End-to-End tests

Grafana Labs uses a minimal homegrown solution built on top of Cypress for its end-to-end (E2E) tests.

Important notes:

- We generally store all element identifiers (CSS selectors) within the framework for reuse and maintainability.
- We generally do not use stubs or mocks as to fully simulate a real user.
- Cypress' promises do not behave as you'd expect.
- Testing core Grafana is slightly different than testing plugins.

Framework structure

Inspired by https://martinfowler.com/bliki/PageObject.html

- `Selector`: A unique identifier that is used from the E2E framework to retrieve an element from the Browser
- `Page`: An abstraction for an object that contains one or more `Selectors` with `visit` function to navigate to the page.
- `Component`: An abstraction for an object that contains one or more `Selectors` but without `visit` function
- `Flow`: An abstraction that contains a sequence of actions on one or more `Pages` that can be reused and shared between tests

Basic example

Let's start with a simple JSX example containing a single input field that we want to populate during our E2E test:

```
<input className="gf-form-input login-form-input" type="text" />
```

We *could* target the field with a CSS selector like `.gf-form-input.login-form-input` but that would be brittle as style changes occur frequently. Furthermore there is nothing that signals to future developers that this input is part of an E2E test. At Grafana, we use `aria-label` attributes as our preferred way of defining selectors instead of `data-*` as they also aid in accessibility:

```
<input aria-label="Username input field" className="gf-form-input login-form-input"
type="text" />
```

The next step is to create a `Page` representation in our E2E framework to glue the test with the real implementation using the `pageFactory` function. For that function we can supply a `url` and `selectors` like in the example below:

```
export const Login = {
  // Called via `Login.visit()`
  url: '/login',
  // Called via `Login.username()`
  username: 'Username input field',
};
```

The next step is to add the `Login` page to the `Pages` export within *<repo-root>/packages/grafana-e2e-selectors/src/selectors/pages.ts* so that it appears when we type `e2e.pages` in our IDE.

```
export const Pages = {
  Login,
  …,
  …,
  …,
};
```

Now that we have a `Page` called `Login` in our `Pages` const we can use that to add a selector in our html like shown below and now this really signals to future developers that it is part of an E2E test.

```
import { selectors } from '@grafana/e2e-selectors';

<input aria-label={selectors.pages.Login.username} className="gf-form-input login-
form-input" type="text" />;
```

The last step in our example is to use our `Login` page as part of a test.

- The `url` property is used whenever we call the `visit` function and is equivalent to the Cypress' `cy.visit()`.
- Any defined selector can be accessed from the `Login` page by invoking it. This is equivalent to the result of the Cypress function `cy.get(…)`.

```
describe('Login test', () => {
  it('passes', () => {
    e2e.pages.Login.visit();
    // To prevent flaky tests, always do a `.should` on any selector that you expect
to be in the DOM.
    // Read more here: https://docs.cypress.io/guides/core-concepts/retry-
ability.html#Commands-vs-assertions
    e2e.pages.Login.username().should('be.visible').type('admin');
  });
});
```

Advanced example

Let's take a look at an example that uses the same `selector` for multiple items in a list for instance. In this example app we have a list of data sources that we want to click on during an E2E test.

```
<ul>
  {dataSources.map(({ id, name }) => (
    <li className="card-item-wrapper" key={id}>
      <a className="card-item" href={`datasources/edit/${id}`}>
        <div className="card-item-name">{name}</div>
      </a>
    </li>
  ))}
</ul>
```

Just as before in the basic example we'll start by creating a page abstraction using the `pageFactory` function:

```
export const DataSources = {
  url: '/datasources',
  dataSources: (dataSourceName: string) => `Data source list item
${dataSourceName}`,
};
```

You might have noticed that instead of a simple `string` as the `selector`, we're using a `function` that takes a string parameter as an argument and returns a formatted string using the argument.

Just as before we need to add the `DataSources` page to the exported const `Pages` in `packages/grafana-e2e-selectors/src/selectors/pages.ts`.

The next step is to use the `dataSources` selector function as in our example below:

```
<ul>
  {dataSources.map(({ id, name }) => (
    <li className="card-item-wrapper" key={id}>
      <a className="card-item" href={`datasources/edit/${id}`}>
        <div className="card-item-name" aria-label=
{selectors.pages.DataSources.dataSources(name)}>
          {name}
        </div>
      </a>
    </li>
  ))}
</ul>
```

When this list is rendered with the data sources with names `A`, `B` and `C`,the resulting HTML would look like:

```
<div class="card-item-name" aria-label="Data source list item A">A</div>
<div class="card-item-name" aria-label="Data source list item B">B</div>
<div class="card-item-name" aria-label="Data source list item C">C</div>
```

Now we can write our test. The one thing that differs from the basic example above is that we pass in which data source we want to click on as an argument to the selector function:

```
describe('List test', () => {
  it('clicks on data source named B', () => {
    e2e.pages.DataSources.visit();
    // To prevent flaky tests, always do a .should on any selector that you expect
to be in the DOM.
    // Read more here: https://docs.cypress.io/guides/core-concepts/retry-
ability.html#Commands-vs-assertions
    e2e.pages.DataSources.dataSources('B').should('be.visible').click();
  });
});
```

Aria-Labels vs data-testid

Our selectors are set up to work with both aria-labels and data-testid attributes. Aria-labels help assistive technologies such as screenreaders identify interactive elements of a page for our users.

A good example of a time to use an aria-label might be if you have a button with an X to close:

```
<button aria-label="close">X<button>
```

It might be clear visually that the X closes the modal, but audibly it would not be clear for example.

```
<button aria-label="close">Close<button>
```

The example might read aloud to a user as "Close, Close" or something similar.

However adding aria-labels to elements that are already clearly labeled or not interactive can be confusing and redundant for users with assistive technologies.

In such cases rather than adding unnecessary aria-labels to components so as to make them selectable for testing, it is preferable to use a data attribute that would not be read aloud with an assistive technology for example:

```
<button data-testid="modal-close-button">Close<button>
```

We have added support for this in our selectors, to use:

Prefix your selector string with "data-testid":

```
export const Components = {
  Login: {
    openButton: 'open-button', // this would look for an aria-label
    closeButton: 'data-testid modal-close-button', // this would look for a data-testid
  },
};
```

and in your component, import the selectors and add the data test id:

```
<button data-testid={Selectors.Components.Login.closeButton}>
```