Working with Multiple Drivers Injection

Setting Up Multiple Drivers for Your Test

You can configure more than one driver for your test. For instance, if you need to test a website in Chrome, receive an SMS message on your mobile, and then return to the website, you can do so using multiple drivers. Each driver can be initialized with specific capabilities and a URL, as well as a dedicated `CommandExecutor`.

Initializing Specific Driver Capabilities

There are three methods to initialize specific capabilities for a driver:

1. Extracting from a JSON File

Define the capabilities in a JSON file with the following format:

```json
[
  {
    "name": "chromeJson",
    "caps": {
      "browserName": "chrome"
    }
  },
  {
    "name": "edgeJson",
    "caps": {
      "browserName": "MicrosoftEdge"
    }
  },
  {
    "name": "mobileJson",
    "caps": {
      "platformName": "Android",
      "platformVersion": "14"
    }
  }
]
```

Specify the path to this file in the application properties file as follows:

```
capabilities.file.path=./src/test/resources/my_capabilities.json
```

If no path is defined, the default is `src/test/resources/capabilities.json`.

To use these capabilities, reference the name in your test method:

```java
@Test
public void twoDriversWithJsonCapabilities(@DriverCapabilities("chromeJson") VerisoftDriver chromeDriver,
                                           @DriverCapabilities("edgeJson") VerisoftDriver edgeDriver) {}
```

2. By Bean Injection

Use Spring injection to define capabilities. Define a configuration class:

```java
@Configuration
@ComponentScan("co.verisoft.examples")
public class CapabilitiesInjection {
    @Bean("chrome")
    @Primary
    public Capabilities getChromeCapabilities() {
        ChromeOptions options = new ChromeOptions();
        options.addArguments("--no-sandbox", "--headless");
        return options;
    }

    @Bean("edge")
    public Capabilities getEdgeCapabilities() {
        EdgeOptions options = new EdgeOptions();
        options.addArguments("--no-sandbox");
        return options;
    }
}
```

In your test class, include this configuration:

```java
@ExtendWith({SpringExtension.class, DriverInjectionExtension.class, ScreenShotExtension.class})
@ContextConfiguration(classes = {CapabilitiesInjection.class})
public class MultipleDriversTests {}
```

Reference the bean name in your test method:

```java
@Test
public void twoDriversWithBeanCapabilities(@DriverCapabilities("chrome") VerisoftDriver chromeDriver,
                                           @DriverCapabilities("edge") VerisoftDriver edgeDriver) {}
```

3. By Field in the Test Class

If no specific capabilities are defined, the driver will use the capabilities defined in the class field:

```java
@DriverCapabilities
EdgeOptions options = new EdgeOptions();
{
    options.addArguments("--headless");
}
```

Usage: (The edge driver will use the EdgeOptions capabilities)

```java
@Test
public void twoDriversWithFieldCapabilities(@DriverCapabilities("chrome") VerisoftDriver chromeDriver,
                                            VerisoftDriver edgeDriver) {}
```

Initializing a Specific URL for the Driver

There are three options for initializing a specific URL for the driver:

1. **Hard-coded in Annotation**:

```java
@Test
public void driverWithHardCodedURL(@DriverCapabilities("chrome") @DriverUrl("<http://1.2.3.4:4444/wd/hub/>") V
```

2. **By Bean Injection**: (add this bean to CapabilitiesInjection class)

```java
@Bean("seleniumGridUrl")
public URL getSeleniumGridUrl() throws MalformedURLException {
    return new URL("<http://1.2.3.4:4444/wd/hub/>");
}
```

Usage:

```
@Test
public void driverWithUrlBean(@DriverCapabilities("chrome") @DriverUrl("seleniumGridUrl") VerisoftDriver chrom
```

3. **By Field in the Class**:

```
@DriverUrl
String url = "<http://1.2.3.4:4444/wd/hub/>";
```

Usage:

```
@Test
public void driverWithFieldURL(@DriverCapabilities("chrome") VerisoftDriver chromeDriver) {}
```

Initializing DriverCommandExecutor

There are three methods to initialize `DriverCommandExecutor`:

1. **By Sending the Name of the Proxy Class**:

```
@Test
public void driverWithProxyCommandExecutor(@DriverCapabilities("mobileJson") @DriverCommandExecutor("org.openq
```

2. **By Bean Injection**:

```
@Bean("command Executor")
   public HttpCommandExecutor getCommandExecutor() throws MalformedURLException {
       URL serverUrl = new URL("http://127.0.0.1:4723/wd/hub/");
       return new AppiumCommandExecutor(MobileCommand.commandRepository, serverUrl);
   }

```

Usage:

```
@Test
public void driverWithExecutorBean(@DriverCapabilities("mobileLeumi") @DriverCommandExecutor("commandExecutor") V
```

3. **By Field in the Class**:

```
@DriverCommandExecutor
private HttpCommandExecutor commandExecutor = new AppiumCommandExecutor(MobileCommand.commandRepository, new U
```

Usage:

```
@Test
public void driverWithFieldExecutor(@DriverCapabilities("mobilejson") VerisoftMobileDriver driver) {}
```

This structured approach allows you to manage multiple drivers efficiently in your testing environment.

The `@DriverName` Annotation

The `@DriverName` annotation is used to assign a unique name to each driver managed by the `VerisoftDriverManager`. This allows for easy retrieval of specific drivers by name within your tests.

Assigning Names to Drivers

To assign a name to a driver, use the `@DriverName` annotation. You can then retrieve the driver by its name as shown in the example below:

```
@Test
public void getDriverByName(@DriverName("my driver name") VerisoftDriver driver) {
    VerisoftDriver extractedDriver = VerisoftDriverManager.getDriver("my driver name");
}

```

In this example, `@DriverName("my driver name")` assigns the name `"my driver name"` to `driver`. The `VerisoftDriverManager.getDriver(String driverName)` method is then used to retrieve the driver by its name.

If there is one driver, it can be retrieved by `VerisoftDriverManager.getDriver()` without a specific name.

If **no name is provided** for a driver and your test involves **multiple drivers**, the `VerisoftDriverManager.getDriver()` will throw an error. You should give it a name so you can retrieve it if there is more than one driver.

Retrieving All Drivers

You can retrieve a map of all drivers currently managed within the test using the following method:

```
Map<String, VerisoftDriver> allDrivers = VerisoftDriverManager.getDrivers();
```

This method returns a map where each key is the name of a driver and each value is the corresponding `VerisoftDriver` instance.

Using the `@DriverName` annotation simplifies the management and utilization of multiple drivers in complex testing scenarios, ensuring that each driver can be uniquely identified and accessed as needed.

Happy Coding!