# Narzędzie do tworzenia testów *e2e* dla widżetów

## Cel testu

- **forWidget** - definiuje, dla jakiego całego widżetu ma zostać wykonany test
- **forComponent** - definiuje, dla jakiego elementu z atrybutem `data-component` (komponent stworzony w ramach projektu) oraz w jakim widżecie ma zostać wykonany test
- **forElement** - definiuje, dla jakiego elementu z atrybutem `data-testid` oraz w jakim widżecie ma zostać wykonany test

## Typ testu

- **forClients** - dla jakiego klienta
- **forViewports** - na jaką rozdzielczość
- **forColorSchemes** - na jaki schemat kolorystyczny

## Konfiguracja środowiska/wykonania testu

- **withWidgetProps** - parametry widgetu
- **withRouteMock** - konfiguracja odpowiedzi zapytań API
- **withWaitFor** - oczekiwanie testu na szczególne typy wydarzeń (np. załadowanie wykresu)
- **setPageInteraction** - interakcja z widgetem z pomocą obiektu `page` playwrighta

## Pozostałe

- **only** - wykonanie wyłącznie testu z danego pliku
- **withTitle** - ustawienie przedrostka do nazwy testu (reszta jest generowana automatycznie)

## Przykładowe nazwy wygenerowanych obrazów

### Dla widgetu `widget1`:

- `widget1-client1-desktop-chromium.png`
- `widget1-client1-desktop-firefox.png`
- `widget1-client1-desktop-loading-chromium.png`
- `widget1-client1-desktop-loading-firefox.png`
- `widget1-client1-desktop-loading-webkit.png`
- `widget1-client1-desktop-no-data-chromium.png`
- `widget1-client1-desktop-no-data-firefox.png`
- `widget1-client1-desktop-no-data-webkit.png`
- `widget1-client1-desktop-webkit.png`
- `widget1-client1-mobile-chromium.png`
- `widget1-client1-mobile-firefox.png`
- `widget1-client1-mobile-loading-chromium.png`
- `widget1-client1-mobile-loading-firefox.png`
- `widget1-client1-mobile-loading-webkit.png`

- `widget1-client1-mobile-no-data-chromium.png`
- `widget1-client1-mobile-no-data-firefox.png`
- `widget1-client1-mobile-no-data-webkit.png`
- `widget1-v-mobile-webkit.png`

**Dla komponentu `component1`:**

- `component1-client1-desktop-variant2-chromium.png`
- `component1-client1-desktop-variant2-firefox.png`
- `component1-client1-desktop-variant2-webkit.png`
- `component1-client1-desktop-variant1-chromium.png`
- `component1-client1-desktop-variant1-firefox.png`
- `component1-client1-desktop-variant1-webkit.png`
- `component1-client1-mobile-variant2-chromium.png`
- `component1-client1-mobile-variant2-firefox.png`
- `component1-client1-mobile-variant2-webkit.png`
- `component1-client1-mobile-variant1-chromium.png`
- `component1-client1-mobile-variant1-firefox.png`
- `component1-client1-mobile-variant1-webkit.png`
- `component1-client2-desktop-variant2-chromium.png`
- `component1-client2-desktop-variant2-firefox.png`
- `component1-client2-desktop-variant2-webkit.png`
- `component1-client2-desktop-variant1-chromium.png`
- `component1-client2-desktop-variant1-firefox.png`
- `component1-client2-desktop-variant1-webkit.png`
- `component1-client2-mobile-variant2-chromium.png`
- `component1-client2-mobile-variant2-firefox.png`
- `component1-client2-mobile-variant2-webkit.png`
- `component1-client2-mobile-variant1-chromium.png`
- `component1-client2-mobile-variant1-firefox.png`
- `component1-client2-mobile-variant1-webkit.png`

