Cypress überall – Ein einziges Automatisierungswerkzeug für alle Teststufen?!

Dehla Sokenou
WPS – Workplace Solutions
dehla.sokenou[at]wps.de

1 Einführung

Testautomatisierung ist heutzutage der Standard in der Softwareentwicklung, besonders im agilen Umfeld. Bei sich – gewollt – ständig ändernden und neuen Anforderungen ist ein stabiles Gerüst aus Tests die Versicherung gegen ungewollte Seiteneffekte bei der Weiterentwicklung des Softwaresystems. Aktuelle Softwaresysteme bestehen heute oft aus einem Backendteil (z.B. in Java oder C# geschrieben) und einem Webfrontendteil. Letzterer wird in der Regel mit Hilfe eines modernen Single-Page-Framework entwickelt.

Betrachtet man dieses System aus der Sicht eines Testers, so gibt es einige Unterschiede zwischen dem automatisierten Test des Backends und des Frontends. Im Backend bereitet Testautomatisierung meist wenig Probleme. Nachdem es in den vergangenen Jahren oft ganz unterschiedliche Werkzeuge gab, um Tests zu automatisieren, hat sich dort inzwischen weitgehend die xUnit-Familie durchgesetzt, und das oft auf allen Teststufen von den Unit-Tests über Integrationstests bis hin zur Schnittstelle. Frameworks wie Spring unterstützen dies mit Erweiterungen für xUnit, um bspw. Integrations- und API-Tests zu ermöglichen. Diverse Mocking-Frameworks erlauben das isolierte Testen einzelner Teile des Systems.

Im Frontend ist die Situation etwas komplizierter. Hier fehlt es oft allein am Verständnis, was eigentlich eine Unit ist, die zu testen ist. Es vermischen sich Business- und Anzeigelogik – je nach verwendetem Web-Framework mehr oder weniger. Frontend-Tests können durch die automatisierte Nutzerinteraktion langsam und unzuverlässig in der Ausführung sein. Einige Testwerkzeuge können allein durch die Art der Steuerung des Frontends diese Probleme nicht verhindern und erfordern Workarounds, um Tests zu stabilisieren, bspw. durch explizites Warten auf das Erscheinen eines Elements auf dem UI. Zudem gab es bisher kein Werkzeug, das alle Teststufen unterstützt, sondern in der Regel Werkzeuge für den End2End-Test und Werkzeuge für die niedrigen Teststufen – erst mit Cypress [1] und seiner relativ jungen Erweiterung *Component Testing* gibt es einen Vertreter, der für alle Teststufen der Testpyramide im Webfrontend eingesetzt werden kann.

2 Cypress – Eine kurze Historie

Das Testwerkzeug Cypress ist vor knapp 10 Jahren als Gegenpol zu Selenium [2] gestartet. Zunächst wurden nur End2End-Tests unterstützt. Während Selenium den Browser von außen über die WebDriver-API steuert, greift Cypress von innen auf die zu testende Anwendung zu, indem es mit dieser im gleichen Browser-Tab läuft. Die ermöglicht eine deutlich schnellere Testausführung sowie das Mocken von Server-Requests und Responses auch bei End2End-Tests. Gleichzeitig ergeben sich dadurch Einschränkungen, so werden Aktionen nicht direkt vom Browser ausgeführt, sondern durch Javascript-Befehle simuliert. Gerade im End2End-Test kann dies in seltenen Fällen zu schwer bis gar nicht testbaren Aktionen führen. Zudem kann Cypress durch seine technologische Basis keine Multi-Tab-Anwendungen testen. Trotz dieser Nachteile hat sich Cypress inzwischen weitgehend gegen Selenium durchgesetzt.

Im Jahr 2021 kam dann mit Cypress Component Testing eine Erweiterung, mit der auch die unteren Teststufen mit dem gleichen Werkzeug abgedeckt werden konnten. Da hierbei im Gegensatz zu End2End-Tests Komponenten spezifisch gekapselt werden müssen, ist eine Unterstützung für das jeweilige Web-Framework notwendig. Während zunächst nur Vue und React unterstützt wurden, wuchs der Support für andere Web-Frameworks aber schnell und umfasst heute eigentlich alle verbreiteten Web-Frameworks, u.a. auch Angular.

