Code Style

We recommend you use IntelliJ as your IDE. The code style template for the project can be found in the codestyle repository along with our general programming and Java guidelines. In addition to those you should also adhere to the following:

Alphabetize

Alphabetize sections in the documentation source files (both in the table of contents files and other regular documentation files). In general, alphabetize methods/variables/sections if such ordering already exists in the surrounding code.

Use streams

When appropriate, use the stream API. However, note that the stream implementation does not perform well so avoid using it in inner loops or otherwise performance sensitive sections.

Categorize errors when throwing exceptions.

Categorize errors when throwing exceptions. For example, PrestoException takes an error code as an argument, PrestoException(HIVE_TOO_MANY_OPEN_PARTITIONS). This categorization lets you generate reports so you can monitor the frequency of various failures.

Add license header

Ensure that all files have the appropriate license header; you can generate the license by running mvn license:format.

Prefer String formatting

Consider using String formatting (printf style formatting using the Java Formatter class): format("Session property %s is invalid: %s", name, value) (note that format() should always be statically imported). Sometimes, if you only need to append something, consider using the + operator. Please avoid format() or concatenation in performance critical sections of code.

Avoid ternary operator

Avoid using the ternary operator except for trivial expressions.

Define class API for private inner classes too

It is suggested to declare members in private inner classes as public if they are part of the class API.

Avoid mocks

Do not use mocking libraries. These libraries encourage testing specific call sequences, interactions, and other internal behavior, which we believe leads to fragile tests. They also make it possible to mock complex interfaces or classes, which hides the fact that these classes are not (easily) testable. We prefer to write mocks by hand, which forces code to be written in a certain testable style.

Use AssertJ

Prefer AssertJ for complex assertions.

Use Airlift's Assertions

For thing not easily expressible with AssertJ, use Airlift's Assertions class if there is one that covers your case.

Avoid var

Using var is discouraged.

Prefer Guava immutable collections

Prefer using immutable collections from Guava over unmodifiable collections from JDK. The main motivation behind this is deterministic iteration.

Maintain production quality for test code

Maintain the same quality for production and test code.

Format Git commit messages

When writing a Git commit message, follow these guidelines.

Avoid abbreviations

Please avoid abbreviations, slang or inside jokes as this makes harder for non-native english speaker to understand the code. Very well known abbreviations like max or min and ones already very commonly used across the code base like ttl are allowed and encouraged.

Additional IDE configuration

When using IntelliJ to develop Presto, we recommend starting with all of the default inspections, with some modifications.

Enable the following inspections:

- Java | Internationalization | Implicit usage of platform's default charset,
- Java | Control flow issues | Redundant 'else' (including Report when there are no more statements after the 'if' statement option),
- Java | Class structure | Utility class is not 'final',
- Java | Class structure | Utility class with 'public' constructor,
- Java | Class structure | Utility class without 'private' constructor.

Disable the following inspections:

- Java | Performance | Call to 'Arrays.asList()' with too few arguments,
- Java | Abstraction issues | 'Optional' used as field or parameter type.

Enable errorprone (Error Prone Installation#IDEA):

- Install Error Prone Compiler plugin from marketplace,
- In Java Compiler tab, select Javac with error-prone as the compiler,
- Update Additional command line parameters with -XepExcludedPaths:.*/target/generated-(|test-)sources/.* -XepDisableAllChecks -Xep:MissingOverride:ERROR ...... (for current recommended list of command line parameters, see the top level pom.xml, the definition of the errorprone-compiler-presto profile.

Building the Web UI

The Presto Web UI is composed of several React components and is written in JSX and ES6. This source code is compiled and packaged into browser-compatible Javascript, which is then checked in to the Presto source code (in the dist folder). You must have Node.js and Yarn installed to execute these commands. To update this folder after making changes, simply run:

```
yarn --cwd presto-main/src/main/resources/webapp/src install
```

If no Javascript dependencies have changed (i.e., no changes to package.json), it is faster to run:

```
yarn --cwd presto-main/src/main/resources/webapp/src run package
```

To simplify iteration, you can also run in watch mode, which automatically re-compiles when changes to source files are detected:

```
yarn --cwd presto-main/src/main/resources/webapp/src run watch
```

To iterate quickly, simply re-build the project in IntelliJ after packaging is complete. Project resources will be hot-reloaded and changes are reflected on browser refresh.