**Dla elementu `element1` w widgecie `widget2`:**

- `widget2-client1-desktop-element1-chromium.png`
- `widget2-client1-desktop-element1-firefox.png`
- `widget2-client1-desktop-element1-webkit.png`
- `widget2-client1-mobile-element1-chromium.png`
- `widget2-client1-mobile-element1-firefox.png`
- `widget2-client1-mobile-element1-webkit.png`

## Kod programu

```javascript
import { test as t, expect } from "@playwright/test";

const DEFAULT_API_URL = "https://widgets.api.pl/test/api/equities/widgets";

/**
 * @typedef {Object} ViewPortResolution
 * @property {number} width
 * @property {number} height
 */

class Builder {
  /** @type {string|null} */
  #widgetId = null;

  /** @type {string[]} */
  #clients = ["default_client"];

  /** @type {('desktop' | 'mobile')[]} */
  #viewport = ["desktop", "mobile"];

  /** @type {('default' | 'dark')[]} */
  #colorSchemes = ["default"];

  /** @type {string|null} */
  #componentName = null;

  /** @type {Object|null} */
  #widgetProps = null;

  /** @type {Array<Object>} */
  #apiMocks = [];

  /** @type {function|null} */
  #pageInteraction = null;

  /** @type {boolean} */
  #onlyThis = false;

  /** @type {Array<"canvas" | "timeout">} */
  #waitFor = [];

  /** @type {string} */
  #widgetState = "default";

  /** @type {string|null} */
  #elementTestId = null;

  /** @type {string|null} */
  #title = null;

  /** @type {Object<string, ViewPortResolution>} */
  #viewPortResolution = {
    desktop: {
      width: 1396,
      height: 480,
    },
    mobile: {
      width: 600,
      height: 480,
    },
  };

  /**
   * @param {string} widgetId
   * @returns {this}
   */
  forWidget(widgetId) {
    this.#widgetId = widgetId;
    this.#elementTestId = null;
    return this;
  }
```

```

/**
 * @param {string} componentName
 * @param {string} widgetId
 * @returns {this}
 */
forComponent(componentName, widgetId) {
  this.#componentName = componentName;
  this.#widgetId = widgetId;
  return this;
}

/**
 * @param {string} elementTestId
 * @returns {this}
 */
forElement(elementTestId) {
  this.#elementTestId = elementTestId;
  return this;
}

/**
 * @returns {this}
 */
only() {
  this.#onlyThis = true;
  return this;
}

/**
 * @param {string[]} clients
 * @returns {this}
 */
forClients(clients) {
  this.#clients = clients;
  return this;
}

/**
 * @param {('desktop' | 'mobile')[]} viewport
 * @returns {this}
 */
forViewports(viewport) {
  this.#viewport = viewport;
  return this;
}

/**
 * @param {('default' | 'dark')[]} colorSchemes
 * @returns {this}
 */
forColorSchemes(colorSchemes) {
  this.#colorSchemes = colorSchemes;
  return this;
}

/**
 * @param {Object|function(Object):Object} newPropsOrCallback
 * @returns {this}
 */
withWidgetProps(newPropsOrCallback) {
  if (typeof newPropsOrCallback === "function") {
    this.#widgetProps = newPropsOrCallback(this.#widgetProps);
  } else {
    this.#widgetProps = newPropsOrCallback;
  }
  return this;
}

/**
 * @param {string} affix
 * @param {Object|string} data
 * @param {string} contentType
```

```
   * @returns {this}
   */
  withRouteMock(affix, data, contentType) {
    this.#apiMocks = this.#apiMocks.filter((mock) => mock.endpoint !== affix);

    this.#apiMocks = [
      ...this.#apiMocks,
      {
        endpoint: affix,
        mockData: data,
        contentType: contentType,
      },
    ];
    return this;
  }

  /**
   * @param {Array<"canvas" | "timeout">} waitFor
   * @returns {this}
   */
  withWaitFor(waitFor) {
    this.#waitFor = waitFor;
    return this;
  }

  /**
   * @param {function(import('playwright').Page):Promise<void>} [pageInteraction]
   * @returns {this}
   */
  setPageInteraction(pageInteraction) {
    this.#pageInteraction = pageInteraction;
    return this;
  }

  /**
   * Sets the widget state for the test.
   * @param {('no-data' | 'loading' | 'no-response' | 'default')} widgetState
   * @returns {this}
   */
  setWidgetState(widgetState) {
    this.#widgetState = widgetState;
