NIMBLE DOCUMENTAION

Transform your testing game with Nimble

Abstract

Nimble, where automation meets simplicity and efficiency

.

In a world where testing can feel like a never-ending maze, NIMBLE emerges as your trusty guide. With our fully automated test framework, we take the pain out of testing processes. Say goodbye to the days of confusion and frustration, and hello to a streamlined, efficient approach that even your grandma could master!

Viom Technology Services

Table of Contents

Setting Up a Nimble Test Project

1. Using Sample Template:

If you're here, we assume you've already downloaded **Nimble** from VIOM's portal. This guide will help you set up a sample test project that will contain your feature files and test scenarios.

✅ Step 1: Download Sample Test Project

Download the **Sample-Nimble-Client** project using this link. This project contains everything needed to get started quickly.

✅ Step 2: Locate Setup Scripts

Once extracted, go to the root folder of the project and locate the following files:

- `nimble_mobile_setup_mac.sh`
- `nimble_web_setup_mac.sh`
- `nimble_mobile_setup_windows.bat`

✅ Step 3: Install Prerequisites

Depending on your OS and whether you're testing Mobile, Web, or both, run the appropriate script:

For macOS/Linux/Unix:

- **Mobile or Cross-platform:**
  Run: `nimble_mobile_setup_mac.sh`
- **Web or Cross-platform:**
  Run: `nimble_web_setup_mac.sh`

For Windows:

- **Mobile or Cross-platform:**
  Run: `nimble_mobile_setup_windows.bat`

✅ Step 4: Verify Installation

Once the scripts complete, verify installations using:

```
java -version
mvn -version
node -v  # (only if needed)
```

✅ Step 5: Install Nimble Jar File

1. Download the Nimble package from www.viom.tech.
2. Extract the ZIP file to a known location.
3. Run the following command to install the `.jar` into your local Maven `.m2` repository:

```
mvn install:install-file \
  -Dfile=/your/path/to/nimble-1.0.0.jar \
  -DgroupId=tech.viom \
  -DartifactId=nimble \
  -Dversion=1.0.0 \
  -Dpackaging=jar
```

✅ Note:

- Make sure the path in `-Dfile` is correct and points to the `.jar` file.
- Ensure the version mentioned in `-Dversion` matches your download.

That's it! You're all set to start writing and executing test scenarios using Nimble.

2. From Scratch

✅ Overview

The Nimble framework is a pluggable test automation solution that supports both **web and mobile** testing. To use it in a **client project**, you need to:

1. Structure your project properly
2. Download and Import the Nimble JAR as a dependency.
3. Implement custom hooks and properties
4. Create your `testng.xml` or `feature` files
5. Write step definitions or reuse existing ones
6. Configure Maven to run tests

🔧 Step-by-Step Setup

1. Create a Maven Project Structure

```
Sample-Nimble-Client/
│
├── pom.xml
├── testng.xml (or your runner)
├── libs/                          # Place Nimble JAR here (optional)
├── src/
│   ├── main/
│   │   └── java/                  # (Optional for app code)
│   └── test/
│       ├── java/
│       │   ├── com.client.hooks/ # Your implementation of ICustomHooks
│       │   ├── com.client.steps/ # Additional step definitions if needed
│       │   └── com.client.tests/ # Test runner / entry point
│       └── resources/
│           └── features/          # Cucumber .feature files
```

---

2. Install Nimble JAR to Local Maven Repo (Recommended)

Run:

```
mvn install:install-file \
  -Dfile=/path/to/nimble-1.0.0.jar \
  -DgroupId=tech.viom \
  -DartifactId=nimble \
  -Dversion=1.0.0 \
  -Dpackaging=jar
```

This puts Nimble into your `~/.m2` repo for Maven to pick up.

3. Add Nimble Dependency to `pom.xml`

```
<dependency>
    <groupId>tech.viom</groupId>
    <artifactId>nimble</artifactId>
    <version>1.0.0</version>
</dependency>
```

---

4. Implement Your `ICustomHooks`

Create a class like this:

```
public class ClientHooks implements ICustomHooks {

    @Override
    public Map<String, String> storeEnvironmentProperties() {
        Map<String, String> props = new HashMap<>();
        props.put("env", "staging");
        return props;
    }

    @Override
    public Platform setPlatform() {
        return Platform.MOBILE; // or Platform.WEB
    }

    @Override
    public PlatformName setPlatformName() {
        return PlatformName.ANDROID; // OR CHROME, SAFARI, etc.
    }

    @Override
    public String setUrl() {
        return "http://localhost:4723/wd/hub"; // or Selenium Grid URL
    }

    @Override
    public DesiredCapabilities setDesiredCapabilities() {
        DesiredCapabilities caps = new DesiredCapabilities();
        caps.setCapability("platformName", "Android");
        return caps;
    }

    @Override
    public void initialize() {
        // optional: custom logic before test starts
    }

    @Override
    public void terminate() {
        // optional: custom logic after all tests
    }
}
```

Ensure it's in the classpath and is the **only implementation** of `ICustomHooks`.

5. Add `testng.xml`

Here's a minimal example:

```
<suite name="Sample Suite">
  <test name="Cucumber Tests">
    <classes>
      <class name="com.nimble.runner.TestRunner"/>
    </classes>
  </test>
</suite>
```

6. Create Feature Files

Example (`src/test/resources/features/login.feature`):

```
Feature: Login functionality

  Scenario: User logs in with valid credentials
    Given launch the app
    And enter "user" into "Username"
    And enter "pass" into "Password"
    And click "Login"
    Then Check that screen contains "Welcome"
```

Nimble provides generic step definitions for common actions like click, input, verify, etc.

Running Feature files in Sample-Nimble-Client

Run all tests for a web app and on specific browser and a specific tag

```
mvn clean test -Dcucumber.features=src/test/java/com/client/feature  -Dplatform=web -
Dplatform.name=chrome -Durl=https://www.emirates.com/ae/english/

OR

mvn clean test -Dcucumber.features=src/test/java/com/client/feature -
Dcucumber.filter.tags="@RunFirst or @Emirates" -Dplatform=web -Dplatform.name=chrome
```

Run tests with specific tag on specific browser

```
mvn clean test -Dcucumber.features=src/test/java/com/client/feature -Dcucumber.filter.tags="@Try" -
Dplatform=web -Dplatform.name=edge

OR

mvn clean test -Dcucumber.features=src/test/java/com/client/feature -
Dcucumber.filter.tags="@RunFirst or @Emirates" -Dplatform=web -Dplatform.name=edge
```

Run a specific feature file of a webapp on a specific browser

```
mvn test -Dcucumber.features="src/test/resources/features/<feature_file>.feature" -Dplatform=web -
Dplatform.name=firefox

OR

mvn clean test -Dcucumber.features=src/test/java/com/client/feature -
Dcucumber.filter.tags="@RunFirst or @Emirates" -Dplatform=web -Dplatform.name=firefox
```

Run a specific feature file of a android app

```
mvn clean test -Dcucumber.features=src/test/java/com/client/feature -Dplatform=mobile -
Dplatform.name=android -Dplatform.version=14 -Dcucumber.filter.tags="@TryMobile"
```

Run a specific feature file of a iOS app

```
mvn clean test -Dcucumber.features=src/test/java/com/client/feature  -Dplatform=mobile -
Dplatform.name=ios -Dplatform.version=14 -Dcucumber.filter.tags="@AHB"
```

Configurations

Environment specific configurations should managed in environment specific property files as shown below.

Environment	Description
`DEV`	config-dev.properties
`QA`	config-qa.properties
`SIT`	config-sit.properties

Reports

After execution, test reports can be found in:

```
target/cucumber-reports/
```

To open an HTML report:

```
open target/cucumber-reports/index.html
```

Nimble Commands

Clicking On The UI Elements

This guide provides an overview of Cucumber commands available in the Nimble framework for clicking UI elements in mobile and web applications. These commands are designed to be intuitive and flexible, allowing you to interact with buttons, links, forms, and other elements during automated testing. Each command is written in Gherkin syntax and can be used in your .feature files.

General Guidelines

- **Text in Quotes:** When specifying text (e.g., button labels), enclose it in double quotes (e.g., "Submit").

