}, Duration.Inf)

  def getSavedGridCount: Int = Await.result(database.run(grids.result),Duration.Inf).toList.size
}

case class Grids(tag: Tag) extends Table[(Int, String)](tag, _tableName = "Grids2") {
  def id: Rep[Int] = column[Int]("ID", O.PrimaryKey, O.AutoInc)
  def grid: Rep[String] = column[String]("Grid")

  def * : ProvenShape[(Int, String)] = (id,grid)
}
```

```
h2DB = {
  url = "jdbc:h2:mem:test1"
  driver = org.h2.Driver
  connectionPool = disabled
  keepAliveConnection = true
}
```

MongoDB

```
class DAOMongoDb extends DAOInterface {
  val codecRegistry: CodecRegistry = fromRegistries(fromProviders(classOf[Grid]), DEFAULT_CODEC_REGISTRY )
  val mongoClient: MongoClient = MongoClient()
  val database: MongoDatabase = mongoClient.getDatabase( name = "grids").withCodecRegistry(codecRegistry)
  val collection: MongoCollection[Grid] = database.getCollection( collectionName = "grid")

  override def create(gridJson: String): Unit = Await.result(collection.insertOne(Grid(getSavedGridCount + 1, gridJson))
    .toFuture(), Duration.Inf)

  override def read(id: Int): String = Await.result(collection.find(equal( fieldName = "_id", id)).first()
    .toFuture(), Duration.Inf).get()._2

  override def update(id: Int, gridJson: String): Boolean = {
    val filter = Document("_id" -> id)
    val mod = Document("$set" -> Document("grid" -> gridJson))

    Await.result(collection.updateOne(filter, mod).toFuture(), Duration.Inf).wasAcknowledged()
  }

  override def delete(id: Int): Boolean = Await.result(collection.deleteOne(equal( fieldName = "_id", id))
    .toFuture(), Duration.Inf).wasAcknowledged()

  def getSavedGridCount: Int = Await.result(collection.find().toFuture(),Duration.Inf).toList.map(grid => grid.get()).size
}

case class Grid(_id: Int, grid: String) {
  def get(): (Int, String) = (_id, grid)
}
```

Microservices

```
object DaoMicroservice {
  def main(args: Array[String]): Unit = {
    new DaoMicroserviceServer(new DaoMicroservice)
  }
}

class DaoMicroservice {
  val injector: Injector = Guice.createInjector(new ChessModule)
  var database: DAOInterface = injector.getInstance(classOf[DAOInterface])
}
```

```
class DaoMicroserviceServer(daoMicroservice: DaoMicroservice) {
  implicit val system: ActorSystem = ActorSystem()
  implicit val materializer: ActorMaterializer = ActorMaterializer()
  implicit val executionContext: ExecutionContextExecutor = system.dispatcher
  implicit val timeout: Timeout = 5.seconds

  val dbMicroserviceHost: String = ConfigFactory.load().getString( path = "dbMicroserviceHost")
  val dbMicroservicePort: Int = ConfigFactory.load().getInt( path = "dbMicroservicePort")
  val daoActor: ActorRef = system.actorOf(Props(new DaoActor(daoMicroservice.database)), name = "daoActor")

  val route: Route = {
    path( pm = "db" / "create") {
      post {
        parameter("gridJson".as[String]) { gridJson =>
          daoActor ! (Create, gridJson)
          complete((StatusCodes.Accepted, "Created"))
        }
      }
    }~
    path( pm = "db" / "read") {
      get {
        parameter("id".as[Int]) { id =>
          val gridJson: Future[String] = (daoActor ? (Read, id)).mapTo[String]
          complete((StatusCodes.Accepted, gridJson))
        }
      }
    }~
```

Microservices

```
class DaoActor(daoInterface: DAOInterface) extends Actor with ActorLogging with Reactor {
  override def receive: PartialFunction[Any, Unit] = {
    case (Create, gridJson) => sender() ! daoInterface.create(gridJson.asInstanceOf[String])
    case (Read, id) => sender() ! daoInterface.read(id.asInstanceOf[Int])
    case (Update, id, gridJson) => sender() ! daoInterface.update(id.asInstanceOf[Int], gridJson.asInstanceOf[String])
    case (Delete, id) => sender() ! daoInterface.delete(id.asInstanceOf[Int])
    case _ => log.info("Invalid message")
  }
}
```

