Purpose of Each Line

SparkSession.builder()

Creates a SparkSession.Builder object, which allows you to configure your SparkSession. This is the starting point for initializing Spark in an application.

.appName("Example")

Sets the name of your Spark application. This name appears in the Spark UI and logs, making it easier to identify and debug your application.

.master("local[*]")

Configures the deployment mode. In this case:
"local[*]" means the application runs in local mode, using all available CPU cores on the machine.
This is useful for testing and development on a single machine.

.getOrCreate()

Creates a new SparkSession if one does not already exist. If a SparkSession is already running, it returns the existing session.

Parentheses Grouping

The parentheses are used to group the chain of method calls. This is especially useful in Scala's REPL (spark-shell) to make the code more readable and avoid issues with method chaining.

val spark

Stores the created SparkSession object in the variable spark, which you use throughout your application to:
Read data.
Create DataFrames or Datasets.
Run queries.

SparkSession is needed to:

Interact with Spark's core components like DataFrames, Datasets, and SQL.
Access Spark's distributed data processing capabilities.
Configure application-specific settings, such as:
Application name (appName).
Resource allocation (master).

Example Usage:

Read a file into a DataFrame:

```
val df = spark.read.format("json").load("/path/to/data.json")
df.show()
```

Perform SQL queries:

```
df.createOrReplaceTempView("data")
spark.sql("SELECT * FROM data").show()
```

Why Use .master("local[*]")?

The "local[*]" mode is for testing and debugging on your local machine. The [*] indicates Spark will utilize all available CPU cores for processing, making it efficient for local development.

If deploying to a cluster, you would configure the master URL to point to the cluster manager (e.g., YARN, Mesos, or Kubernetes).

A schema is a StructType made up of a number of fields, StructFields, that have a name, type, a Boolean flag which specifies whether that column can contain missing or null values, and, finally, users can optionally specify associated metadata with that column.

The metadata is a way of storing information about this column (Spark uses this in its machine learning library).

The example that follows shows how to create and enforce a specific schema on a DataFrame.

```
import org.apache.spark.sql.types.{StructField, StructType, StringType, LongType}
import org.apache.spark.sql.types.Metadata
val myManualSchema = StructType(Array(
    StructField("DEST_COUNTRY_NAME", StringType, true),
    StructField("ORIGIN_COUNTRY_NAME", StringType, true),
    StructField("count", LongType, false,
        Metadata.fromJson("{\"hello\":\"world\"}"))
))
val df = spark.read.format("json").schema(myManualSchema)
.load("/home/kaizen/data/flight-data/json/2015-summary.json")
```

Columns and Expressions in Spark

Columns

1. Columns in Spark are similar to those in:

   Spreadsheets.
   R DataFrame.
   pandas DataFrame.
2. Operations like selecting, manipulating, and removing columns are represented as **expressions**.
3. Columns are logical constructs representing a value computed on a per-record basis using expressions.
4. A column's value depends on:

   The presence of a **row**.
   The row being part of a **DataFrame**.
5. Individual column manipulation is only possible within the context of a DataFrame using Spark transformations.
6. Simplest ways to refer to columns:

```
import org.apache.spark.sql.functions.{col, column}
col("someColumnName")
column("someColumnName")
```

7. Column resolution (matching column names with those in the catalog) happens during the **analyzer phase**.

Explicit Column References

8. Use the col method on a specific DataFrame to explicitly refer to its columns:

   df.col("count")

9. Explicit references are useful in scenarios like joins, where columns from different DataFrames might share the same name.

10. Explicit references help Spark skip column resolution during the analyzer phase, as it is already specified.

Expressions

11. Columns are essentially **expressions**.
12. An **expression**:

    Represents transformations on one or more values in a DataFrame record.
    Functions like a **function**, resolving input column names and applying transformations.
13. Expressions can output **complex types** like:

Map.
Array.

Simplest expression:
```
expr("someCol") == col("someCol")
```

Example of transformations using expressions:
```
expr("someCol - 5") == col("someCol") – 5
```

Spark compiles expressions into a **logical tree** that specifies the order of operations.