- Optional Parameters: Some commands include optional parts (e.g., tags, classes, exact matches). Include them only when needed.

- Supported Platforms: These commands work for both mobile and web apps, automatically adapting to the application type.

- Errors: If an element isn't found, Nimble will raise an error (e.g., "Element not found"). Ensure your UI contains the specified elements.

1. Click an Element by Text

Description: Clicks an element based on its visible text, with optional filters like tag type, CSS class, or exact phrase matching.

Syntax:

```
click "TEXT" [inside TAG] [with class "CLASS"] [with exact phrase]
```

- "TEXT": The visible text of the element (e.g., "Submit").

- inside TAG (optional): The HTML tag type (e.g., button, a, div). Supported tags: select, option, label, button, a, input, textarea, form, div, span, p, img, svg, iframe.

- with class "CLASS" (optional): The CSS class of the element (e.g., "nav-item").

- with exact phrase (optional): Ensures the text matches exactly (case-sensitive).

Examples:

```
Given click "Submit"                    # Clicks any element with "Submit"
Given click "Login" inside button       # Clicks a button with "Login"
Given click "Next" inside div with class "navigation"  # Clicks a div with class "navigation"
containing "Next"
```

```
Given click "Download" inside a with exact phrase      # Clicks a link with the exact text
"Download"
```

2. Click the Nth Element by Text

Description**:** Clicks a specific occurrence (e.g., 1st, 2nd, 3rd) of an element based on its text, with optional tag and class filters.

Syntax:

```
click on the NUMBER(st|nd|rd|th) "TEXT" [inside TAG] [with class "CLASS"] [with exact phrase]
```

- NUMBER: The position (e.g., 1, 2, 3).

- (st|nd|rd|th): The ordinal suffix (e.g., 1st, 2nd, 3rd, 4th).

- "TEXT": The visible text of the element.

- inside TAG (optional): The HTML tag type (see supported tags above).

- with class "CLASS" (optional): The CSS class of the element.

- with exact phrase (optional): Ensures an exact text match.

Examples:

```
Given click on the 1st "Submit" button         # Clicks the first "Submit" button
Given click on the 2nd "Next" inside a         # Clicks the second "Next" link
Given click on the 3rd "Item" with class "list-item"  # Clicks the third "Item" with class "list-
item"
Given click on the 4th "Proceed" with exact phrase    # Clicks the fourth "Proceed" with exact text
```

Note: The position starts at 1 (e.g., 1st is the first occurrence).

3. Click a Relative Element

Description: Clicks an element relative to another (e.g., parent, child, sibling) based on its text or the currently focused element.

Syntax:

```
click on the NUMBER(st|nd|rd|th) RELATION [of the element with text "TEXT" [with exact phrase] | of
the element in focus]
```

- NUMBER: The position (e.g., 1, 2, 3).

- (st|nd|rd|th): The ordinal suffix.

- RELATION: The relationship:

- previous-sibling: Element before the reference.

- next-sibling: Element after the reference.

- child: Element inside the reference.

- parent: Element containing the reference.

- "TEXT" (optional): The text of the reference element.

- with exact phrase (optional): Ensures an exact text match.

- of the element in focus (optional): Uses the currently focused element instead of text.

Examples:

```
Given click on the 1st child of the element with text "Menu"      # Clicks the first child of "Menu"
Given click on the 2nd parent of the element in focus            # Clicks the second parent of the
focused element
Given click on the 3rd next-sibling of the element with text "Item" with exact phrase  # Clicks the
third next sibling of "Item"
```

4. Click an Element by ID

Description: Clicks an element using its ID, with an optional fallback ID or index if multiple elements match.

Syntax:

```
click on the element with id as "ID1" [or "ID2"] [at index NUMBER]
```

- "ID1": The primary ID of the element.

- or "ID2" (optional): A fallback ID if the first isn't found.

- at index NUMBER (optional): The position (e.g., 1, 2) if multiple elements have the ID.

Examples:

```
Given click on the element with id as "submitButton"              # Clicks element with ID
"submitButton"
Given click on the element with id as "login" or "loginBtn"       # Tries "login", then "loginBtn"
Given click on the element with id as "item" at index 2           # Clicks the second element with
ID "item"
```

5. Click an Element by Content Description (Mobile Only)

Description: Clicks a mobile element based on its content description (accessibility label), with an optional index.

Syntax:

```
click on the element contains desc as "DESC" [at index NUMBER]
```

- "DESC": The content description of the element.

- at index NUMBER (optional): The position if multiple elements match.

Examples:

```
Given click on the element contains desc as "Submit"         # Clicks element with description "Submit"
Given click on the element contains desc as "Item" at index 2  # Clicks the second element with description "Item"
```

Note: This is primarily for mobile apps where elements have accessibility descriptions.

6. Tap Near an Element (Mobile Only)

Description: Taps at a specific offset from an element's position, useful for mobile apps.

Syntax:

```
tap near "TEXT" with offset x NUMBER [and y NUMBER] [with exact phrase]
```

- "TEXT": The text of the reference element.

- x NUMBER: The X-axis offset in pixels from the element's position.

- and y NUMBER (optional): The Y-axis offset in pixels (defaults to the element's Y position if omitted).

- with exact phrase (optional): Ensures an exact text match.

Examples:

```
Given tap near "Submit" with offset x 100                  # Taps 100 pixels right of "Submit"
Given tap near "Login" with offset x 50 and y 20           # Taps 50 pixels right and 20 pixels down from "Login"
Given tap near "Button" with offset x 200 with exact phrase  # Taps 200 pixels right of exact "Button"
```

Note: This is mobile-specific and useful for tapping outside an element (e.g., near a button).

Tips for Writing Feature Files

- Combine Steps: Use these commands in Given, When, or Then clauses as needed:

```
  Scenario: Submit a form
    Given click "Login" inside button
    When click on the element with id as "submitButton"
    Then click on the 1st child of the element with text "Success"
```

- Test Your UI: Ensure the text, IDs, or descriptions match what's in your app.

- Handle Ambiguity: Use tags, classes, or exact phrases to avoid clicking the wrong element.

Generating Random Values

This guide outlines the Cucumber commands available in the Nimble framework for generating random values during automated testing. These commands allow you to create random strings, numbers, or dates based on regex patterns and save them to a cache for later use. Each command is written in Gherkin syntax and can be used in your .feature files.

General Guidelines

- Text in Quotes: Enclose regex patterns and variable names in double quotes (e.g., "[a-zA-Z0-9]{2}", "randomString").

- Cache Storage: Generated values are stored in the cache under the specified variable name.

- Regex Patterns: Use specific patterns to define the type and length of the generated value (see below).

- Supported Platforms: These commands are platform-agnostic and work in any testing context.

- Errors: Invalid regex patterns will raise an error (e.g., "Pattern not supported").

1. Generate a Random Value

Description: Generates a random value based on a regex pattern and saves it to the cache.

Syntax:

```
generate "REGEX" and save it as "VARIABLE"
```

- "REGEX": The pattern defining the value to generate (see supported patterns below).

- "VARIABLE": The name to store the generated value in the cache (e.g., "randomString").

Supported Patterns:

- "[a-zA-Z0-9]{LENGTH}": Alphanumeric string (e.g., "ab12") of specified length.

- "[a-zA-Z]{LENGTH}": Alphabetic string (e.g., "abc") of specified length.

- "\\d{LENGTH}": Numeric string (e.g., "123") of specified length.

- "\\d{4}-\\d{2}-\\d{2}{DAYS}": Date in "YYYY-MM-DD" format with days offset (e.g., "2025-03-22" for +2 days).

- "\\d{4}/\\d{2}/\\d{2}{DAYS}": Date in "YYYY/MM/DD" format with days offset (e.g., "2025/03/22" for +2 days).

- LENGTH: A number (e.g., 2, 4) for string length; if omitted, a random length up to 10 is used.

- DAYS: Positive (e.g., 2) or negative (e.g., -2) integer for days relative to today.

