**Anatomy of a Play application**

**The Play application layout**

The layout of a Play application is standardized to keep things as simple as possible. After the first successful compilation, the project structure looks like this:

app → Application sources

└ assets → Compiled asset sources

└ stylesheets → Typically LESS CSS sources

└ javascripts → Typically CoffeeScript sources

└ controllers → Application controllers

└ models → Application business layer

└ views → Templates

build.sbt → Application build script

conf → Configurations files and other non-compiled resources (on classpath)

└ application.conf → Main configuration file

└ routes → Routes definition

dist → Arbitrary files to be included in your projects distribution

public → Public assets

└ stylesheets → CSS files

└ javascripts → Javascript files

└ images → Image files

project → sbt configuration files

└ build.properties → Marker for sbt project

└ plugins.sbt → sbt plugins including the declaration for Play itself

lib → Unmanaged libraries dependencies

logs → Logs folder

└ application.log → Default log file

target → Generated stuff

└ resolution-cache → Info about dependencies

└ scala-2.11

└ api → Generated API docs

└ classes → Compiled class files

└ routes → Sources generated from routes

└ twirl → Sources generated from templates

└ universal → Application packaging

└ web → Compiled web assets

test → source folder for unit or functional tests

**The app/ directory**

The app directory contains all executable artifacts: Java and Scala source code, templates and compiled assets’ sources.

There are three packages in the app directory, one for each component of the MVC architectural pattern:

* app/controllers
* app/models
* app/views

You can add your own packages, for example, an app/utils package.

Note that in Play, the controllers, models, and views package names are simply conventions that can be changed if needed (such as prefixing everything with com.yourcompany).

There is also an optional directory called app/assets for compiled assets such as [LESS sources](http://lesscss.org/) and [CoffeeScript sources](https://coffeescript.org/).

**The public/ directory**

Resources stored in the public directory are static assets that are served directly by the Web server.

This directory is split into three sub-directories for images, CSS stylesheets and JavaScript files. You should organize your static assets like this to keep all Play applications consistent.

In a newly-created application, the /public directory is mapped to the /assets URL path, but you can easily change that, or even use several directories for your static assets.

**The conf/ directory**

The conf directory contains the application’s configuration files. There are two main configuration files:

* application.conf, the main configuration file for the application, which contains configuration parameters
* routes, the routes definition file.

If you need to add configuration options that are specific to your application, it’s a good idea to add more options to the application.conf file.

If a library needs a specific configuration file, it is good to provide it under the conf directory.

**The lib/ directory**

The lib directory is optional and contains unmanaged library dependencies, ie. all JAR files you want to manually manage outside the build system. Just drop any JAR files here and they will be added to your application classpath.

**The build.sbt file**

Your project’s main build declarations are generally found in build.sbt at the root of the project. The .scala files in the project/ directory can also be used to declare your project’s build.

**The project/ directory**

The project directory contains the sbt build definitions:

* plugins.sbt defines sbt plugins used by this project.
* build.properties contains the sbt version to use to build your app.

**The target/ directory**

The target directory contains everything generated by the build system. It can be useful to know what is generated here:

* classes/ contains all compiled classes (from both Java and Scala sources).
* classes\_managed/ contains only the classes that are managed by the framework (such as the classes generated by the router or the template system). It can be useful to add this class folder as an external class folder in your IDE project.
* resource\_managed/ contains generated resources, typically compiled assets such as LESS CSS and CoffeeScript compilation results.
* src\_managed/ contains generated sources, such as the Scala sources generated by the template system.
* web/ contains assets processed by [sbt-web](https://github.com/sbt/sbt-web#sbt-web) such as those from the app/assets and public folders.

**Typical .gitignore file**

Generated folders should be ignored by your version control system. Here is the typical .gitignore file for a Play application:

logs

project/project

project/target

target

tmp

dist

.cache

**Default sbt layout**

You also have the option of using the default layout used by SBT and Maven. In order to use this layout, you must disable the layout plugin and set up explicit monitoring for twirl templates:

disablePlugins(PlayLayoutPlugin)

This will stop Play from overriding the default sbt layout, which looks like this:

build.sbt → Application build script

src → Application sources

└ main → Compiled asset sources

└ java → Java sources

└ controllers → Java controllers

└ models → Java business layer

└ scala → Scala sources

└ controllers → Scala controllers

└ models → Scala business layer

└ resources → Configurations files and other non-compiled resources (on classpath)

└ application.conf → Main configuration file

└ routes → Routes definition

└ twirl

└ views → Templates

└ assets → Compiled asset sources

└ css → Typically LESS CSS sources

└ js → Typically CoffeeScript sources

└ public → Public assets

└ css → CSS files

└ js → Javascript files

└ images → Image files

└ test → Unit or functional tests

└ java → Java source folder for unit or functional tests

└ scala → Scala source folder for unit or functional tests

└ resources → Resource folder for unit or functional tests

└ universal → Arbitrary files to be included in your projects distribution

project → sbt configuration files

└ build.properties → Marker for sbt project

└ plugins.sbt → sbt plugins including the declaration for Play itself

lib → Unmanaged libraries dependencies

logs → Logs folder

└ application.log → Default log file

target → Generated stuff

└ scala-2.11.12

└ cache

└ classes → Compiled class files

└ classes\_managed → Managed class files (templates, ...)

└ resource\_managed → Managed resources (less, ...)

└ src\_managed → Generated sources (templates, ...)

Django uses a threaded architecture unlike Play framework which uses Evented architecture. I wanted to compare Django with Play framework.

Which one performs better if we deploy the services written in Django and Play on the hardware with same configuration, also given that business logic for the service has been written efficiently in both the frameworks?