Für den Webentwickler ergeben sich dadurch einige Vorteile. Er ist nicht mehr auf den Einsatz unterschiedlicher Testwerkzeuge angewiesen, sondern kann sich auf ein Werkzeug – Cypress – beschränken. War bisher beim Wechsel des Web-Frameworks oder bei einem neuen Projekt mit einer anderen Frontend-Technologie meist der Einsatz eines neuen Testframeworks für die unteren Teststufen notwendig, ist dies nicht mehr notwendig. Ähnliches galt, wenn man zuvor bereits ein universell verwendbares Werkzeug wie Jest [3] verwendet hat, das ebenfalls technologieunabhängig eingesetzt werden kann. Jest hat aber gegenüber Cypress andere Nachteile, wie die durch die verwendete Browser-Emulation sehr langsame Testausführung und das fehlende visuelle Feedback.

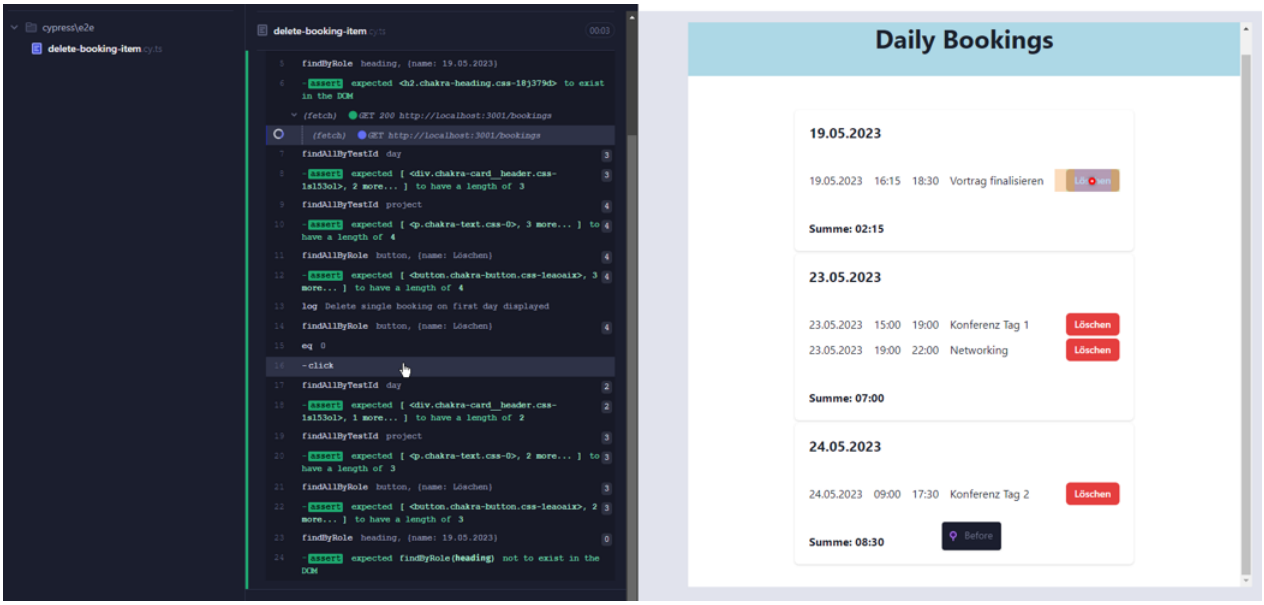


Abbildung 1: Testausführung im interaktiven Cypress-Runner

Einige besondere Eigenschaften von Cypress erleichtern das Schreiben von Tests und die Analyse von Fehlern beim Testen deutlich, was aus unserer Sicht eines seiner Erfolgsgeheimnisse ist. Neben Mocking ist die Clock ein wichtiges Feature, dass zur Simulation von Zeitverhalten dient, bspw. um Tests zu bestimmten Zeitpunkten auszuführen oder eine gewisse Zeit zwischen zwei Aktionen vergehen zu lassen. Eingabe erfolgen in Benutzergeschwindigkeit, das heißt, es wird wirklich getippt, so dass Javascript-Funktionen an Feldern getriggert werden können. Das Warten auf das Erscheinen und Verschwinden von UI-Elementen erfolgt implizit und ist Teil der Assertions, so dass kein explizites Warten mehr notwendig ist. Der interaktive Testreport besteht aus DOM-Snapshots, in denen man vor und zurückspringen kann und die den Zustand vor und nach einer Aktion anzeigen, so dass man den Testablauf wie eine Art Zeitreise noch einmal abspielen kann. In der CI-Pipeline lassen sich automatische DOM-Snapshots im Fehlerfall aktivieren, so dass auch dort die Analyse bei fehlgeschlagenen Tests erleichtert wird. Der Tester erhält dadurch ein visuelles Feedback, dass wir für Frontend-Tests für unabdingbar halten. Durch Cypress Component-Tests lässt sich das Frontend testgetrieben entwickeln, da die Testausführung parallel zur Entwicklung laufen kann und bei Änderungen an Tests oder Produktivcode die Tests automatisch neu ausführt – das visuelle Feedback ist dabei eine große Hilfe.