Examples:

```
Given generate "[a-zA-Z0-9]{5}" and save it as "randomCode"    # Generates a 5-character alphanumeric (e.g., "Xy7k9")
Given generate "\\d{3}" and save it as "randomNum"            # Generates a 3-digit number (e.g., "472")
Given generate "\\d{4}-\\d{2}-\\d{2}{2}" and save it as "futureDate"  # Generates a date 2 days from now (e.g., "2025-03-22")
Given generate "[a-zA-Z]{4}" and save it as "randomName"      # Generates a 4-letter string (e.g., "abcd")
```

Notes:

- Dates are based on the current date (e.g., today is March 20, 2025).

- Use positive DAYS for future dates, negative for past dates.

Usage Example in a Feature File

```
Scenario: Generate and use random data
  Given generate "[a-zA-Z0-9]{6}" and save it as "userId"
  Given generate "\\d{4}-\\d{2}-\\d{2}{3}" and save it as "eventDate"
  Then enter variable "userId" into "userIdField"
  Then enter variable "eventDate" into "dateField"
```

- This example generates a random user ID and a date 3 days from now, then enters them into fields.

Entering Text into UI Elements

This guide outlines the Cucumber commands available in the Nimble framework for entering text into UI elements in mobile and web applications. These commands allow you to simulate user input, such as typing into text fields, during automated testing. Each command is written in Gherkin syntax and can be used in your .feature files.

General Guidelines

- Text in Quotes: Enclose text values and element identifiers in double quotes (e.g., "username", "usernameField").

- Optional Parameters: Some commands include optional parts (e.g., exact phrase matching, variable usage). Use them only when needed.

- Supported Platforms: These commands work for both mobile and web apps, automatically adapting to the application type.

- Visibility: For web apps, the target element must be visible; hidden elements will cause an error.

- Errors: If an element isn't found or conditions aren't met, Nimble will raise an error (e.g., "Element not found" or "Operation not supported").

1. Enter Text into an Element by Text

Description: Enters a specified value into a UI element identified by its visible text, with an optional exact phrase match.

Syntax:

```
enter ["variable "]"VALUE" into "ELEMENT_TEXT" [with exact phrase]
```

- "VALUE": The text to enter (e.g., "username").

- "variable " (optional): Indicates the value is stored in a variable (retrieved from a cache). Include the word variable before the value if applicable.

- "ELEMENT_TEXT": The visible text of the element (e.g., "usernameField").

- with exact phrase (optional): Ensures the element's text matches exactly (case-sensitive).

Examples:

```
Given enter "username" into "usernameField"                # Enters "username" into an element with
text "usernameField"
Given enter "password" into "passwordField" with exact phrase  # Enters "password" with exact match
for "passwordField"
Given enter variable "user" into "loginField"              # Enters a cached value named "user" into
"loginField"
```

Notes:

- Use variable when the value comes from a previous test step (e.g., a stored result).

- The element must be present and, for web apps, visible.

2. Enter Text into the Nth Element by Text

Description: Enters a value into a specific occurrence (e.g., 1st, 2nd, 3rd) of an element identified by its text, with an optional exact phrase match.

Syntax:

```
enter ["variable "]"VALUE" into the NUMBER(st|nd|rd|th) "ELEMENT_TEXT" [with exact phrase]
```

- "VALUE": The text to enter.

- "variable " (optional): Indicates the value is from a variable.

- NUMBER: The position (e.g., 1, 2, 3).

- (st|nd|rd|th): The ordinal suffix (e.g., 1st, 2nd, 3rd, 4th).

- "ELEMENT_TEXT": The visible text of the element.

- with exact phrase (optional): Ensures an exact text match.

Examples:

```
Given enter "sample text" into the 2nd "inputField"          # Enters "sample text" into the second
"inputField"
Given enter "12345" into the 3rd "code" with exact phrase    # Enters "12345" into the third "code"
with exact match
Given enter variable "pass" into the 1st "password"          # Enters a cached "pass" value into the
first "password"
```

Notes:

- The position starts at 1 (e.g., 1st is the first occurrence).

- An error occurs if the specified position doesn't exist.

3. Enter Text Directly (No Specific Element)

Description: Enters a value directly into the application without specifying a target element, useful for scenarios like global input fields or active focus.

Syntax:

```
enter ["variable "]"VALUE"
```

- "VALUE": The text to enter.

- "variable " (optional): Indicates the value is from a variable.

Examples:

```
Given enter "username"                # Enters "username" into the active input field
Given enter variable "storedValue"    # Enters a cached value named "storedValue"
```

Notes:

- Assumes an active input field exists (e.g., one in focus).

- Useful for simple scenarios where the target is implied.

Reading UI Element Values

This guide outlines the Cucumber commands available in the Nimble framework for reading text values from UI elements in mobile and web applications and saving them to a cache. These commands allow you to capture data from the UI during automated testing for later use or verification. Each command is written in Gherkin syntax and can be used in your .feature files.

General Guidelines

- Text in Quotes: Enclose element identifiers and variable names in double quotes (e.g., "usernameField", "storedUsername").

- Cache Storage: Values are stored in a cache under the specified variable name for use in other steps.

- Supported Platforms: These commands work for both mobile and web apps, adapting automatically.

- Errors: If an element isn't found, an error will be logged or raised (e.g., "Element not found").

1. Read Value from Element by Text

Description: Reads the text value from a UI element identified by its visible text and saves it to the cache.

Syntax:

```
read value from "ELEMENT_TEXT" and save it as "VARIABLE_NAME"
```

- "ELEMENT_TEXT": The visible text of the element (e.g., "usernameField").

- "VARIABLE_NAME": The name under which the value is stored in the cache (e.g., "storedUsername").

Examples:

```
Given read value from "usernameField" and save it as "storedUsername"    # Reads text from "usernameField" and saves it
Given read value from "errorMessage" and save it as "errorText"          # Reads text from "errorMessage" and saves it
```

Notes:

- Uses exact text matching by default.

- The element must be present in the UI.

2. Read Value from Element by ID

Description: Reads the text value from a UI element identified by its ID (with an optional fallback ID and index) and saves it to the cache.

Syntax:

```
read value from the element with id as "ID1" [or "ID2"] [at index NUMBER] and save it as
"VARIABLE_NAME"
```

- "ID1": The primary ID of the element.

- or "ID2" (optional): A fallback ID if the primary ID isn't found.

- at index NUMBER (optional): The position (e.g., 1, 2) if multiple elements match the ID.

- "VARIABLE_NAME": The name under which the value is stored in the cache.

Examples:

```
Given read value from the element with id as "userId" and save it as "userValue"                #
Reads from "userId"
Given read value from the element with id as "primaryId" or "backupId" and save it as "text"  #
Tries "primaryId", then "backupId"
```

Given read value from the element with id as "itemId" at index 2 and save it as "itemText" # Reads the second "itemId"

Notes:

- For mobile, tries the primary ID first, then the fallback if provided.

- Index starts at 1 (e.g., "at index 2" means the second occurrence).

- The element must exist, or an error will be logged.

Usage Example in a Feature File

Scenario: Capture and verify user data

```
  Given read value from "usernameField" and save it as "storedUsername"
  Given read value from the element with id as "status" and save it as "userStatus"
  Then check that variable "storedUsername" is equals to "admin"
```

- This example reads values from a text field and an ID, then verifies one of them.

Scrolling and Swiping the Screen

This guide outlines the Cucumber commands available in the Nimble framework for scrolling, swiping, and interacting with sliders in mobile applications. These commands allow you to navigate the UI by moving the screen until specific elements are visible or adjusting sliders during automated testing. Each command is written in Gherkin syntax and can be used in your .feature files.

General Guidelines

- Text in Quotes: Enclose element text or identifiers in double quotes (e.g., "Load more", "volume").

- Mobile Only: These commands are designed for mobile apps and may not work in web contexts.

- Directions: Specify the direction of movement (e.g., "up", "down", "left", "right").

- Exact Matching: Use "with exact phrase" for precise text matches; omit it for partial matches.

- Errors: If an element isn't found or the action fails, an error may be raised or logged.

1. Scroll Until Text is Visible

Description: Scrolls the screen up or down until it contains the specified text, with an optional exact phrase match.

Syntax:

```
scroll (up|down) until screen contains "TEXT" [with exact phrase]
```

- (up|down): The direction to scroll.