Which framework has the potential to scale to 1000+ concurrent requests?

Threads and events are not a dichotomy, Play gives you threads and an event loop.

This question is too broad, it really depends on what you're trying to implement…..

- in many many use cases the web framework is not the bottleneck, it's the database.

But if you insist on getting on a comparison that completely ignores whatever your specific use case is, then here's a comparison:

Once you have sbt installed, the following at the command prompt will start up Play in development mode (move to application directory): C:> sbt run

Play will start up on the HTTP port at <http://localhost:9000/>. You don't need to deploy or reload anything -- changing any source code while the server is running will automatically recompile and hot-reload the application on the next HTTP request (browser refresh).

**Scala advantages**

By choosing Spark as a processing framework that is internally written in Scala, you will be limited in programming languages to Scala, Python, Java, C# and R. However, you become enabled to write unit and integration tests in a framework of your choice, set up a team-based development project with less painful code merges, leverage source control, build, deployment and continuous integration features. The framework and language will protect you from many errors and bugs that are bread and butter of every IT project.

Besides all advantages of modern programming language, Scala gives you benefits such as:

1. conciseness and less code than in a language such as Java or C#,
2. immutable data structures allowing for parallel, lock-free data processing,
3. full-featured object-oriented paradigms, so you are not limited to functional programming
4. performance that is better than interpreted languages such as Python,
5. good compatibility with the MapReduce processing model,
6. a plethora of well-designed libraries for scientific computing.

case class WordCount(word: String, count: Int)

object WordCountExample extends Base {

  def main(args: Array[String]) = {

    val spark = SparkSession.builder

      .master("local")

      .appName("Word Count Example")

      .getOrCreate()

    // Extract

    val textFile = spark.sparkContext.textFile("C:/spark/README.md")

    // Transform

    val stopWords = List("the", "and", "to", "for")

    val wordCounts = count(spark, textFile, stopWords)

   // Load

   wordCounts.map {

     case WordCount(word, count) => (word: String, count: Int)

   }.rdd.saveAsTextFile("C:/wordcount/")

  }

  def count(ss: SparkSession, lines: RDD[String]): Dataset[WordCount] = {

    count(ss, lines, List.empty)

  }

  def count(ss: SparkSession, lines: RDD[String], stopWords: List[String]): Dataset[WordCount] = {

    val words = lines.flatMap(\_.split("\\s"))

      .map(\_.stripSuffix(".").stripSuffix(",").toLowerCase)

      .filter(!stopWords.contains(\_))

      .filter(!\_.isEmpty)

    val wordCounts: RDD[WordCount] = words.map(word => (word, 1)).reduceByKey(\_ + \_).map {

      case (word: String, count: Int) => WordCount(word, count)

    }

    import ss.implicits.\_

    wordCounts.toDS

  }

}

ETL code refactored

I must just say that Scala achieves this behind the curtain by linearization and these class-like constructs are called traits, that can be mixed with your base class. At this point you might think that traits are like C#/Java interfaces with concrete methods, but they can actually do much more. This benefits our ETL to achieve great flexibility in how you design an entire solution.

This is one of the goals of Spark development team where simple things should be simple, and complex things should be possible.

One of the most interesting features is concept of a dataset, which is a strongly typed collection of domain-specific objects. These can be transformed in parallel by using functional or relational operations. The core of the dataset is a new concept called an encoder, which is responsible for converting between JVM objects and their tabular representation. The tabular representation is stored using Spark internal binary format, allowing for operations on serialized data and improved memory utilization. Thanks to this, dataset implementation runs much faster with less memory pressure than the naive RDD implementation.

If you look closer at a unit test example introduced in previous code snippets, you can see that we are already used a Spark dataset. In the example below, the dataset operates on type-safe domain objects with a compile-time error on not existing domain member. (Type-safe dataset)

val df: DataFrame = spark.read.json("people.json")

 // Convert data to domain objects

case class Person(name: String, age: Long)

val ds: Dataset[Person] = df.as[Person]

// Show the results

ds.filter(\_.age > 30).show()

ds.filter(\_.name startsWith "J").show()

// Compile time error: value lastname is not a member of Person

ds.filter(\_.lastname startsWith "A").show()

The noteworthy thing here is the way that data is converted to domain objects. We use the case class construct, very popular in Scala due to its many advantages. We use it because it eliminates a lot of unnecessary boiler plate code – at a small price. You have to write the case modifier, and your classes become a bit larger. Case classes fit ideally for data structures and Spark leverages this Scala feature with type-safe datasets objects.

It gives you as a developer a convenience that at least during development you don’t need to worry about schema validation of your text data sources. From now on, most schema related issues can be captured during compilation instead of runtime. That is a huge relief.

I was trying to show you that there are different approaches to doing ETL. However, it isn’t easy to choose an architecture that is right for you. It is not just about the volume of data but also the development approach of the organization and the available skills of the development team. The issues are not just about the technology but also about the culture of work and mindset in your organization.

It is important to note that Spark is a Big Data framework, so you must build a full Hadoop cluster for your ETL. This could be expensive, even for open-source products and cloud solutions. On the other hand, if you are not a Big Data fan, you still need to make an investment in an expensive enterprise-ready ETL tool. Probably you will also have to provision a dedicated server environment for your ETL if your dataset is quite big. So, there is still a trade-off. What I can say that right now, with Scala and Spark you have an alternative that does not force you to use a hammer to drive in a screw.

Support for various IDEs: You can add support for your IDE by entering: playeclipsify, playidealize, or

playnetbeansify. Every command generates the files needed to import a Play application into your favorite IDE.