Wie UI-Component Tests unsere E2E-Tests schneller und robuster machen

In der schnelllebigen Softwareentwicklung ist es wichtiger denn je, dass Anwendungen stabil und zuverlässig funktionieren und die Kundenansprache und das Design in den unterschiedlichen Komponenten der Anwendung möglichst konsistent sind. Aus diesem Grund ist das Testen in verschieden Testebenen, um dem Qualitäts- und Kostendruck gerecht zu werden, ein wichtiger Aspekt. Dieser Blog-Eintrag beschäftigt sich mit UI-Component Tests und einem mächtigen Werkzeug namens Cypress, welches Teams ermöglicht, ihre UI-Komponenten schnell und effizient zu testen. Cypress bietet eine intuitiv verwendbare Oberfläche, die es Entwickler*innen ermöglicht, ihre Tests auf einfache Weise zu schreiben und auszuführen.

Die Situation in unserem Team

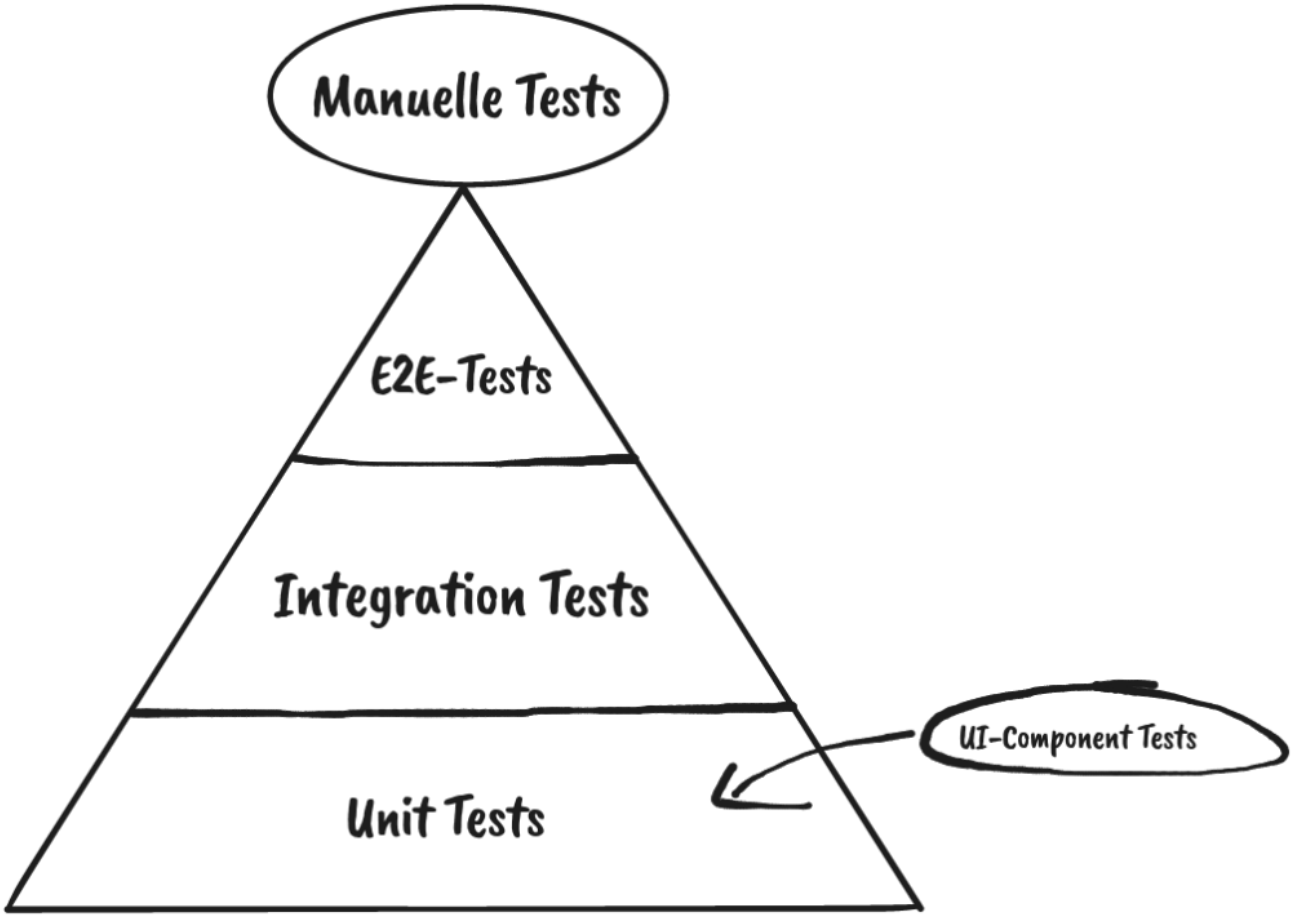

Abb. 1: Eine Abbildung unserer Testpyramide (Quelle: Eigene Darstellung)

Das agile Entwicklungsteam „Kundenbindung und Abo“ (KubA) steht vor der Herausforderung, dass unsere Services immer umfangreichere Front-End Komponenten ausliefern. Wichtig ist dabei, diese nicht nur auf ihre Funktionalität zu testen, sondern auch unseren unterschiedlichen Mandanten (thalia.de, orellfuessli.ch, osiander.de, etc.) regelmäßig auf ihre Konsistenz, beispielsweise in Bezug auf Styling, Wordings und Währungen zu überprüfen. Um dieser Herausforderung zu begegnen, setzten wir zu Beginn nur auf E2E-Tests mit Cypress, die sowohl das Testen der Funktionalität als auch die Überprüfung der Darstellung übernahmen (s. Abb. 1). Mit der Zeit übernahm das Team immer mehr Themen wie Payback, KultClub, Bonus- und Premiumkarte, wodurch immer mehr Frontend ausgeliefert werden musste. Deswegen führten wir zunächst E2E-Tests mit Cypress ein, um das Frontend in diesen Services zu testen, aber diese wurden im Laufe der Zeit immer umfangreicher.