- "TEXT": The text to search for (e.g., "Load more").

- with exact phrase (optional): Ensures an exact text match (case-sensitive).

Examples:

```
Given scroll down until screen contains "Load more"              # Scrolls down until "Load more"
appears
Given scroll up until screen contains "Header" with exact phrase  # Scrolls up until exact "Header"
appears
```

Notes:

- Continues scrolling until the text is found or the end is reached.

- Useful for lists or long screens.

2. Swipe Until Text is Visible

Description: Swipes the screen left or right until it contains the specified text, with an optional exact phrase match.

Syntax:

```
swipe (left|right) until screen contains "TEXT" [with exact phrase]
```

- **(left|right):** The direction to swipe.

- "TEXT": The text to search for (e.g., "Settings").

- with exact phrase (optional): Ensures an exact text match.

Examples:

```
Given swipe right until screen contains "Settings"              # Swipes right until "Settings"
appears
Given swipe left until screen contains "Welcome" with exact phrase  # Swipes left until exact
"Welcome" appears
```

Notes:

- Ideal for horizontal navigation (e.g., carousels or tabs).

- Stops when the text is found or the swipe limit is reached.

3. Slide a Slider by Content Description

Description: Adjusts a slider identified by its content description (accessibility label).

Syntax:

```
slide the "CONTENT_DESC" slider
```

- "CONTENT_DESC": The content description of the slider (e.g., "volume").

Examples:

```
Given slide the "volume" slider              # Adjusts the "volume" slider
Given slide the "brightness" slider          # Adjusts the "brightness" slider
```

Notes:

- Attempts to find the slider by content description first, then by ID if not found.

- Specific to mobile apps with sliders (e.g., volume or brightness controls).

Usage Example in a Feature File

Scenario: Navigate and adjust UI

```
  Given scroll down until screen contains "Load more" with exact phrase
  Given swipe right until screen contains "Options"
  Given slide the "volume" slider
```

- This example scrolls to a button, swipes to a menu, and adjusts a slider.

Pausing the Execution

This guide outlines the Cucumber commands available in the Nimble framework for pausing test execution or waiting for specific conditions in mobile and web applications. These commands allow you to introduce delays or wait for text to appear during automated testing. Each command is written in Gherkin syntax and can be used in your .feature files.

General Guidelines

- Text in Quotes: Enclose text to wait for in double quotes (e.g., "Loading").

- Time in Seconds: Specify wait durations as integers (e.g., 5 for 5 seconds).

- Supported Platforms: These commands work for both mobile and web apps, adapting automatically.

- Exact Matching: Use "with exact phrase" for precise text matches; omit it for partial matches.

- Errors: If a condition isn't met (e.g., text doesn't appear), the step may fail depending on implementation.

1. Wait for a Specified Duration

Description: Pauses the test execution for a specified number of seconds.

Syntax:

```
wait NUMBER (sec|secs)
```

- NUMBER: The number of seconds to wait (e.g., 5).

- (sec|secs): Use either "sec" or "secs" (both are accepted).

Examples:

```
Given wait 5 sec     # Pauses for 5 seconds
Given wait 10 secs   # Pauses for 10 seconds
```

Notes:

- Useful for waiting on animations, loading screens, or asynchronous actions.

2. Wait for Text to Appear

Description: Waits until the specified text appears on the screen, with an optional exact phrase match.

Syntax:

```
wait for "TEXT" [with exact phrase]
```

- "TEXT": The text to wait for (e.g., "Loading").

- with exact phrase (optional): Ensures an exact text match (case-sensitive).

Examples:

```
Given wait for "Loading"              # Waits for "Loading" (partial match allowed)
Given wait for "Success" with exact phrase  # Waits for exact "Success"
```

Notes:

- Continues waiting until the text appears or a timeout occurs (timeout depends on framework settings).

- Ideal for synchronization with dynamic UI updates.

Usage Example in a Feature File

Scenario: Wait for loading and proceed

```
  Given wait 3 sec
  Given wait for "Welcome" with exact phrase
  Then click "Continue"
```

- This example waits 3 seconds, then waits for "Welcome" to appear exactly before clicking a button.

Application Related Controls

This guide outlines the Cucumber commands available in the Nimble framework for controlling the application state in mobile and web testing. These commands allow you to relaunch the app, navigate back, or restore it from the background during automated testing. Each command is written in Gherkin syntax and can be used in your .feature files.

General Guidelines

- No Parameters: These commands do not require additional text or values beyond the step itself.

- Supported Platforms: These commands work for both mobile and web apps, adapting automatically where applicable.

- Errors: If an operation isn't supported (e.g., no previous screen to go back to), an error may be raised.

1. Relaunch the Application

Description: Restarts the application to ensure a clean state.

Syntax:

```
relaunch the app
```

Examples:

```
Given relaunch the app    # Restarts the application
```

Notes:

- Clears the current state and reloads the app from its initial screen.

- Useful for resetting between test scenarios.

2. Go Back

Description: Navigates back to the previous screen or state in the application.

Syntax:

```
go back
```

Examples:

```
Given go back    # Returns to the previous screen
```

Notes:

- Simulates pressing the back button or equivalent navigation.

- May fail if there's no previous screen (e.g., at the app's root).

3. Restore App to Foreground

Description: Brings the application from the background to the foreground.

Syntax:

```
restore the app to the foreground
```

Examples:

```
Given restore the app to the foreground    # Activates the app from the background
```

Notes:

- Requires the app to have been previously sent to the background.

- Useful for testing app behavior after being minimized.

Usage Example in a Feature File

Scenario: Reset and navigate

```
  Given relaunch the app
  Given click "Settings"
  Given go back
  Given restore the app to the foreground
```

- This example relaunches the app, navigates to Settings, goes back, and restores the app if it was minimized

Using Conditional Execution

This guide outlines the Cucumber commands available in the Nimble framework for defining and managing conditional logic in mobile and web application tests. These commands allow you to create conditions based on variables and operators, which can influence the flow of your test scenarios. Each command is written in Gherkin syntax and can be used in your .feature files.

General Guidelines

- Text in Quotes: Enclose variable names, operators, and values in double quotes (e.g., "username", "Equals", "admin").

- Stack-Based: Conditions are stored in a stack, meaning they are added (pushed) and removed (popped) in a last-in, first-out order.

- Supported Platforms: These commands are platform-agnostic and work with mobile, web, or API testing contexts.

- Operators: Use valid operator names (e.g., "Equals", "NotEquals") as defined by the framework.

- Errors: If the stack is empty when removing a condition, or if an invalid operator is used, an error will occur.

1. Open a Condition Block

Description: Defines a new condition by specifying a variable, an operator, and a value, adding it to the condition stack.

Syntax:

```
if "VARIABLE" "OPERATOR" "VALUE"
```

- "VARIABLE": The name of the variable to evaluate (e.g., "username").

- "OPERATOR": The comparison operator. Supported operators include:

- *"Equals":* Checks if the variable equals the value.

- "*NotEquals*": Checks if the variable does not equal the value.

(Additional operators may be supported; check your framework documentation.)

- "VALUE": The value to compare against (e.g., "admin").

Examples:

```
Given if "username" "Equals" "admin"          # Adds condition: username equals "admin"
Given if "status" "NotEquals" "error"         # Adds condition: status does not equal "error"
```

Notes:

- The condition is pushed onto a stack for later evaluation or control flow.

- Ensure the operator matches a valid name (case-sensitive).

2. Close a Condition Block

Description: Closes a conditional block by removing the most recent condition from the stack.

Syntax:

```
endif
```

Examples:

Given endif # Removes the last condition from the stack

Notes:

- Use this to end a conditional block started with "if".

- The stack must not be empty, or an error will occur.

Usage Example in a Feature File

Scenario: Conditional login check

```
  Given if "username" "Equals" "admin"
    And enter "admin123" into "passwordField"
  And endif
```

- This example checks if "username" is "admin" and, if true, enters a password. The "endif" closes the condition.

Database Operations

This guide outlines the Cucumber commands available in the Nimble framework for interacting with databases during automated testing. These commands allow you to connect to a database, execute queries, disconnect, and validate query results. Each command is written in Gherkin syntax and can be used in your .feature files.

