Anatomy of a full-stack Scala/Scala.js Web App

Intro to Self

- Previous at Dropbox
- Currently at Bright Technology, a Data-Science/Scala consultancy
  - We do training and consulting projects around Python/Numpy/Scipy, Scala & related tech
  - Built the Fluent Code Browser www.fluentcode.com
- Contributor to Scala.js, author of Ammonite, FastParse, Scalatags, ...
  - www.lihaoyi.com
  - haoyi.sg@gmail.com

Agenda

- Deep dive into how a Scala/Scala.js projects ends up looking
- Not meant to be a “prep talk” or inspirational
- Full of nitty-gritty details

Intro to the Fluent Code Browser

- demo.fluentcode.com
- Blazing-fast online repository browser and search engine
- Works on repositories of all sizes, up to millions of lines of code
- Read-only view, background indexing
- Three person project, two engineers

Fluent Architecture

- Isomorphic Scala/Scala.js
  - 6500LOC JVM, 5500LOC JS, 2200LOC Shared
  - Akka-HTTP
  - Autowire/uPickle Ajax Routes
- Single-process
  - “Stateless” web-controller layer
  - Multiple background threads mirroring and indexing repositories
Shared Code

Constants

Colors

```javascript
object Colors {
```
```
val sidePane = "#212121"
val browsePane = "#2b2b2b"
val topPane = "#424242"

...
```java
else if(millisDelta / year == 1) "1 year ago"
else if(millisDelta / month > 1) millisDelta / month + " months ago"
...
}
```

Scalatags HTML Templates

**Standard Icons**

```scala
def devopsIcon(name: String) = {
  span(
    cls := s"devicons devicons-$name",
    styles.Base.devopIconStyle
  )
}
```

**Standard Tables**

```scala
def wrappingTable(tableHead: Option[Frag], contents: Frag*) = {
  table(
    cls := "table",
    tableLayout.fixed,
    styles.Base.normalTxt
  )(
    tableHead,
    tbody(contents)
  )
}
```

Wildly Different code

- Backend web server
- Frontend GUI

**Backend**

- Split into Stateless and Stateful code
- Stateless code is your typical web front-end: controllers, templates, etc.
  - No mutable state
  - Pure-ish functional
- Stateful code dealing with cloning/indexing git repos lives in repo-manager threads
  - Some mutable state
  - No global state
- Lives in same process for simplicity; could easily be split into separate workers

**Pure-ish Functional Controller Code**
```scala
def fetchPreview(filePath: GitPath, commitId: String) = {
  val commit = resolveIndexed(commitId)
  gitApi.queryFileOrFolder(commit, filePath) match{
    case Some(Left(objectId)) =>
      val lines = gitApi.open(objectId).lines.toArray
      PreviewResult.File(lines)
    case Some(Right(_)) => PreviewResult.Folder(...)
    case None => ???
  }
}
```

**Stateful Background Indexer**

```scala
var lastVersion = "...
var currentIndex: Option[Index] = None
while(true){
  pullRepo()
  val newVersion = currentVersion()
  if (newVersion == lastVersion) sleep()
  else{
    currentIndex = reIndex()
    lastVersion = currentVersion
  }
}
```

**Frontend**

- Lots of globals
- Lots of mutation via the DOM; currently not using React or other frameworks
- Decomposed hierarchically into Views

**Lots of globals:**

- Global click handler to close popups when you click somewhere else
- Global resize handler to make sure we only respond to resize events once
- Global Highlight.js lang-pack downloader & cache
- Modeled as top-level objects with mutable state
- Intrinsic global state in DOM

**WindowResize**

```scala
object WindowResize {
  def register(f: () => Unit) = ...
  def handle(e: dom.Event) = {
    val allElements = dom.document.getElementsByTagName("resize-callback-cls")
    for(k <- allElements) k.asInstanceOf[js.Dynamic].resizeCallback()
  }
}
```
Lots of mutation via the DOM; currently not using React or other frameworks
- Scala.Rx for simple "keep-things-in-sync" mutations
- Manual mangling for more ad-hoc mutations

```scala
def initCanvas(graphCanvas: dom.html.Canvas) = {
  graphCanvas.style.display = "block"
  graphCanvas.style.width = slice.pixelWidth.toString
  graphCanvas.height = (24 * dom.window.devicePixelRatio).toInt
  graphCanvas.style.height = 24.toString
}
```

Decomposed hierarchically into Views