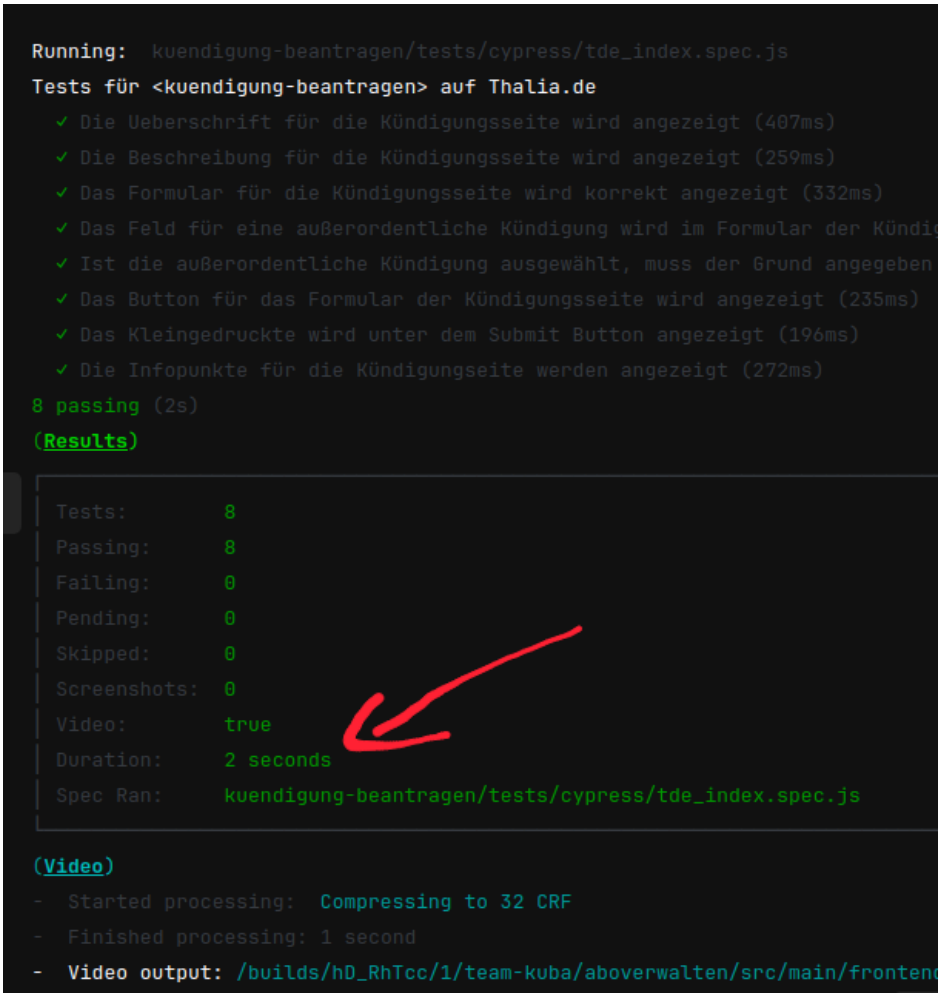

Abb. 2: Ausschnitt der Testausführung in einem Feature-Branch in Gitlab (Quelle: Eigene Darstellung)

Dies hatte eine erschwerte Wartbarkeit und eine längere Testlaufzeit zur Folge. Außerdem mussten die E2E-Tests auch bei kleinsten Änderungen angepasst werden, weil vergessen wurde, in Teststep „x“ String „y“ zu ändern, was wiederum zu Verzögerungen bei der Entwicklung geführt hat. Deswegen suchten wir nach Lösungen und wurden beim Team „Suchen & Beraten“ fündig. Das Team hatte in Rahmen eines PoCs eine Kombination aus Cypress und Express.js eingesetzt, sodass es möglich war Komponenten und ihre Darstellung für den jeweiligen Mandaten automatisiert innerhalb weniger Sekunden (s. Abb. 2) bereits im Feature-Branch zu testen. Deswegen entschlossen wir uns, dieses Vorgehen für unser Team zu adaptieren, um wieder auf unseren Fail-Fast Ansatz einzuzahlen. In der Testpyramide befinden sich diese Tests auf der Unit-Test Ebene (s. Abb. 1).

Was sind UI-Component Tests?

Doch bevor wir uns sofort in Details verlieren, werden im Folgenden Grundlagen zu den Themen UI-Component Tests und Cypress erläutert.

UI-Component Tests sind ein wichtiger Teil einer unserer Teststrategie, da sie dazu beitragen, die Integrität und Zuverlässigkeit der Benutzeroberfläche (UI) einer Anwendung sicherzustellen. Sie überprüfen, ob die UI wie erwartet funktioniert und ob sie den Anforderungen der Benutzer*innen entsprechen. Außerdem können sie dabei helfen, GUI-/E2E-Tests zu entschlacken, um diese schneller und robuster zu machen.