3 Cypress in Action

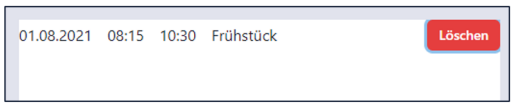
In der Entwicklung arbeitet man mit Cypress in der Regel mit dem interaktiven Cypress-Runner. Bei Änderungen wird der aktuell ausgewählte Test automatisch ausgeführt, was ein schnelles Arbeiten ermöglicht. Abbildung 1 zeigt ein Beispielsystem zur Buchung von Terminen nach der Testausführung im Cypress-Runner. Gezeigt ist ein End2End-Test, der das Löschen eines Buchungseintrags testet. End2End-Tests rufen die Oberfläche eines laufenden Systems im Browser auf, wobei das Backend entweder real angesprochen oder aber gemockt werden kann.

```
describe('Delete Booking Items', () => {
  it('should delete single booking on a day and remove card', () => {
    cy.viewport(800, 800);
    cy.visit('/');

    cy.findByText('Daily Bookings').should('exist');
    cy.findByRole('heading', { name: '19.05.2023' }).should('exist');
    cy.findAllByTestId('day').should('have.length', 3);
    cy.findAllByTestId('project').should('have.length', 4);

    cy.log('Delete single booking on first day displayed');
    cy.findAllByRole('button', { name: 'Löschen' }).eq(0).click();

    cy.findAllByTestId('day').should('have.length', 2);
    cy.findAllByTestId('project').should('have.length', 3);
    cy.findByRole('heading', { name: '19.05.2023' }).should('not.exist');
  }); });
```

Abbildung 2: End2End-Test Löschen einer Buchung

Das dazugehörende Listing zeigt Abbildung 2. Cypress-Tests werden – wie im Listing gezeigt – in Javascript implementiert, einer Sprache also, die Webfrontend-Entwickler nutzen. Da End2End-Tests zumindest für das Frontend ein laufendes System erfordern, sind sie nicht nur langsam in Bezug auf die Testausführung, sondern der Zyklus Test – Entwicklung ist auch verlangsamt, da Änderungen am Produktivcode einen kompletten Buildzyklus erfordern.

Besser wäre es, wenn nicht das gesamte UI im Browser hochgefahren werden müsste, sondern nur die




```
describe('BookingItemViewer', () => {

 it('should be possible to delete an item', () => {
  // arrange
  const bookingItem: BookingItem = {
   id: 1,
   start: 1627798500000,
   end: 1627806600000,
   project: 'Frühstück',
  };
  const handleDelete = cy.stub().as('onDelete');
  cy.mount(
    <BookingItemViewer bookingItem={bookingItem}
              onDelete={handleDelete} />
  );

  // act
  cy.findByRole('button', { name: 'Löschen' }).click();

  // assert
  cy.get('@onDelete').should('be.calledOnceWith', 1);
 }); });
```

Abbildung 3: Component-Test Anzeige einer Buchung

```
describe('Date Util', () => {
 const emptyBookings: BookingItem[] = [];
 const singleBookings: BookingItem[] = [...];
 const multipleBookings: BookingItem[] = [...];

 [
  { bookingItems: emptyBookings, result: '00:00' },
  { bookingItems: singleBookings, result: '01:00' },
  { bookingItems: multipleBookings, result: '03:30' },
 ].forEach(({ bookingItems, result }) => {
  it('should calculate correct daily sum for bookings', () => {
   // act
   const formattedSum = formatSum(bookingItems);

   // assert
   expect(formattedSum).to.eq(result);
}); }); });
```

Abbildung 4: Unit-Test Berechnung Gesamtzeit der Tagesbuchungen

Komponente, an der gerade gearbeitet wird. Dies hat nicht nur Vorteile in Bezug auf die Geschwindigkeit, sondern es können auch gezielt einzelne Funktionen getestet werden, ohne zunächst im Test den Workflow bis zur zu testenden Komponente durchlaufen zu müssen. Daten können gezielt bereitgestellt werden, auch die Provokation von Ausnahmen ist einfach möglich, um das Fehlerverhalten zu testen.