```
path( pm = "chess" / "dbMicroservice" / "create") {
  post {
    val result = scalaj.http.Http(dbMicroserviceUrl + "/db/create").method("post")
      .param("gridJson", Json.prettyPrint(GridGenerator.gridToJson(controller.grid))).asString.body
    complete((StatusCodes.Accepted, "<pre>" + result + "</pre>"))
  }
}~
path( pm = "chess" / "dbMicroservice" / "read") {
  get {
    parameter("id".as[Int]) { id =>
      controller.grid = GridGenerator.gridFromJson(scalaj.http.Http(dbMicroserviceUrl + "/db/read").method("get")
        .param("id", id.toString).asString.body)
      complete(HttpEntity(ContentTypes.`text/html(UTF-8)`, "<pre>" + controller.gridString + "</pre>"))
    }
  }
}~
```

Docker

```
FROM zgwmike/akka-sbt
WORKDIR /main
EXPOSE 8070
ADD . /main
CMD sbt "runMain de.htwg.se.SE_Chess_HTWG.model.daoComponent.microserviceImpl.DaoMicroservice"
```

```
FROM zgwmike/akka-sbt
WORKDIR /main
EXPOSE 8080
ADD . /main
CMD sbt "runMain de.htwg.se.SE_Chess_HTWG.SE_Chess_HTWG"
```

```
dbMicroserviceHost = "0.0.0.0"
dbMicroservicePort = 8070
dockerDbMicroServiceUrl = "http://db-service"

mainServiceHost = "0.0.0.0"
mainServicePort = 8080
```

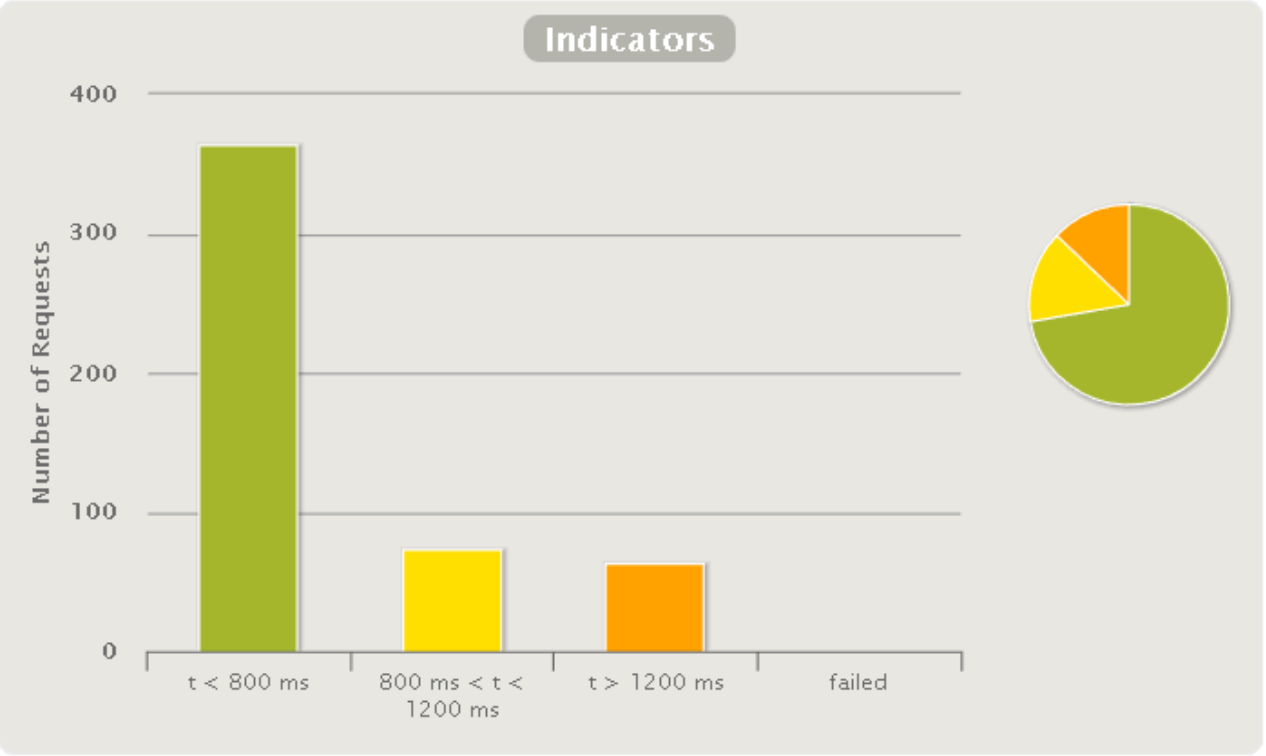
Docker

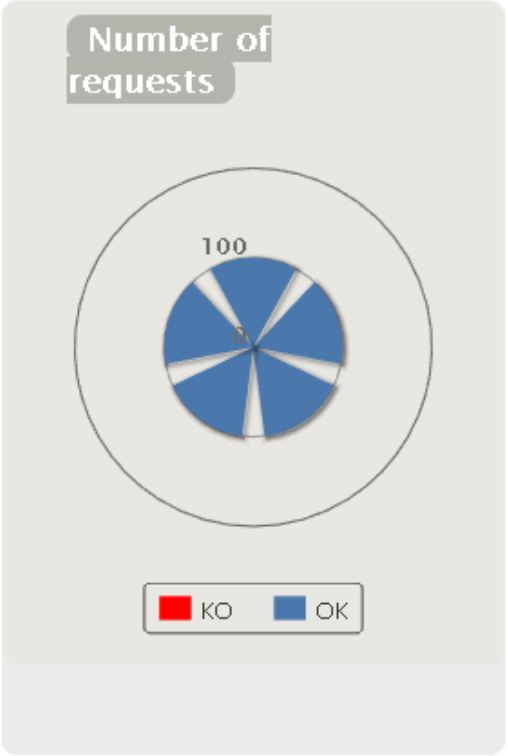
```yaml
version: '3.7'
services:
  mongosb:
    imagee: mongo
    volumes:
      - ./mongodb:/data/db
    ports:
      - 127.0.0.1:27017:27017

