Using Report observers to implement new report types without having to write API calls directly during tests.

During the tests, we log information and report to various mechanisms. The very basic ones are logs and some sort of a reporting mechanism, such as ExtentsReport, Allure, etc.

During the development, we might be asked to report and log information for other reasons - log information for Jira, introduce another reporting tool like Report portal, and write to a central logging mechanism such as Splunk. We might need some specific information like a list of all incoming and outgoing HTTP calls which were made during the test

If we were to write a line of code for each interface we want to report to, we would have far more logging and reporting lines of code than tests. This is why the Verisoft framework uses the Observer mechanism in order to report once during the test, and let other modules do the work of reporting to all the necessary logging and reporting module. If you are not familiar with the Observer design pattern, or the Singleton design pattern now it's a good time to pause and read about it a bit.

How does it work?

Verisoft framework holds a singleton named ReportPublisher. The ReportPublisher is the subject (which is also known as a publisher, and we will use this phrase from now on) of the observers. All other observers register to that publisher. The framework ships with the following default observers: SLF4J observer, Extent Report Observer, ReportPortalObserver, CloudBeat Observer. In addition, it is super easy to create your own observer, and we will demonstrate that feature shortly.

When a new logging message arrives to the publisher, it notifies all the observers - the message, its additional object (if applicable), and metadata. The observer desires what to do with the message.

Practically, during the test we recommend using the `Report` class, which contains APIs to the ReportPublisher object. Here is a simple example of using the publisher, which, in turn, notifies all the observers that a new message was created:

```
1  Report.info("Info message - SHOULD appear on log");
```

That's it! This simple line of code will be sent to the publisher, which will then notify all the registered observers. A full example test might look like this. This is a unit test to test a sample new observer:

```
1  @Test
2  public void shouldWriteEntry() {
3
4      // Setup
5      List<String> list = sampleObserver.getEntries();
6
7      // Test
8      Report.trace("Trace message - should only appear on log");
9
10     // Note!! Verisoft Assert
11     Asserts.assertTrue(list.stream().anyMatch(x -> x.contains("Trace message")),
12                                                          "Should appear on log");
13 }
```

Note that Verisoft's Assert mechanism also writes messages to the ReportPublisher. This is why we recommend using the framework's Assert instead of other Assert mechanisms, even though every Assert mechanism will function with the framework itself, just without the logging and reporting part of it.

The publisher holds log levels - Trace, Debug, Info, Warning, Error, and Fatal.

The structure of a ReportPublisher message

A ReportPublisher message contains the following information:

- Report source (optional value)- an enum with the values LOG, REPORT, and OTHER.
- Report level (mandatory value)- an enum with the values TRACE, DEBUG, INFO, WARNING, ERROR, and FATAL
- The message we would like to log (mandatory value)
- An additional object we would like to attach to the message - it's an Object. It can hold any type of object. Common usages are Exception objects or Screenshots.

Here is an example of how to write directly to the ReportPublisher without the use of the Report. The following method is actually a method taken from the `Report` class:

```
@Synchronized
public static void report(ReportSource source, ReportLevel level,
                                                String msg, Object object) {

    ReportEntry reportEntry = ReportEntry.builder()
                              .reportSource(source)
                              .reportLevel(level)
                               .msg(msg)
                              .additionalObject(object)
                              .build();

    ReportPublisher.getInstance().notifyObserver(reportEntry);
}
```

But, for the most part, it is more convenient to use the Report object instead of calling directly to the ReportPublisher. Here is an example of the overloaded info methods:

```
public static void info(String msg);

public static void info(String msg, Object object);

public static void info(ReportSource source, String msg);

public static void info(ReportSource source, String msg, Object object);
```

Next, we will demonstrate how to create and register a custom observer.