General Guidelines

- Text in Quotes: Enclose database types, variable names, queries, and attributes in double quotes (e.g., "MySQL", "dbConnection", "SELECT * FROM users").

- Data Tables: Use Cucumber data tables for credentials or query parameters.

- Cache Usage: Connections and results are stored in a cache for reuse across steps.

- Supported Databases: Currently supports MySQL, PostgreSQL, and SQL Server; check your setup for others.

- Errors: Invalid database types, connection issues, or query failures will raise exceptions with details.

1. Connect to a Database

Description: Establishes a connection to a specified database type using provided credentials and saves it to the cache.

Syntax:

```
Given connect to the database "DB_TYPE" using the following credentials and save it as
"CONNECTION_VARIABLE"
  | url       | username | password |
  | URL_VALUE| USER     | PASS     |
```

- "DB_TYPE": The type of database (e.g., "MySQL", "PostgreSQL", "SQLServer").

- "CONNECTION_VARIABLE": The name to store the connection in the cache (e.g., "dbConnection").

- Data Table: Columns must be "url", "username", and "password"; one row of credentials.

Examples:

```
Given connect to the database "MySQL" using the following credentials and save it as "dbConnection"
  | url                  | username | password |
  | jdbc:mysql://localhost:3306/test | root     | pass123  |
```

Notes:

- Only one row of credentials is processed.

- Ensure the correct driver is available for the database type.

2. Disconnect a Database Connection

Description: Closes the specified database connection stored in the cache.

Syntax:

```
disconnect the database connection "CONNECTION_VARIABLE"
```

- "CONNECTION_VARIABLE": The name of the cached connection (e.g., "dbConnection").

Examples:

```
Given disconnect the database connection "dbConnection"    # Closes the "dbConnection"
```

Notes:

- Safe to call even if the connection is already closed.

3. Execute a Database Query

Description: Runs a SQL query on a cached database connection, optionally with parameters, and saves results to the cache if specified.

Syntax:

```
execute the query "QUERY" [using following parameters] on the connection "CONNECTION_VARIABLE" [and
save the results as "RESULTS_VARIABLE"]
  | KEY   | VALUE  |
```

- "QUERY": The SQL query to execute (e.g., "SELECT * FROM users").

- using following parameters (optional): Indicates parameters are provided in a data table.

- "CONNECTION_VARIABLE": The cached connection name (e.g., "dbConnection").

- and save the results as "RESULTS_VARIABLE" (optional): The name to store results in the cache (e.g., "userResults").

- Data Table (optional): Key-value pairs for query parameters (e.g., "id" | "1").

Examples:

```
Given execute the query "SELECT * FROM users" on the connection "dbConnection" and save the results
as "userResults"

Given execute the query "SELECT * FROM users WHERE id = $1" using following parameters on the
connection "dbConnection" and save the results as "userData"
  | id | 1 |
```

Notes:

- Use "$1" in the query for parameter substitution when using a data table.

- Results are stored as a list of maps (column names to values) if a results variable is provided.

4. Validate Query Result Attributes

Description: Checks if attributes in a cached query result meet a specified condition (all or at least one row).

Syntax:

```
validate that (all|at-least one) "ATTRIBUTE" attribute in the result set "RESULTS_VARIABLE"
(Contains|Equals|LessThan|GreaterThan|NotEquals) "VALUE"
```

- (all|at-least one): "all" requires all rows to match; "at-least one" requires at least one match.

- "ATTRIBUTE": The column name in the result set (e.g., "name").

- "RESULTS_VARIABLE": The cached result set name (e.g., "userResults").

- (Contains|Equals|LessThan|GreaterThan|NotEquals): The condition to check:

- Contains: Attribute includes the value.

- Equals: Attribute equals the value.

- LessThan: Attribute is less than the value (numeric).

- GreaterThan: Attribute is greater than the value (numeric).

- NotEquals: Attribute does not equal the value.

- "VALUE": The expected value (e.g., "admin").

Examples:

```
Given validate that all "name" attribute in the result set "userResults" Equals "admin"
Given validate that at-least one "age" attribute in the result set "userData" GreaterThan "25"
```

Notes:

- Fails with an assertion error if the condition isn't met, showing actual vs. expected values.

Usage Example in a Feature File

Scenario: Verify user data in database

```
  Given connect to the database "MySQL" using the following credentials and save it as
"dbConnection"
    | url                  | username | password |
    | jdbc:mysql://localhost:3306/test | root     | pass123  |
  Given execute the query "SELECT * FROM users WHERE id = $1" using following parameters on the
connection "dbConnection" and save the results as "userResults"
    | id | 1 |
  Given validate that all "username" attribute in the result set "userResults" Equals "admin"
  Given disconnect the database connection "dbConnection"
```

- This example connects to a MySQL database, queries a user, validates the username, and disconnects.

Publishing and Consumng Kafka Events

This guide outlines the Cucumber commands available in the Nimble framework for interacting with Kafka brokers during automated testing. These commands allow you to connect to a Kafka broker, publish events to topics, and consume events from topics with validation. Each command is written in Gherkin syntax and can be used in your .feature files.

General Guidelines

- Text in Quotes: Enclose broker URLs, credentials, topic names, regex patterns, and variable names in double quotes (e.g., "kafka.example.com:9092", "userEvent").

- Cache Usage: Connections and event payloads are stored in the cache for reuse.

- Parameters: Optional parameters (e.g., group ID, additional settings) can be included as needed.

- Time in Seconds: Specify poll times for consuming events as integers (e.g., 30).

- Errors: Invalid configurations, missing connections, or unmatched events will raise exceptions or assertions.

1. Connect to a Kafka Broker

Description: Establishes a connection to a Kafka broker as a producer or consumer, using credentials, and saves it to the cache.

Syntax:

```
connect to the Kafka broker "BROKER_URL" using "USERNAME" and "PASSWORD" as (producer|consumer)
[with "GROUP_ID"] and save it as "VARIABLE"
```

- "BROKER_URL": The Kafka broker address (e.g., "kafka.example.com:9092").

- "USERNAME" and "PASSWORD": Authentication credentials.

- (producer|consumer): Specifies the connection type.

- with "GROUP_ID" (optional): Consumer group ID; if omitted, a random ID is generated.

- "VARIABLE": The name to store the connection in the cache (e.g., "kafkaConnection1").

Examples:

```
Given connect to the Kafka broker "kafka.example.com:9092" using "user" and "pass" as producer and
save it as "kafkaProducer1"

Given connect to the Kafka broker "kafka.example.com:9092" using "user" and "pass" as consumer with
"group1" and save it as "kafkaConsumer1"
```

Notes:

- Uses SASL/PLAIN authentication; ensure credentials are valid.

- Consumer group ID is optional but recommended for specific consumer behavior.

2. Publish an Event to a Kafka Topic

Description: Sends an event with a key and value to a specified Kafka topic using a cached producer connection.

Syntax:

```
publish an event with key "EVENT_KEY" and value "EVENT_VALUE" to topic "TOPIC_NAME" using "VARIABLE"
[with additional parameters like "PARAMS"]
```

- "EVENT_KEY": The key for the event (e.g., "userId123").

- "EVENT_VALUE": The value of the event (e.g., "userDetails").

- "TOPIC_NAME": The target Kafka topic (e.g., "user-topic").

- "VARIABLE": The cached producer connection name (e.g., "kafkaProducer1").

- with additional parameters like "PARAMS" (optional): Key-value pairs for extra settings (currently unused in the code).

Examples:

```
Given publish an event with key "userId123" and value "userDetails" to topic "user-topic" using
"kafkaProducer1"
```

Notes:

- Supports variable substitution in "EVENT_VALUE" using "$1" and a params map (if provided).

- Logs success or failure of the send operation.

3. Consume an Event from a Kafka Topic

Description: Consumes an event from a specified topic, validates it against a regex pattern, and optionally saves the payload or closes the connection.

Syntax:

```
consume an event from topic "TOPIC_NAME" that matches "REGEX" [and save it as "PAYLOAD_VARIABLE"]
within NUMBER secs using "VARIABLE" [and close connection]
```

- "TOPIC_NAME": The Kafka topic to consume from (e.g., "user-topic").

