Notes for Instructions on Creating Storage Plug-ins

(Rough notes being collected from exploration; to be refined into instructions.)

Terminology:

- "*storage plug-in*" refers to a software module (as opposed to a configuration or a run-time instance thereof); (e.g., the Drill file plug-in, a Cassandra plug-in).
- "*storage plug-in configuration class*" refers to a (concrete) classes implementing/extending `StoragePluginConfig`
- "*storage plugin implementation class*" refers to a (concrete) classes implementing/extending `StoragePlugin`
- "*storage plug-in configuration*" refers generally to a combination of values for the configuration parameters for a storage plug-in; it's general in that it can refer to any of:
    - a storage plug-in configuration class instance (the combination represented by it),
    - a JSON (or other) representation of such a combination, and
    - the combination associated with a storage plug-in implementation class instance.
- "*storage plug-in configuration name*" refers to the name corresponding to a storage plug-in configuration (e.g., "`dfs`")
- "*storage plug-in type name*" refers to the name used with the "`type`" field (of what?) to identify a specific storage plug-in (or storage plug-in type?); (e.g., "`file`")
- There is also the natural-language phrase used to refer to a plug-in (as opposed to its type name); (e.g., "the file storage plug-in", "the Hive plug-in").
- There are old and/or ambiguous uses (in un-updated references in the Web UI, documentation, and/or code):
    - some old "storage plugin" mean "storage plug-in configuration" [UI]
    - some old "storage name" mean "storage plug-in configuration name" [UI]
    - some "configuration" mean "storage plug-in configuration" [UI]
    - some old "storage plugin" mean "storage plug-in configuration name"
    - some old "storage engine" mean "storage plug-in configuration name" [code]
    - some old "storage engine" mean "storage plug-in configuration" [code]
    - some old "storage plug-in instance" mean "storage plug-in configuration"
    - (Other cases surely exist.)

Need class for storage plug-in configuration ("storage plugin configuration class"):

- per `StoragePluginConfig` in `org.apache.drill.common.logical`
    - [`StoragePluginConfig` probably should be interface (not abstract class)]
- instance represents/holds configuration data for one configuration of a plug-in
- public class
- public no-argument constructor? (depending on Jackson)
- one class per … exactly what?

    - Current code supports having multiple `StoragePluginConfig` subclasses map to a single `StoragePlugin` subclass (distinguishing via parameter type of multiple constructors). Q: What is the intended use of that?
- plug-in type name (for use in "… `dfs: { type: "file"` …") can be specified in Jackson `@JsonTypeName` annotation on subclass's declaration; defaults to simple (unqualified) name of class
- Q: What are requirements on serializability? Presumably, all relevant state must be serializable via Jackson, but what exactly does that devolve to? (What annotations are required? Does Jackson default to JavaBeans rules? Are there usual Jackson annotations?)

Need class for storage plugin ("storage plugin implementation class"):

- per `StoragePlugin` in `org.apache.drill.exec.store`
    - [`StoragePluginConfig` and `StoragePlugin` should be in same module and probably in same package]
- instance implements one configuration of the plug-in
    - (multiple instances per configuration across nodes/Drillbits?)
- public class? (not clearly needed)
- public constructor `SomeStoragePlugin( SomeStoragePluginConfig config, DrillContext context, String configName )` for each associated storage plugin configuration ~type/class, where
    - `SomeStoragePluginConfig` is the *specific* implementation class of `StoragePluginConfig`
    - `configName` is name from ??
    - `configName` is used for ??
    - [Drill should use a factory to define the correspondence from a storage plugin configuration class to a storage plugin classes, rather than using reflection and hidden/undocumented requirements that can't be checked at compile time.]

Classpath-scanning requirements:

- Need `drill-module.conf` file in root of classpath subtree(s) containing plugin's configuration and implementation classes (so classpath scanning scans subtree).
- Need name of configuration class's package listed in configuration property `drill.logical.storage.packages` (so scanning scans package).*
- Need name of implementation class's package listed in configuration property `drill.exec.storage.packages` (so scanning scans package).*
- Normally, list in containing classpath root's `drill-module.conf` file.
- *Note: Package name lists are not currently used. Not putting package name in list may not cause failure currently but cause failure when things change.

Bootstrap configurations:

- Normally want default serialized (JSON) configurations in a `bootstrap-storage-plugins.json` file:
    - Probably can be in any classpath root with a `drill-module.conf` file, but probably should be in "own" classpath root.
- Q: What exactly is format?

    - Format seems to Jackson's serialization of some kind of list of `StoragePluginConfig`.
    - Jackson seems to follow JavaBeans getter/setting mapping rules (?? verified only for ~simple values--String, boolean)
    - ??What else?