Cypress Component-Tests bieten diese Möglichkeit. Das Listing in Abbildung 3 zeigt einen typischen Component-Test, hier für die Anzeige-Komponente für einen einzelnen Buchungseintrag. Der Test prüft, ob beim Klick auf den Löschen-Button das erwartete Event geworfen wird, um den Buchungseintrag an anderer Stelle – nicht innerhalb der Komponente – zu löschen. Die in Cypress angezeigte UI-Komponente zeigt die Abbildung oberhalb des Listings.

Prinzipiell unterscheiden sich die zwei Funktionen E2E und Component Testing von Cypress nur in der Art, wie der Test auf das Testobjekt zugreift. Im Fall von E2E wird eine URL besucht (*visit*), im Fall von Component Testing eine Komponente gemountet (*mount*) – vgl. die Pfeile in den zwei Abbildungen. Grundsätzlich ist davon auszugehen, dass sich die Tests auf den unterschiedlichen Teststufen auch in anderer Hinsicht inhaltlich unterscheiden, da sie verschiedenen Zwecken dienen. Syntaktisch nutzen sie jedoch die gleiche Basis, so dass das Lernen neuer Sprachelemente nicht notwendig ist, wenn man bereits mit Cypress E2E vertraut ist.

Bei der Betrachtung der unterschiedlichen Teststufen fehlt jedoch noch eine Ebene. Auch wenn im Frontend meist der Anteil der Anzeigekomponenten überwiegt, so gibt es doch auch hier reine Geschäftslogik, die ohne UI auskommt. In unserem Beispiel nutzen wir eine Funktion, die u.a. die Gesamtzeit aller Buchungen eines Tages berechnet (siehe Abbildung 4). Tests für solche rein logischen Einheiten im Frontend sind mit Cypress auch möglich, dann wird aber keine Komponente gemountet und somit kann der Cypress-Runner zwar den Testlauf anzeigen, aber keine UI. Das visuelle Feedback ist hier naturgemäß nicht gegeben, weshalb einer der großen Pluspunkte von Cypress nicht zum Tragen kommt. Cypress bewegt als im Bereich der reinen Unit-Tests ohne UI-Anteil auf der gleichen Ebene wie andere Testwerkzeuge.

4 Einschränkungen

Wir hatten bei der Vorstellung von Cypress bereits einige Einschränkungen genannt. Neben der fehlenden Möglichkeit, Multi-Tab-Anwendungen zu testen, ist auch ein domänenübergreifender Test nur mit Hilfe eines Workarounds möglich, da die Anwendung in einem iFrame innerhalb von Cypress läuft und somit alle Einschränkungen für iFrames auch für die zu testende Anwendung gelten. Ein allumfassender Ansatz ist mit Cypress nicht möglich. Möchte man neben dem Webfrontend das gleiche Werkzeug auch für andere Oberflächen, bspw. Rich-Clients, anwenden oder gar ein Backend, das nicht auf Javascript basiert, dann ist Cypress sicher die falsche Wahl. In diesem Fall wird man in der Regel auf kommerzielle Werkzeuge ausweichen müssen, die oft einen ganzheitlicheren Ansatz als Open-Source-Werkzeuge bieten.

Da Cypress das Testobjekt nicht von außen steuert, sondern von innen per Javascript, sind einige Aktionen nicht oder nur durch aufwendige Zusatzimplementierungen möglich. So hatten wir bspw. das Problem, dass Button-Klicks bei einer Anwendung, die auf Remix basiert, nicht korrekt ausgeführt wurden.