- "REGEX": The regular expression to match the event payload (e.g., ".*userId.*").

- and save it as "PAYLOAD_VARIABLE" (optional): The name to store the matched payload in the cache (e.g., "userEvent").

- NUMBER: Maximum poll time in seconds (e.g., 30).

- "VARIABLE": The cached consumer connection name (e.g., "kafkaConsumer1").

- and close connection (optional): Closes the consumer after consuming.

Examples:

```
Given consume an event from topic "user-topic" that matches ".*userId.*" and save it as "userEvent"
within 30 secs using "kafkaConsumer1" and close connection

Given consume an event from topic "order-topic" that matches ".*orderId.*" within 60 secs using
"orderConsumer"
```

Notes:

- Subscribes to the topic and polls for up to the specified time.

- Supports variable substitution in "REGEX" using "$1" and a params map (if provided).

- Fails if no matching event is found within the time limit.

Usage Example in a Feature File

Scenario: Publish and consume a Kafka event

```
 Given connect to the Kafka broker "kafka.example.com:9092" using "user" and "pass" as producer and
save it as "kafkaProducer1"
  Given connect to the Kafka broker "kafka.example.com:9092" using "user" and "pass" as consumer
with "group1" and save it as "kafkaConsumer1"
  Given publish an event with key "userId123" and value "userDetails123" to topic "user-topic" using
"kafkaProducer1"
  Given consume an event from topic "user-topic" that matches ".*userDetails.*" and save it as
"userEvent" within 30 secs using "kafkaConsumer1" and close connection
  Then check that variable "userEvent" contains "userDetails123"
- This example connects as a producer and consumer, publishes an event, consumes it with validation,
and verifies the payload.
```

Writing Re-usable Script

This guide outlines the Cucumber commands available in the Nimble framework for creating and executing reusable subroutines in automated testing. These commands allow you to define a set of steps as a subroutine and invoke them later by name, promoting reusability across test scenarios. Each command is written in Gherkin syntax and can be used in your .feature files.

General Guidelines

- Text in Quotes: Enclose subroutine names in double quotes (e.g., "loginFlow").

- DocString: Use triple quotes (""") to define multi-line steps for subroutines.

- Supported Platforms: These commands are platform-agnostic and work in any testing context.

- Step Syntax: Subroutine steps must start with "Given", "When", "Then", or "And" and follow valid Cucumber syntax.

- Errors: Attempting to execute a non-existent subroutine will raise an error.

1. Create a Subroutine

Description: Defines a reusable subroutine with a name and a set of steps provided in a DocString.

Syntax:

Given create sub-routine "SUBROUTINE_NAME" with the following steps:

"""

STEP1

STEP2

...

"""

- "SUBROUTINE_NAME": The name of the subroutine (e.g., "loginFlow").

- DocString: A block of steps enclosed in triple quotes, each starting with "Given", "When", "Then", or "And".

Examples:

Given create sub-routine "loginFlow" with the following steps:

"""

Given navigate to the login page

And enter "testUser" into "usernameField"

And enter "testPass" into "passwordField"

And click "loginButton"

"""

Notes:

- Steps are stored in a registry for later use.

- Each step must be a valid Cucumber step recognized by the framework.

2. Execute a Subroutine

Description: Runs a previously defined subroutine by its name.

Syntax:

```
execute sub-routine "SUBROUTINE_NAME"
```

- "SUBROUTINE_NAME": The name of the subroutine to execute (e.g., "loginFlow").

Examples:

```
Given execute sub-routine "loginFlow"    # Executes the "loginFlow" subroutine
```

Notes:

- Fails with an error if the subroutine doesn’t exist.

- Executes steps in the order defined, using the glue packages from the TestRunner class.

Usage Example in a Feature File

Scenario: Use subroutine for login

```
 Given create sub-routine "loginFlow" with the following steps:
    """
    Given navigate to the login page
    And enter "testUser" into "usernameField"
    And enter "testPass" into "passwordField"
    And click "loginButton"
    """
  Given execute sub-routine "loginFlow"
  Then check that screen contains "Welcome"
- This example defines a login subroutine and executes it, then verifies the result.
```

API Testing

This guide outlines the Cucumber commands available in the Nimble framework for composing, configuring, and executing API requests during automated testing. These commands allow you to prepare an HTTP request (GET, POST, PUT, PATCH, DELETE), add a JSON payload, and save the response for later use. Each command is written in Gherkin syntax and can be used in your .feature files.

General Guidelines

- Text in Quotes: Enclose HTTP methods, URLs, and variable names in double quotes (e.g., "post", "https://example.com", "response").

- Data Tables: Use Cucumber data tables to specify headers.

- DocString: Use triple quotes (""") to define JSON payloads.

- Sequence: Steps must be used in order: compose the request, add the payload (if needed), then execute.

- Cache Usage: Responses are stored in the cache under the specified variable name.

- Errors: Missing prerequisites (e.g., no URL) or invalid parameters will raise exceptions.

1. Compose an API Request

Description: Prepares an API request with a specified HTTP method, URL, and headers.

Syntax:

```
compose a (get|post|put|patch|delete) request to "URL" with headers
  | HEADER_NAME | HEADER_VALUE |
```

- (get|post|put|patch|delete): The HTTP method to use.

- "URL": The target endpoint (e.g., "https://example.com/api").

- Data Table: Key-value pairs for headers (e.g., "Content-Type" | "application/json").

Examples:

```
When compose a post request to "https://example.com/onboarding/v2/devices" with headers
  | Content-Type   | application/json          |
  | X-Request-Id   | Ahb-9888                 |
  | x-api-key      | d8d4a69e-beb6-4878-be50-ee3455fc09f9 |
```

Notes:

- Supports variable substitution in the URL using "$1" and params (if provided).

- Headers can include cached values if prefixed with "header:" (e.g., "header:X-Token").

2. Add a JSON Request Payload

Description: Attaches a JSON payload to the prepared API request.

Syntax:

```
with the below JSON request
  """
  JSON_PAYLOAD
  """
```

- JSON_PAYLOAD: The request body in JSON format, enclosed in triple quotes.

Examples:

```
And with the below JSON request
  """
  {
    "deviceUniqueId": "26734tyughrehwjfdsyiyutghfds",
    "deviceHash": "2ytu3gjh4rweklifuydjcgh",
    "deviceId": "879iyujhknbdshjyuughjas"
  }
  """
```

Notes:

- Requires a prior "compose" step to set the URL, method, and headers.

- Must be valid JSON syntax.

3. Execute the API Request and Save Response

Description: Sends the prepared API request and stores the response in the cache.

Syntax:

```
execute and save the response as "VARIABLE"
```

- "VARIABLE": The name to store the response in the cache (e.g., "response").

Examples:

```
And execute and save the response as "response"    # Sends the request and saves the response
```

Notes:

- Uses the method, URL, headers, and payload from previous steps.

- Response can be validated later (e.g., with CheckSteps).

Usage Example in a Feature File

Scenario: Validate API call

```
When compose a post request to "https://example.com/onboarding/v2/devices" with headers
   | Content-Type    | application/json          |
   | X-Request-Id    | Ahb-9888                  |
   | x-api-key       | d8d4a69e-beb6-4878-be50-ee3455fc09f9 |
 And with the below JSON request
   """
   {
     "deviceUniqueId": "26734tyughrehwjfdsyiyutghfds",
     "deviceHash": "2ytu3gjh4rweklifuydjcgh",
     "deviceId": "879iyujhknbdshjyuughjas"
   }
   """
 And execute and save the response as "deviceResponse"
 Then check that response "deviceResponse" has "200" as status code
```

- This example composes a POST request, adds a JSON payload, executes it, and verifies the status code.

Assertions

This guide outlines the Cucumber commands available in the Nimble framework for verifying conditions in mobile and web applications, as well as API responses and cached variables. These commands allow you to check the presence of text, button states, variable values, HTTP status codes, and JSON data during automated testing. Each command is written in Gherkin syntax and can be used in your .feature files.

General Guidelines

- Text in Quotes: Enclose text values, variable names, and identifiers in double quotes (e.g., "Welcome Message", "username").

