Flow tests

Package: `e2e/src/test/java/eu/nets/test/flows`

This package contains flow test classes, organized by functionality. Each flow is coupled with one or multiple test data classes, that can be found in `e2e/src/test/java/eu/nets/test/data`. The `data` package also includes data model classes (`models` package) and data classes shared among different tests (`shared` package).

Data (package `eu.nets.test.flows.data`)

Add here folders containing test-specific data for parametized tests. For naming consistency, folder names should match flow test folder names. Test data must be encapsulated in classes. These classes must contain methods returning `Stream<Arguments>` objects, required by JUnit `@MethodSource` to inject input data into parametized tests.

Data Models (package `eu.nets.test.flows.data.models`)

`record` classes modeling user and app data.

Shared Data (package `eu.nets.test.flows.data.shared`)

Classes listing common data used across multiple tests.

Identifying and Mapping UI elements via Appium Inspector and MpaWidget

Please refer to the project README for instructions on how to run Appium Inspector.

When an element has been identified via Appium inspector, add a new `MpaWidget` entry and use its fields to map it:

- `androidType`: the type of the Android UI element (e.g., `android.widget.Button`). If a short name is provided (e.g, `Button`), it is automatically prefixed with `android.widget.`.
- `androidId`: the resource ID of the Android UI element.
- `androidResourceId`: the full Android resource identifier in the format `appId:id/resourceId`. Constructed at runtime based on `appOrBundleId` and

`androidId`.

- `iosType`: the type of the iOS UI element (e.g., `XCUIElementTypeButton`), derived from the provided short type name. Currently, it does not support full type names.
- `iosAccessibilityId`: the `accessibilityId` of the iOS UI element. Used to locate the element via Appium directly or through XPath. It corresponds to the `name` attribute of the iOS element.
- `iosValue`: the `value` attribute of the iOS element, typically used for element lookup via XPath.
- `iosLabel`: the `label` attribute of the iOS element, typically used for element lookup via XPath.
- `appOrBundleId`: the Android application ID (package name) or the iOS bundle ID. Used for constructing full resource identifiers.
- `lokaliseKey`: the key used to retrieve the element's localized text from Lokalise.
- `enText`: the English text associated with the widget, if applicable, typically used as a default string for matching or verification.
- `color`: an optional `Color` object that may be used for color comparison during visual assertions or UI validation.
- `img`: an optional image path associated with the widget, used for image-based UI verification or screenshot comparison.

Writing a flow test

Flow tests (i.e., end-to-end tests) use AbstracFlow as a common structure/template, JUnit for definying and launching tests (and related test data) on the JVM, Allure for reporting and logging.

Main steps to create a new flow test

1. Create a new subclass which extends `AbstractFlow.` The class name must end with `Flow` (not required by the framework, but important for keeping the naming convention consistent across the repository).
2. Annotate the subclasses with `@TestInstance(TestInstance.Lifecycle.PER_CLASS)` to enable usage of non-static `@BeforeAll` and `@AfterAll` JUnit methods.

3. Annotate the subclasses `@ExtendWith(AllureJunit5.class)` on subclasses to generate a clean Allure report for each test run, especially when combining results under `build/allure-results`.
4. Define the test method as `runTest`. Annotate it with `@ParameterizedTest` + `@MethodSource` if your test accepts input data, otherwise use `@Test`. This method must be a wrapper which must call the `run` method (see 6.).
5. [Optional] Use `@Epic`, `@Feature`, `@Story`, `@Description` to add useful descriptions or links to the test.
6. Define a `run` method, not annotated with any JUnit annotations, which must contain the test code. Separating the test code from the JUnit test method allows to either run the test via JUnit or use it as a step within another test, thus making the tests modular and reusable.

Guidelines

- Use methods provided by `MpaDriver` implementations to interact with elements in a platform-specific way, scroll or swipe, manage app installation, etc.
- Override `startupAndroidSnapshot()` if your test requires booting from a specific state. In this way, the registration steps will be skipped and the test will be more efficient. When running the test on iOS, registration flows must always be called before performing test steps, as iOS does not support snapshots.
- Use `LokaliseUtil.getBundle()` to retrieve and verify localized strings, if needed.
- Add Allure steps using `Allure.step("description", () -> { ... })`. These steps will be added to the Allure report generated automatically after each test run.
- Within Allure steps, add Allure logs using `AllureUtil`.
