Are UI based E2E Tests the Best Way to Reduce Risk?

Stephen Kilbourn
PNSQC 2021
About Me

- Quality Engineer for 10+ years
- Scrum Master
- Loves automation in selenium, appium, Cypress, and more
- My passion is to help my teams accelerate the achievement of shippable quality
“Write tests to verify everything.”

This seems to be the expectation of some in the tech industry, but

1. Are UI based end to end (E2E) tests the best tool to verify things?
2. Is there a way to reduce risks before testing?
I propose: UI based E2E Tests in Web & Mobile Apps

- Provide some value
- Are best used in as part of a holistic risk reduction strategy
- Should be a supplement to team collaboration
- Should be used sparingly
To get to that conclusion, I will:

- Give an Intro to UI Based E2E Tests
- Step back and remind ourselves why we test
- Introduce a Holistic Risk Reduction strategy
Intro to UI Based E2E Tests

10,000 foot view of how these tools operate
Robotic Users?
Yes and No

- Code is written to locate elements on a page
- If expected element exists, then code can interact with it
- These Interactions are pieced together into a “flow” of a simulated user journey
How do you find elements?
Consider this button

- **XPath?**
  ```html
  /html/body/div/form/button
  ```

- **CSS Selector?**
  ```html
  body > div > form > button
  ```

- **Role?**
  ```javascript
  findByRole('button' { name: 'Submit' });
  ```
Why this is imprecise:

- Slow loading elements may cause failures
- Addition of new elements may break your method for locating the element
- Unintended layout changes won’t necessarily be caught
- This isn’t how a real user interacts with a page
Additional Challenges

- Slow to run
- Complex test environments, especially as microservices increase
- Reliant on third party dependencies
- Delayed developer feedback until pieces are integrated
Things they can’t catch

- Visual defects
- Unexplored edge cases
- Missed requirements
Why Do We Test?

Because E2E has challenges, let’s revisit what we want out of our tests.
Why Do We Test?

- Provide Confidence in what we deliver to the end user
- Provide fast, accurate, reliable, and predictable feedback
- Make maintenance easier

In other words, we want tests to allow us to accelerate the achievement of a shippable quality product.
Holistic Risk Reduction

Looking beyond E2E tests
Holistic Risk Reduction

Leverage static testing
Test throughout the application stack
Test contracts between consumers & clients
Minimize feedback loops with frequent conversations
Sparingly use E2E tests
Step 1: Leverage Static Testing

Catch issues as you type
Static Tests

- Automatically applied style guide (eg. prettier.js)
- linting
  - catches syntactic or semantic errors before code is run
- type checking
  - verifies a variable is used correctly throughout the code

Confused? Let's run Static Tests

```javascript
const a = b || c && d
const aa = b && c || d

const TenHello = () => {
  const Hello = "hello"
  for (let i=0; i > 10; i++) {console.log(Hello)}
}
```
Easier to read and we see our error

Originally

```javascript
const a = b || c && d
const aa = b && c || d

const TenHello = () => {
    const Hello = "hello"
    for (let i=0; i > 10; i++) {console.log(Hello)}
}
```

After Static Checks

```javascript
const a = b || (c && d);
const aa = (b && c) || d;

const TenHello = () => {
    const Hello = "hello"
    for (let i = 0; i < 10; i++) {
        console.log(Hello);
    }
};
```
Step 2: Test Throughout the Stack

Catch bugs as early as possible
Traditional Unit Tests

- Generally we think of just testing small functions (i.e., units)

```javascript
const a = b || c && d
const aa = b && c || d

const TenHello = () => {
    const Hello = "hello"
    for (let i=0; i < 10; i++)
        console.log(Hello)
}
```
@testing-library “Unit Tests”

“The more your tests resemble the way your software is used, the more confidence they can give you.”