```scala
trait View extends scalatags.jsdom.Frag{
  val view: dom.Node
}
class TreeView(...) extends View {...}
class LargeListView(...) extends View {...}
class DropdownInput(...) extends View {...}
```

Breakdown

<table>
<thead>
<tr>
<th></th>
<th>Server</th>
<th>Shared</th>
<th>Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines</td>
<td>6,500</td>
<td>2,200</td>
<td>5,500</td>
</tr>
<tr>
<td>Code</td>
<td>Akka-HTTP, JGit, Koloboke Collections, java.io, java.nio</td>
<td>Constants, Data-structures, Helper Functions, HTML Templates, CSS Stylesheets</td>
<td>Scala.Rx, Highlight.js, DOM interactions</td>
</tr>
<tr>
<td>Structure</td>
<td>Stateless controllers, Pure-ish functional, Long-lived, Lots of file IO</td>
<td>A grab-bag of standalone pieces of code</td>
<td>A hierarchy of stateful Views, Lots of references to third-part Javascript APIs</td>
</tr>
</tbody>
</table>

Performance Optimizations
- Both front-end and back-end are optimized to work well with large repos
- Back-end indexing must fit in memory and not take too long to create
- Front-end must lazy-load data and lazy-display UI to avoid crashing browser
Interesting back-end data-structures

- **Aggregator[T]**: specialized mutable.Buffer, reduces memory needed to store indices

```scala
class Aggregator[@specialized(Int, Long) T: ClassTag](initialSize: Int = 1) {
  // Can't be `private` because it makes `@specialized` explode
  protected[this] var data = new Array[T](initialSize)
  protected[this] var length0 = 0
  def length = length0
  def apply(i: Int) = data(i)
  def append(i: T) = {
    if (length >= data.length) {
      val newData = new Array[T](data.length * 3 / 2 + 1)
      System.arraycopy(data, 0, newData, 0, length)
      data = newData
    }
    data(length) = i
    length0 += 1
  }
}
```

Interesting front-end abstractions

- **FetcherLite**: Batching downloader
  - `FetcherLite.get(i: Int): Future[T]`
  - Pre-fetches items from `i-N` to `i+N` and caches them
  - Returns them instantly if asked for later

```scala
abstract class FetcherLite[T]{
  def fetchBatch(startCommitIndex: Int): Future[IndexedSeq[T]]
  var totalCount = rx.Var(0)
  var currentlyFetching = false
  var fetchQueue = List.empty[(Int, Promise[T])]
  var lastFetch: Option[(Int, IndexedSeq[T])] = None

  def get(commitIndex: Int): Future[T] = lastFetch match {
    case Some((lastStartIndex, lastFetchedCommits))
      if lastStartIndex <= commitIndex
      && commitIndex < lastStartIndex + lastFetchedCommits.length =>
        Future.successful(lastFetchedCommits(commitIndex - lastStartIndex))
    case None =>
      case Some((lastStartIndex, lastFetchedCommits))
      if commitIndex < lastStartIndex =>
        case Some((lastStartIndex, lastFetchedCommits))
        if commitIndex > lastStartIndex
```
```scala
  case _ =>
    val promise = Promise[T]()
    fetchQueue = (commitIndex -> promise :: fetchQueue)
    kickOffFetchIfNecessary()
    promise.future
  }
```

Final Thoughts

- Scala.js Benefits
  - Saves you from dealing with Javascript
  - Use Scala to type-check front-end, especially with Autowire
  - Use Scala to abstract common code patterns
  - Share common code between back-end and front-end
  - Shared libraries e.g. Scalatags/uPickle/autowire
  - Easy for Scala programmers to pick up
    - Other engineer who joined project had zero front-end experience
    - Was able to start making useful contributions in a few days

- Scala.js Limitations
  - Very different coding style for backend vs backend, despite same language
    - Stateless vs heavily Stateful
    - No Globals vs lots of Globals
    - "Principled" 3rd party APIs vs YOLO 3rd party APIs
  - Still need to write Front-end code, which is inherently hard/messy
    - Swing/AWT/QT/etc. aren't any better!
    - Still need to set up your own conventions/architecture/framework to keep things sane
    - Or use a third-party framework: React.js, Vue.js, Angular.js, Diode, ...

Conclusion

- Scala.js largely solves the problem of Javascript being complicated
- Scala.js doesn’t solve the problem of front-end UI being complicated
- Scala/Scala.js largely avoids incidental differences in client/server code
- Scala/Scala.js doesn’t avoid intrinsic differences in client/server code

Anatomy of a full-stack Scala/Scala.js Web App