Je nach Zielgruppe ist abzuwägen, ob die Implementierung von Tests für die Tester ein gangbarer Weg ist. Record&Play-Werkzeuge wie Selenium IDE, Behavior-Driven Development oder Keyword-Driven-Tests sind für Tester ohne Programmiererfahrung die bessere Alternative.

```
it('should be possible to delete an item', async () => {

 fetch.mockResponse(initialBookings);

 const onDelete = jest.fn();

 render(
   <MyComponent onDelete={onDelete} />);

 expect(screen.getByTestId('date')).toHaveTextContent('01.08.2021‘);

 const sums = screen.getAllByTestId('sum‘);
 expect(sums[0]).toHaveTextContent('01:00');

 fireEvent.click(await screen.findByRole('button', { name: 'Löschen' }));

 expect(onDelete).toHaveBeenCalledWith(1);

 expect(formattedSum).to.eq(result);
});
```

```
it('should be possible to delete an item', () => {

 cy.intercept(url, initialBookings);

 const onDelete = cy.stub().as('onDelete')

 cy.mount(
   <MyComponent onDelete={onDelete} />);

 cy.findByTestId('date').should('have.text', '01.08.2021‘);

 cy.findAllByTestId('sum').as('sum‘);
 cy.get('@sum').eq(0).should('contain.text', '01:00');

 cy.findByRole('button', { name: 'Löschen' }).click();

 cy.get('@onDelete').should('be.calledWith', 'id');

 expect(formattedDate).toEqual(result);
});
```

Abbildung 5: Migration von Tests - links Jest, rechts Cypress

Es gibt zwar für IDE Support, der aber im Gegensatz zu anderen Testwerkzeugen aktuell noch eher rudimentär ist. So fehlt u.a. in einigen Umgebungen der Play-Button, der einen Tests direkt aus der IDE ausführt. Das Starten der interaktiven Umgebung ist zudem etwas langsamer als das direkte Abspielen von Tests aus der IDE heraus. Ist der Cypress-Runner aber erst einmal gestartet, werden die Tests sehr schnell ausgeführt. Beide Nachteile umgehen wir, indem wir die interaktive Testausführungsumgebung neben unsere IDE legen und diese die Tests während der Entwicklung stetig automatisch bei Änderungen ausführt. Diese andere Arbeitsweise unterstützt die testgetriebene Entwicklung von UIs sehr gut.

Bevor man also seine bisherigen Testsuiten auf Cypress migriert, sollte man sich fragen, ob diese Einschränkungen für das eigene Projekt ein Show-Stopper sind. Wenn allerdings Cypress zum Einsatz kommen soll, dann ist die Migration bestehender Testsuiten – je nach Anzahl der bereits bestehenden Tests – zwar zeitlich aufwendig, aber nach unserer Erfahrung relativ einfach möglich. 80 - 90% der Arbeit war in unserem Fall reine Übersetzungsarbeit von einer Syntax (hier: Jest) in die andere (hier: Cypress), siehe auch Abbildung 5. Dabei hat es geholfen, dass wir auch für die Jest-Tests die etablierte Testing Library [4] verwendet haben, die auch im Falle von Cypress unterstützt wird – analoges gilt für andere etablierte Libraries. Für die Migrationsarbeit haben wir das Pfadfinderprinzip angewendet, d.h. wenn Tests angepasst werden mussten oder zuvor flaky waren, wurden sie während der laufenden Entwicklung in einen Cypress Component-Test umgewandelt. Gleichzeitig diente die Migrationsarbeit dazu, fehlende Tests zu ergänzen und überflüssige Tests zu entfernen. Unsere Testsuiten laufen mit Cypress sehr viel stabiler und schneller als zuvor.

5 Zusammenfassung und Ausblick

Cypress bietet mit der Erweiterung Component Testing ein ganzheitliches Paket für die Frontend-Entwicklung an. War Cypress bereits sehr beliebt, als nur End2End-Tests unterstützt wurden, sorgt die Unterstützung weiterer Teststufen für eine noch größere Akzeptanz des Werkzeugs. Alles in allem kann man sagen, dass ein Projekt ausschließlich auf Basis von Cypress für das Testen im Webfrontend möglich ist und eine Reihe von Vorteilen bietet.

Obwohl aktuell Cypress aufgrund seiner Fähigkeiten eines der beliebtesten Testautomatisierungswerkzeuge im Frontend ist, so dreht die Welt sich gerade im Bereich der Webentwicklung sehr schnell weiter. Neue Frameworks laufen etablierten Playern den Rang ab und auch im Testbereich erscheinen immer wieder neue Werkzeuge auf dem Markt, wie aktuell Playwright [5] für End2End-Tests. Welche sich am Ende davon auch in Zukunft durchsetzen werden, bleibt abzuwarten.

Referenzen

[1] Cypress.
https://www.cypress.io/.

[2] Selenium.
https://www.selenium.dev/.

[3] Jest.
https://jestjs.io/.

[4] Testing Library.
https://testing-library.com/.

[5] Playwright.
https://playwright.dev/.