    return this;
  }

  /**
   * Sets title prefix for the test.
   * @param {string} title
   * @returns {this}
   */
  withTitle(title) {
    this.#title = title;
    return this;
  }

  /**
   * Runs the test with the given variant name.
   * @param {string} [variantName]
   * @returns {this}
   */
  test(variantName) {
    if (!this.#widgetId) throw new Error("Widget ID is not set");
    const playwrightTest = this.#onlyThis ? t.only : t;

    const testState = {
      widgetProps: this.#widgetProps,
      pageInteraction: this.#pageInteraction,
      widgetState: this.#widgetState,
      elementTestId: this.#elementTestId,
      waitFor: this.#waitFor,
    };

    for (const client of this.#clients) {
      for (const viewPort of this.#viewport) {
        for (const colorScheme of this.#colorSchemes) {
          playwrightTest(
            this.#getTestDescriptionFor(
              client,
              viewPort,
              colorScheme,
              testState.widgetState,
              variantName,
              testState.elementTestId,
              this.#title
            ),
            async ({ page }) => {
              await this.#mockCurrentDate(page);

              await this.#mockApiCall(page, testState.widgetState);

              await this.#setViewportFor(viewPort, page);

              await this.#addWidgetToPage(
                client,
                page,
                testState.widgetProps,
                colorScheme
              );
              await this.#loadPage(page, colorScheme, testState.waitFor);

              if (testState.pageInteraction) {
                await testState.pageInteraction(page);
              }

              expect(
                await this.#takeWidgetScreenshot(page, testState.elementTestId)
              ).toMatchSnapshot(
                this.#getReferenceFileFor([
                  client,
                  viewPort,
                  colorScheme !== "default" && colorScheme,
                  testState.widgetState !== "default" && testState.widgetState,
                  testState.elementTestId,
                  variantName,
                ])
              );
            }
          );
        }
      }
    }

    this.#resetState();
    return this;
  }

  /**
   * @private
   * @param {string} client
   * @param {string} viewPort
   * @param {string} colorScheme
   * @param {string} widgetState
   * @param {string} [variantName]
   * @param {string|null} elementTestId
   * @param {string|null} title
   * @returns {string}
   */
  #getTestDescriptionFor(
    client,
    viewPort,
    colorScheme,
    widgetState,
    variantName,
    elementTestId,
    title
  ) {
    return [
      ${title ? title : ""},
      this.#componentName || this.#widgetId,
       for client @${client},
       in @${viewPort} viewPort,
      colorScheme !== "default" && , @${colorScheme} color scheme,
      widgetState !== "default" && , @${widgetState} state,
      variantName && , @${variantName} variant,
      elementTestId && , @${elementTestId} element,
    ]
      .filter(Boolean)
      .join("");
  }

  /**
   * @private
   * @param {import('playwright').Page} page
   * @returns {Promise<void>}
   */
  async #mockCurrentDate(page) {
    const mockDate = new Date(Date.UTC(2023, 7, 4)).valueOf();
    await page.addInitScript({
              Date = class extends Date {
                constructor(...args) {
                  if (args.length === 0) {
                    super(${mockDate});
                  } else {
                    super(...args);
                  }
                }
              }
              const __DateNowOffset = ${mockDate} - Date.now();
              const __DateNow = Date.now;
              Date.now = () => __DateNow() + __DateNowOffset;
            });
  }

  /**
   * @private
   * @param {import('playwright').Page} page
   * @param {string} widgetState
   * @returns {Promise<void>}
   */
  async #mockApiCall(page, widgetState) {