Need to implement `StoragePlugin` (and `SchemaFactory`) methods:

- TBD: What additional documentation is needed in code? Then, what is needed here?
- `StoragePluginConfig getConfig()`
- `boolean supportsRead()`
- `boolean supportsWrite()`
- `void registerSchemas(SchemaConfig schemaConfig, SchemaPlus parent)`
- `Set<StoragePluginOptimizerRule> getOptimizerRules()`
- `AbstractGroupScan getPhysicalScan(String userName, JSONOptions selection)`
- `AbstractGroupScan getPhysicalScan(String userName, JSONOptions selection, List<SchemaPath> columns)`

Need to expose schema(s), tables and columns, create/drop operations, etc.

Need to register schemas starting with `StoragePlugin.registerSchemas( SchemaConfig schemaConfig, SchemaPlus parent )`:

- ROUGH NOTES:
    - `SchemaPlus` is from Calcite
    - `SchemaConfig` is from Drill
    - Need to create and register schema(s) provided by plug-in (instance) under given schema (always root?).
    - Normally one top-level schema?
    - Can have subschemas. If more than one subschema, then normally alias "`default`" for one. What if subschema but only one (other than "`default`")--is there normally "`default`" then?
    - Is top-level schema's name supposed to match storage plug-in configuration's name, or is that just the common case?
    - Need to implement `AbstractSchema` (etc.) methods:
        - ??: Remove from list `AbstractSchema` not normally needing overriding.
        - TBD: What additional documentation is needed in code? Then, what is needed here?
        - `Schema.getTable(String)?`
        - `Schema.getTableNames()?`
        - `Schema.getFunctions(String)?`
        - `Schema.getFunctionNames()?`
        - `Schema.getSubSchema(String)?`
        - `Schema.getSubSchemaNames()?`
        - `Schema.getExpression(SchemaPlus, String)?`
        - `Schema.isMutable()?`

        - `Schema.contentsHaveChangedSince(long, long)?`
        - `AbstractSchema.AbstractSchema(List<String>, String)?`
        - `AbstractSchema.getSubPartitions(String, List<String>, List<String>)?`
        - `AbstractSchema.getName()?`
        - `AbstractSchema.getSchemaPath()?`
        - `AbstractSchema.getFullSchemaName()?`
        - `AbstractSchema.getTypeName()?`
        - `AbstractSchema.getDefaultSchema()?`
        - `AbstractSchema.createView(View)?`
        - `AbstractSchema.dropView(String)?`
        - `AbstractSchema.createNewTable(String, List<String>)?`
        - `AbstractSchema.showInInformationSchema()?`
        - `AbstractSchema.close()?`
        - `SchemaPartitionExplorer.getSubPartitions(String, List<String>, List<String>)?`

Need to represent table metadata using `Table` and `DrillTable`:

- ROUGH NOTES:
    - Apparently, implementations of table must all be subclasses of `DrillTable`.
    - Need to implement `Table` and `DrillTable` methods:
        - TBD: What additional documentation is needed in code? Then, what is needed here?
        - `Table.getRowType(RelDataTypeFactory)`
        - `Table.getStatistic()`
        - `Table.getJdbcTableType()`
        - `DrillTable.DrillTable(String, StoragePlugin, String, Object)`
        - `DrillTable.DrillTable(String, StoragePlugin, Object)`
        - `DrillTable.getGroupScan()`
        - `DrillTable.getSelection()`
        - `DrillTable.getStatistic()`
        - `DrillTable.toRel(ToRelContext, RelOptTable)`
        - `DrillTable.getJdbcTableType()`
        - `DrillTable.hashCode()`
        - `DrillTable.equals(Object)`
    - Need to create Calcite `RelDataType` for row type, including `RelDataType` for each column/field in row type.
        - Might need to be careful to be careful to get right Drill-specific (not Calcite-default) default sizes/precisions for types.

- `StoragePlugin`'s `getOptimizerRules()`
- `StoragePlugin`'s `getPhysicalScan(String, JSONOptions)`

- `StoragePlugin`'s `getPhysicalScan(String, JSONOptions, List<SchemaPath>)`

- basic scan
- optimized scan?
- optimizer rules (`getOptimizerRules()`)

Pending questions:
- Q: What does the `@JsonTypeInf`o annotation on `StoragePluginConfig` do? Specifically, how exactly does it relate to "`type`" in '`type: "file"`' and to "`NAME`" and "`name`" fields (JavaBeans/Jackson properties?) on plugin classes? (What is the chain from serialized configuration through to schema names visible in SQL?)

- Q: What does `@JsonProperty` do/mean? (note "`size.calculator.enabled`")
- Q: What does `SystemTablePluginConfig`'s *public* `NAME` field do?
- Q: What does `SystemTablePluginConfig`'s *public* `INSTANCE` field do?
- Q: Which name is "`storageEngineName`" / "storage engine name"?