  main-service:
    build:
      context: .
      dockerfile: DockerfileMain
    image: main-service
    volumes:
      - .:/main
    ports:
      - "8080:8080"

  db-service:
    build:
      context: .
      dockerfile: DockerfileDb
    image: db-service
    volumes:
      - .:/main
    ports:
      - "8070:8070"
```

Gatling

```
class BasicSimulation extends Simulation{

  val httpProtocol = http.baseURL( url = "http://localhost:8080")

  val uri = "http://localhost:8080/"

  val scenario1 = scenario( scenarioName = "BasicSimulation")
      .exec(http( requestName = "request_0")
        .get("/chess/view"))
      .exec(http( requestName = "request_1")
        .get("/chess/new"))
      .exec(http( requestName = "request_2")
        .get("/chess/select?row=1&col=1"))
      .exec(http( requestName = "request_3")
        .get("/chess/select?row=2&col=1"))
      .exec(http( requestName = "request_4")
        .get("/chess/view"))

  setUp(
    scenario1.inject(atOnceUsers( users = 200))
  ).protocols(httpProtocol)
}
```

> Global Information



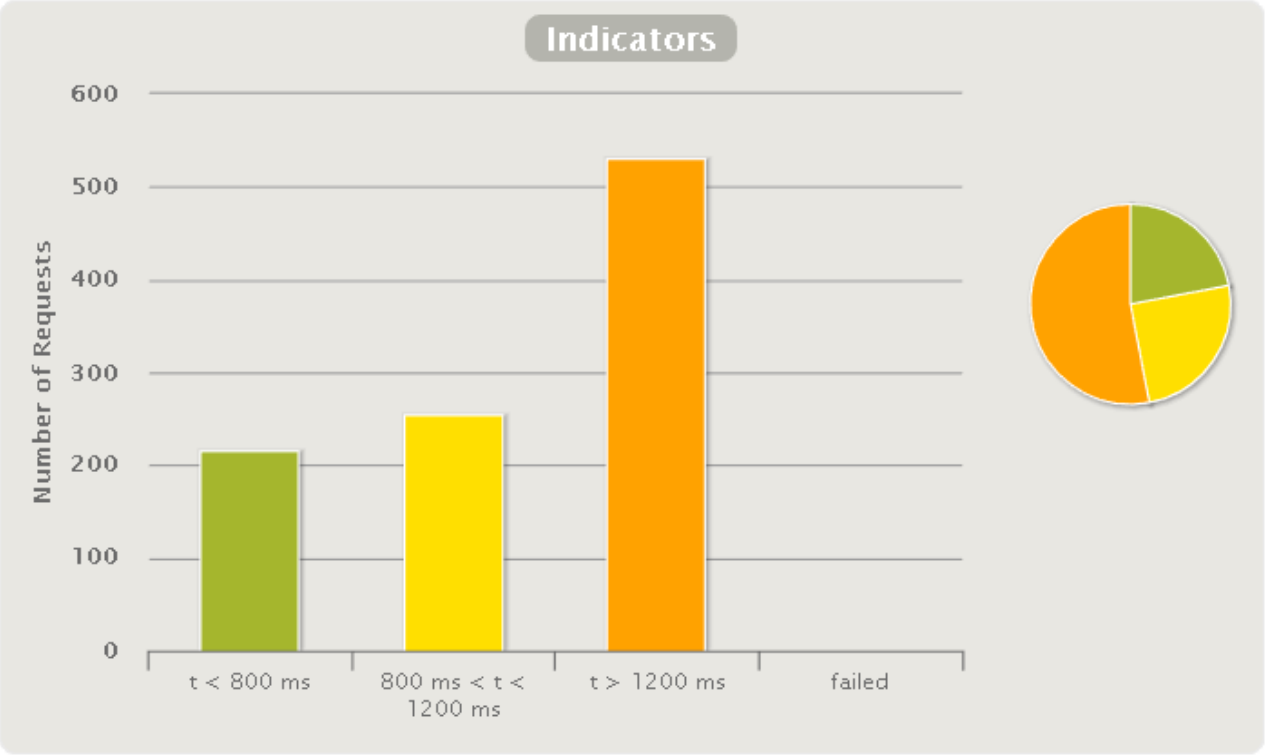


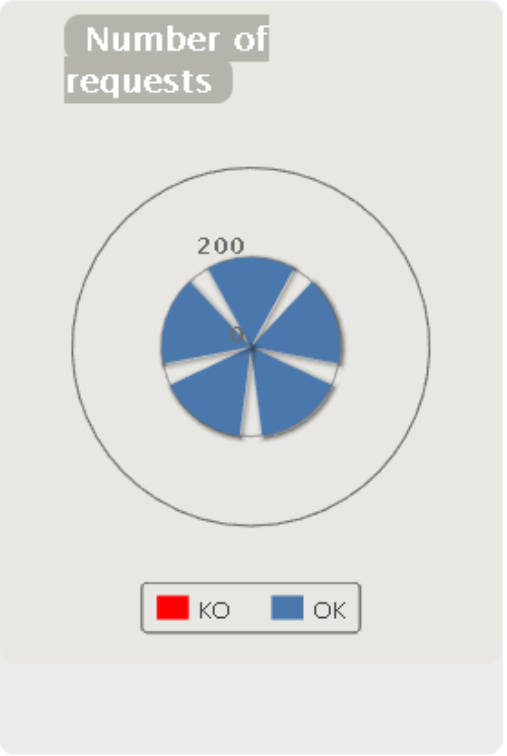
STATISTICS

Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	500	500	0	0%	62.5	2	544	858	1795	2717	3713	681	554
request_0	100	100	0	0%	12.5	9	1335	1800	2723	3609	3713	1376	769
request_1	100	100	0	0%	12.5	16	553	806	1100	1230	1639	618	305
request_2	100	100	0	0%	12.5	3	465	661	1077	1102	1219	511	265
request_3	100	100	0	0%	12.5	5	446	640	966	1107	1126	461	262
request_4	100	100	0	0%	12.5	2	422	601	979	1226	1410	441	300

> Global Information





STATISTICS

Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	1000	1000	0	0%	62.5	7	1230	1804	4335	5498	7042	1536	1127
request_0	200	200	0	0%	12.5	152	2959	4336	5499	6366	7042	2963	1619
request_1	200	200	0	0%	12.5	219	1236	1585	2086	2826	3222	1261	530
request_2	200	200	0	0%	12.5	193	1105	1531	1937	2459	2927	1188	483
request_3	200	200	0	0%	12.5	9	1156	1504	2114	2944	3053	1174	561
request_4	200	200	0	0%	12.5	7	1126	1427	2077	2739	3047	1093	580

Demonstration