    if (widgetState === "no-response") return;

    let mockApi;

    try {
      mockApi = await import(../../mock-api/${this.#widgetId}.mock);
    } catch (error) {
      return;
    }

    const { mockApiPresets } = mockApi;

    for (const {
      endpoint,
      data,
      contentType,
      customQuery = "",
      apiUrl,
    } of mockApiPresets.e2e[widgetState === "no-data" ? "noData" : "default"]) {
      await page.route(
        ${apiUrl || DEFAULT_API_URL}/${endpoint}${customQuery}*,
        async (route) => {
          if (widgetState === "loading") {
            return;
          } else if (contentType === "text/html") {
            await route.fulfill({
              contentType: "text/html",
              body: data,
            });
          } else {
            await route.fulfill({ json: data });
          }
        }
      );
    }
  }

  /**
   * @private
   * @param {string} viewPort
   * @param {import('playwright').Page} page
   * @returns {Promise<void>}
   */
  async #setViewportFor(viewPort, page) {
    await page.setViewportSize(this.#viewPortResolution[viewPort]);
  }

  /**
   * @private
   * @param {string} client
   * @param {import('playwright').Page} page
   * @param {Object|null} widgetProps
   * @param {string} colorScheme
   * @returns {Promise<void>}
   */
  async #addWidgetToPage(client, page, widgetProps, colorScheme) {
    await page.addInitScript({
      content:
            window.widget = ${JSON.stringify({
              "widget-id": this.#widgetId,
              "no-animations": "1",
              "dark-mode": colorScheme === "dark" ? "on" : "off",
              ...widgetProps,
            })};
            window.CONFIG_CLIENT_ID = ${JSON.stringify(client)};
          ,
    });
  }

  /**
   * @private
   * @param {import('playwright').Page} page
   * @param {string} colorScheme
   * @param {Array<"canvas" | "timeout">} waitFor
   * @returns {Promise<void>}
   */
  async #loadPage(page, colorScheme, waitFor) {
    await page.goto("./");
    if (colorScheme === "dark") {
      await page.addStyleTag({
        content:
                  body {
                    background-color: #1f2124 !important;
                  }
                ,
      });
    }
    await page.evaluate(() => document.fonts.ready);
    await page.waitForLoadState("load");
    if (waitFor.includes("canvas")) {
      await page.waitForSelector("canvas");
    }
    if (waitFor.includes("timeout")) {
      await new Promise((resolve) => setTimeout(resolve, 2000));
    }
  }

  /**
   * @private
   * @param {import('playwright').Page} page
   * @param {string|null} elementTestId
   * @returns {Promise<Buffer>}
   */
  async #takeWidgetScreenshot(page, elementTestId) {
    const element = await page
      .locator(
        this.#componentName
          ? [data-component=${this.#componentName}]
          : elementTestId
          ? [data-testid=${elementTestId}]
          : [widget-id=${this.#widgetId}]
      )
      .first();

    if (elementTestId) {
      const boundingBox = await element.boundingBox();
      const padding = 10;
      const screenshotOptions = {
        clip: {
          x: boundingBox.x - padding,
          y: boundingBox.y - padding,
          width: boundingBox.width + 2 * padding,
          height: boundingBox.height + 2 * padding,
        },
      };

      return await page.screenshot(screenshotOptions);
    } else {
      return await element.screenshot();
    }
  }

  /**
   * @private
   * @param {Array<string|boolean>} parts
   * @returns {Array<string>}
   */
  #getReferenceFileFor(parts) {
    return [
      this.#componentName || this.#widgetId,
      [this.#componentName || this.#widgetId, ...parts.filter(Boolean)].join(
        "-"
      ),
    ];
  }

  /**
   * Resets the state of the builder.
   * @private
   */
  #resetState() {
    this.#widgetState = "default";
  }
}

export default Builder;

// STB screenshot test builder v. 1.0
```