Warum Cypress?

Abb. 3: Logo Cypress (Quelle: cypress.io)

Cypress ist ein modernes (End-to-End-) Testing Framework, das speziell für die Entwicklung von Web-Anwendungen entwickelt wurde. Es ermöglicht Entwickler*innen, ihre Anwendungen automatisch zu testen, indem es direkt im Browser ausgeführt wird.

Ein wichtiger Vorteil von Cypress ist, dass nicht nur die Funktionalität der Anwendung, sondern auch die Benutzerinteraktion getestet werden kann. Cypress arbeitet direkt mit dem DOM (Document Object Model) und simuliert das Verhalten von Benutzer*innen, was es zu einem idealen Werkzeug für das Testing von UI-Komponenten macht.

Cypress bietet eine einfache und intuitiv verwendbare API, die es Entwickler*innen ermöglicht, ihre Tests schnell und einfach zu schreiben und auszuführen. Es unterstützt automatisch paralleles Testing und bietet umfangreiche Fehlerdiagnose- und Debugging-Funktionen. Infolgedessen können Probleme in ihren Tests identifiziert und effektiv behoben werden.

Cypress ist auch umfangreich dokumentiert und wird von einer aktiven und hilfsbereiten Community unterstützt, die ständig neue Funktionen und Erweiterungen entwickelt. Außerdem kommt Cypress nicht nur bei uns, sondern auch bei vielen anderen Entwicklungsteams im Zusammenhang mit E2E-Tests zum Einsatz, wodurch wir uns die Einführung eines weiteren Tools sparen können.