- Optional Parameters: Some commands include optional parts (e.g., exact phrase matching, variable usage). Use them only when needed.

- Supported Platforms: Most commands work for both mobile and web apps, adapting automatically; some are specific to API or variable checks.

- Tables: Commands with lists (e.g., multiple text values) use Cucumber data tables.

- Assertions: Failures (e.g., element not found, condition not met) will raise an error with details for debugging.

1. Check Text Presence on Screen

Description: Verifies whether the screen contains or does not contain specific text, with an optional exact phrase match.

Syntax:

```
check that screen (contains|does not contain) ["variable "]"TEXT" [with exact phrase]
```

- (contains|does not contain): Choose "contains" to check presence or "does not contain" to check absence.

- "variable " (optional): Indicates the text is a variable name whose value is retrieved from cache.

- "TEXT": The text to check (e.g., "Welcome Message").

- with exact phrase (optional): Ensures an exact text match (case-sensitive).

Examples:

```
Given check that screen contains "Welcome Message"                # Checks if "Welcome Message" is
present

Given check that screen does not contain "Error"                 # Checks if "Error" is absent

Given check that screen contains variable "greeting" with exact phrase  # Checks a cached value with
exact match
```

Notes:

- Use "variable" when checking a value stored earlier in the test.

2. Check Multiple Texts on Screen

Description: Verifies whether the screen contains or does not contain a list of text values, using a data table.

Syntax:

```
check that screen (contains|does not contain) following
  | TEXT1 |
  | TEXT2 |
  | ...   |
```

- (contains|does not contain): Choose "contains" or "does not contain".

- TEXT1, TEXT2, etc.: List of text values to check.

Examples:

```
Given check that screen contains following
  | Element 1 |
  | Element 2 |
  | Element 3 |

Given check that screen does not contain following
  | Error 1 |
  | Error 2 |
```

Notes:

- Each text is checked individually; all must pass for the step to succeed.

- Uses exact matching by default.

3. Check Button State

Description: Verifies whether a button with specific text is enabled or disabled, with an optional exact phrase match.

Syntax:

```
check that button "TEXT" is (enabled|disabled) [with exact phrase]
```

- "TEXT": The button's visible text (e.g., "Submit").

- (enabled|disabled): The expected state.

- with exact phrase (optional): Ensures an exact text match.

Examples:

```
Given check that button "Submit" is enabled              # Checks if "Submit" button is clickable
Given check that button "Cancel" is disabled with exact phrase  # Checks if "Cancel" is not
clickable
```

Notes:

- Applies to buttons only; other elements may not support this check.

4. Check Variable Contains Exact Phrase from List

Description: Verifies that a cached variable's value exactly matches one of a list of phrases.

Syntax:

```
check that variable "VARIABLE" contains either of the following with exact phrase
  | VALUE1 |
  | VALUE2 |
  | ...    |
```

- "VARIABLE": The variable name in the cache.

- VALUE1, VALUE2, etc.: List of expected exact phrases.

Examples:

```
Given check that variable "username" contains either of the following with exact phrase
  | Alice |
  | Bob   |
```

Notes:

- Trims whitespace from values; must match exactly (e.g., "Alice" ≠ "alice").

5. Check Variable Contains Any Value from List

Description: Verifies that a cached variable's value contains at least one value from a list (partial match).

Syntax:

```
check that variable "VARIABLE" contains either of the following
  | VALUE1 |
  | VALUE2 |
  | ...    |
```

- "VARIABLE": The variable name in the cache.

- VALUE1, VALUE2, etc.: List of values to check for.

Examples:

```
Given check that variable "message" contains either of the following
  | Success |
  | Complete|
```

Notes:

- Checks for substring matches (e.g., "Success" in "Operation Success").

6. Check Variable with Comparison

Description: Validates a cached variable against an expected value using various comparison types.

Syntax:

```
check that variable "VARIABLE" (contains|is numerically less than|is numerically greater than|is
equals to|is not equals to) "VALUE"
```

- "VARIABLE": The variable name in the cache.

- Comparison types:

  - contains: Checks if VALUE is a substring.

  - is equals to: Checks exact equality (numeric or text).

  - is not equals to: Checks inequality.

  - is numerically less than: Compares numbers (less than).

  - is numerically greater than: Compares numbers (greater than).

- "VALUE": The expected value.

Examples:

```
Given check that variable "status" is equals to "completed"          # Checks exact match
Given check that variable "price" is numerically less than "5000"    # Checks number < 5000
Given check that variable "error" contains "failed"                  # Checks substring
```

Notes:

- Numeric comparisons require valid numbers (e.g., "1,234.56" is supported).

7. Check HTTP Response Status Code

Description: Verifies that a cached HTTP response has the expected status code.

Syntax:

```
check that response "VARIABLE" has "STATUS" as status code
```

- "VARIABLE": The variable name storing the response in the cache.

- "STATUS": The expected status code (e.g., "200").

Examples:

```
Given check that response "apiResult" has "200" as status code    # Checks for 200 OK
```

Notes:

- Response must be cached from a prior API call.

8. Check API Response JSON Value

Description: Verifies that a cached API response contains an expected value at a specific JSON path.

Syntax:

```
check that API response "VARIABLE" contains "VALUE" at "JSONPATH"
```

- "VARIABLE": The variable name storing the response in the cache.

- "VALUE": The expected value.

- "JSONPATH": The JSON path (e.g., "$.data.name").

Examples:

```
Given check that API response "userData" contains "Alice" at "$.data.name"  # Checks JSON value
```

Notes:

- Uses JSONPath format; response must be valid JSON.

Date Manipulation

This guide outlines the Cucumber commands available in the Nimble framework for manipulating dates during automated testing. These commands allow you to calculate differences between dates, format dates, and adjust dates by specific time units, storing the results in a cache. Each command is written in Gherkin syntax and can be used in your .feature files.

General Guidelines

- Text in Quotes: Enclose dates, formats, and variable names in double quotes (e.g., "2025-03-20", "MM/dd/yyyy", "dateDiff").

- Cache Usage: Input dates must be in the cache for some steps; results are stored in the cache.

- Date Format: Dates must be in "yyyy-MM-dd" (ISO format) unless otherwise specified.

- Supported Platforms: These commands are platform-agnostic and work in any testing context.

- Errors: Invalid date formats, units, or missing cached values will raise exceptions.

1. Calculate Days Between Two Dates

Description: Computes the number of days between two dates and stores the result in the cache.

Syntax:

```
calculate days between "START_DATE" and "END_DATE" and store in "VARIABLE"
```

- "START_DATE": The earlier date in "yyyy-MM-dd" format (e.g., "2025-03-20").

- "END_DATE": The later date in "yyyy-MM-dd" format (e.g., "2025-03-25").

- "VARIABLE": The name to store the difference in days (e.g., "dateDiff").

Examples:

```
Given calculate days between "2025-03-20" and "2025-03-25" and store in "dateDiff"    # Stores "5"
```

Notes:

- Result is a string representing the number of days (e.g., "5").

- Negative values are possible if START_DATE is after END_DATE.

2. Format a Date

Description: Formats a cached date according to a specified pattern and stores it in the cache.

Syntax:

```
save "INPUT_DATE" as "FORMAT" in "VARIABLE"
```

- "INPUT_DATE": The cached variable name containing the date in "yyyy-MM-dd" format (e.g., "startDate").

- "FORMAT": The target format using DateTimeFormatter patterns (e.g., "MM/dd/yyyy").

- **"VARIABLE":** The name to store the formatted date (e.g., "formattedDate").

Examples:

```
Given save "startDate" as "MM/dd/yyyy" in "formattedDate"    # Formats "2025-03-20" as "03/20/2025"
```

Notes:

- INPUT_DATE must be a cached value in "yyyy-MM-dd" format.

- Common formats: "yyyy-MM-dd" (default), "MM/dd/yyyy", "dd-MMM-yyyy" (e.g., "20-Mar-2025").

3. Adjust a Date

Description: Adds or subtracts a specified number of days, weeks, months, or years from a cached date and stores the result.