- Allow testing of front end very similar to how a real user (or E2E tests) use the application
Render the Form and Test Behaviour

```javascript
<template>
  <div>
    <label for="email">Email address</label>
    <input type="email" id="email" aria-describedby="email-help" placeholder="Your email"
   />
  </div>
  <div>
    <label for="password">Password</label>
    <input type="password" id="password" placeholder="password"
   />
  </div>
  <button type="submit">Submit</button>
</template>

test('testing login form', async () => {
  const handleSubmit = jest.fn()
  render(<LoginForm onSubmit={(handleSubmit} />
  userEvent.type(screen.getByLabelText('/email address/i'), 'johndoe@email.com')
  userEvent.type(screen.getByLabelText('/password/i'), 'secret')
  userEvent.click(screen.getByRole('button', { name: /submit/i })))
  await waitFor(() =>
    expect(handleSubmit).toHaveBeen CalledWith({
      email: 'johndoe@email.com',
      firstName: 'password',
    }, expect.anything())
  )
})
```
Holistic Risk Reduction

Leverage static testing ✅

Unit & Static Tests throughout the application stack ✅

But what happens when we start integrating?
Step 3: Test Consumer & Provider Contracts

Make sure we integrate together
Contract Testing

- E2E tests help verify that all components of a system can communicate
- Contract Tests provide a way to verify this earlier without the E2E challenges
Consumer and Providers

- Consumers make requests
- Providers provide a response

Consumer driven contract tests help us verify that the provider knows what request a consumer will make and will provide a response in the format a consumer expects.
Consumer Driven Contract Test

- Consumers makes a contract of how they expect a provider to behave
- Consumer publishes the contract
- Provider checks that it meets the Consumer’s expectations
- The provider publishes its results
Pact Contract Example

- Register the interactions that the consumer is expecting to receive from the provider

```javascript
describe('Songs Service', () => {
  describe('When a request to list all songs is made', () => {
    beforeAll(() => {
      provider.setup().then(() => {
        provider.addInteraction(
          uponReceiving: 'a request to list all songs',
          withRequest: {
            method: 'GET',
            path: '/songs',
          },
          willRespondWith: {
            status: 200,
            body: eachLike({
              id: 1,
              name: like('I want to Hold Your Hand'),
              year: like(1964),
            },
            { min: 5 } ),
          },
        );
      });
    });
    test('should return the correct data', async () => {
      const response = await fetchSongs(URL, PORT);
      expect(response[0].name).toBe('I want to Hold Your Hand');
      expect(response[0].year).toBe(1964);
    });
    afterEach(() => provider.verify());
    afterAll(() => provider.finalize());
  });
});
```

#PNSQC2021
Pact Contract Example

- Consumer uploads the contract to a broker
Pact Contract Example

- The provider then checks out the contracts from all its consumers from the broker and verifies that it meets those expectations
Holistic Risk Reduction
Leverage static testing ✅
Unit & Static Tests throughout the application stack ✅
Contract testing at integration points ✅
But what happens when if misunderstood requirements?
Step 4: Frequent Conversations

Tools are great, but won’t prevent missed requirements
Traditionally Conversations are Limited

- Requirements gathering & backlog grooming devolves into generating a “to-do” list of Acceptance Criteria
- Maybe a team does BDD formulation
- Little conversation happens after the developer begins work
- Missed or misunderstood requirements turn into bugs or rejected stories
The Importance of Conversations

- Consider Edge cases early
- Discover missed requirements
- Share context between the team members
Increase Conversation Points

- Requirement gathering
- Story kickoff
- Questions during development
- Code review and demo work as a team
- During feature integration
Step 5: Sparingly Use E2E Tests

Reserve these for specific sanity checks
Key E2E Tests

- Test flows to verify connectivity between components
- Ensure proper environment variables were in place
- Validate that end users can accomplish key tasks

**DO NOT** try to validate every Acceptance criteria that can be validated elsewhere
Conclusion

Go beyond E2E Tests for Holistic Risk Reduction
Are E2E Tests the Best Way to Reduce Risk?