Syntax:

```
adjust NUMBER (day|week|month|year|days|weeks|months|years) (to|from) "INPUT_DATE" and save as "VARIABLE"
```

- **NUMBER**: The integer amount to adjust (e.g., 5).

- **(day|week|month|year|days|weeks|months|years):** The time unit (singular or plural).

- **(to|from):** "to" adds time (future); "from" subtracts time (past).

- **"INPUT_DATE":** The cached variable name containing the date in "yyyy-MM-dd" format (e.g., "startDate").

- **"VARIABLE":** The name to store the adjusted date (e.g., "newDate").

Examples:

```
Given adjust 5 days to "startDate" and save as "newDate"    # Adds 5 days (e.g., "2025-03-25" from "2025-03-20")

Given adjust 2 months from "endDate" and save as "prevDate"  # Subtracts 2 months (e.g., "2025-01-25" from "2025-03-25")
```

Notes:

- INPUT_DATE must be a cached value in "yyyy-MM-dd" format.

- Result is stored in "yyyy-MM-dd" format.

- Invalid units (e.g., "hours") will raise an error.

Usage Example in a Feature File

Scenario: Manipulate and verify dates

```
Given generate "\\d{4}-\\d{2}-\\d{2}{0}" and save it as "startDate"  # "2025-03-20"
Given adjust 5 days to "startDate" and save as "endDate"             # "2025-03-25"
Given calculate days between "startDate" and "endDate" and store in "dateDiff"  # "5"
Given save "startDate" as "MM/dd/yyyy" in "formattedStart"          # "03/20/2025"
Then check that variable "dateDiff" is equals to "5"
```

- This example generates today's date, adjusts it, calculates the difference, formats it, and verifies the result.

String Manipulation

This guide outlines the Cucumber commands available in the Nimble framework for extracting substrings from cached strings during automated testing. These commands allow you to extract parts of a string by position or regex pattern and save the results in the cache. Each command is written in Gherkin syntax and can be used in your .feature files.

General Guidelines

- Text in Quotes: Enclose variable names and regex patterns in double quotes (e.g., "sourceString", "\\d{4}-\\d{2}-\\d{2}").

- Cache Usage: Source strings must be stored in the cache; extracted results are saved to the cache.

- Position-Based: Uses zero-based indexing for start and end positions (e.g., 0 is the first character).

- Regex: Uses Java regex syntax for pattern matching.

- Errors: Invalid positions, missing cached values, or unmatched regex patterns will raise exceptions.

1. Extract Substring by Position

Description: Extracts a substring from a cached string using specified start and end positions and saves it to the cache.

Syntax:

```
extract substring from "SOURCE_STRING" starting at position START and ending at position END and
save it as "NEW_VARIABLE"
```

- "SOURCE_STRING": The source string from substring has to be extracted.

- START: The starting position (inclusive, zero-based) of the substring (e.g., 5).

- END: The ending position (exclusive) of the substring (e.g., 10).

- "NEW_VARIABLE": The name to store the extracted substring (e.g., "substringResult").

Examples:

```
extract substring from "sourceString" starting at position 5 and ending at position 10 and save it
as "substringResult"
```

If "sourceString" is "HelloWorld123", extracts "World" (positions 5 to 9)

Notes:

- Positions are zero-based (e.g., 0 is the first character).

- Fails if START < 0, END > string length, or START > END.

2. Extract Substring by Regex Pattern

Description: Extracts the first substring from a cached string that matches a regex pattern and saves it to the cache.

Syntax:

```
extract substring from "SOURCE_VARIABLE" using the pattern "REGEX" and save it as "NEW_VARIABLE"
```

- "SOURCE_VARIABLE": The cached variable name containing the source string (e.g., "sourceString").

- "REGEX": The regular expression pattern to match (e.g., "\\d{4}-\\d{2}-\\d{2}").

- "NEW_VARIABLE": The name to store the extracted substring (e.g., "dateExtracted").

Examples:

```
extract substring from "sourceString" using the pattern "\\d{4}-\\d{2}-\\d{2}" and save it as
"dateExtracted"
```

If "sourceString" is "Event on 2025-03-20 at noon", extracts "2025-03-20"

Notes:

- Uses Java regex syntax (e.g., "\\d" for digits, "." for any character).

- Extracts the first match; fails if no match is found.

Usage Example in a Feature File

Scenario: Extract and validate string data

```
Given generate "[a-zA-Z0-9]{10}" and save it as "sourceString"  # e.g., "Ab12Cd34Ef"
Given extract substring from "sourceString" starting at position 2 and ending at position 6 and save
it as "extractedPart"  # e.g., "12Cd"
Given extract substring from "sourceString" using the pattern "\\d+" and save it as
"numberExtracted"  # e.g., "12"
Then check that variable "extractedPart" contains "12"
Then check that variable "numberExtracted" is equals to "12"
```

- This example generates a random string, extracts parts by position and regex, and verifies the results.

System-Defined Date and Time Variables

This guide describes the system-defined variables automatically added to the cache by the Nimble framework during test initialization. These variables provide current date and time information in various formats and can be used in your Cucumber .feature files without explicit generation. They are populated via an internal method and stored in the cache for immediate access.

General Guidelines

- No User Action Required: These variables are pre-loaded into the cache; you don’t need to define them.

- Cache Access: Use these variable names directly in steps that support cached variables (e.g., "variable" syntax in CheckSteps, EnterSteps).

- Formats: Dates and times follow standard Java formatting (e.g., "yyyy-MM-dd" for dates, "HH" for 24-hour time).

- Current Date: Values reflect the system date and time when the test runs (e.g., today is March 20, 2025, for this documentation).

- Supported Platforms: These variables are platform-agnostic and available in any testing context.

The following variables are automatically added to the cache during initialization:

1. "PLATFORM"

Description: The testing platform type set via the ICustomHooks setPlatform method.

Value Example: "mobile" or "web"

2. "PLATFORM_NAME"

Description: The specific platform name set via the ICustomHooks setPlatformName method.

Value Examples: "ANDROID", "IOS", "CHROME", "SAFARI", "EDGE", "FIREFOX"

3. "DEVICE_NAME"

Description: Device name of the mobile device

4. Date and Time Variables

The following date/time variables are added:

Variable	Description	Value Example
`"todays-date"`	Today's date in "yyyy-MM-dd" format.	"2025-03-20"
`"todays-datetime"`	Today's date and time in "yyyy-MM-dd hh:mm:ss SSS a" format (12-hour with milliseconds and AM/PM).	"2025-03-20 02:45:30 123 PM"
`"todays-day"`	Day of the month (1-31).	20
`"todays-month"`	Month number (1-12).	3
`"todays-month-in-words"`	Full month name in English.	March
`"todays-month-in-short"`	Three-letter month abbreviation in English.	Mar
`"todays-year"`	Current year in "yyyy" format.	2025
`"todays-weekday"`	Full weekday name in English.	Thursday
`"current-hour"`	Current hour in 12-hour format (01-12).	02
`"current-hour-in-24h"`	Current hour in 24-hour format (00-23).	14
`"current-minute"`	Current minute (0-59).	45
`"current-second"`	Current second (0-59).	30
`"current-millisecond"`	Current millisecond of the second (0-999).	123
`"current-epoch-seconds"`	Unix timestamp in seconds since January 1, 1970 (UTC).	1742527530
`"current-epoch-milliseconds"`	Unix timestamp in milliseconds since January 1, 1970 (UTC).	1742527530123
`"is-leap-year"`	Whether the current year is a leap year (true/false).	false (2025 is not a leap year)
`"week-of-year"`	Current week number of the year (1-53, ISO week numbering).	12
`"day-of-year"`	Day number of the year (1-365 or 366 in leap years).	79
`"am-pm-indicator"`	AM or PM based on the current time.	PM

Usage Example in a Feature File

Scenario: Use system variables in a test

```
Given enter variable "todays-date" into "dateField"          # Enters "2025-03-20"
Given enter variable "platformName" into "platformField"    # Enters "ANDROID"
Then check that variable "todays-month-in-words" is equals to "March"
Then check that variable "env" is equals to "prod"          # Assumes "env" from custom hooks
```

- This example uses pre-populated system variables for date and environment settings.