Das Wichtigste zusammenfassend:

- Echtzeit-Debugging: Cypress bietet eine integrierte Debugging-Umgebung, die es ermöglicht, Tests in Echtzeit auszuführen und zu debuggen. Dies macht es einfacher, Fehler in Tests zu identifizieren und zu beheben.
- Unterstützung für verschiedene Browser: Cypress kann in verschiedenen Browsern wie Chrome, Firefox und Edge ausgeführt werden. Dies ermöglicht es sicherzustellen, dass die UI überall korrekt funktioniert.
- Tool wird bereits verwendet – Knowledge muss nicht erst aufgebaut werden und die Pflege eines weiteren Tools ist nicht nötig
- Große aktive Community – hoher Google-Faktor

Was passiert *vor* und *bei* der Testausführung genau?

Abbildung 2 zeigt bereits den Ausschnitt einer Testausführung in einer CI/CD-Pipeline. Doch was passiert da eigentlich genau?

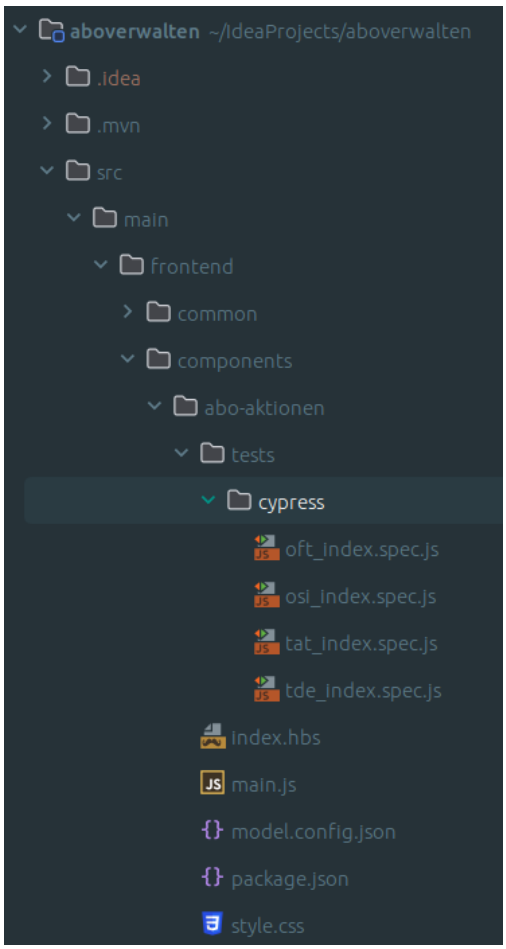

Abb. 4: Eine UI-Komponente mit Tests in der IDE (Quelle: Eigene Darstellung)

In Abbildung 4 ist eine Komponente in der IDE zusehen, welche bereits unter „*tests/cypress*“ für vier Mandanten eine Test-Suite samt Test-Cases hinterlegt hat. Da es sich hier um Tests auf Unit-Test-Ebene handelt, müssen diese nicht unbedingt für nicht-Entwickler*innen lesbar sein, weswegen an dieser Stelle auf einen Cucumber-Vorbau verzichtet wurde, welcher aber durchaus möglich wäre.

Statische Testumgebung aufbauen

Bevor die Tests gestartet werden können, wird die zu testende Komponente, die sich im Wesentlichen hinter der *index.hbs* verbirgt, in eine statische Seite überführt. Der konkrete Zustand dieser Seite wird über die *model.config.json* konfiguriert. Diese erlaubt es, mehrere Varianten (*variants*) derselben Komponente zu definieren, sodass umfangreiches Testen möglich ist. Beispielsweise werden in dieser Komponente Kund*innen unterschiedliche Aktionen angeboten, abhängig davon, ob es sich um ein aktives, pausiertes oder gekündigtes Abo handelt. Mithilfe der Variante können all diese Zustände bespielt werden. Wenn die statischen Seiten gebaut sind, sind diese im Ordner

„target/classes/hbs-templates/[service]/resources/[componente]“

auffindbar.

Express.js liefert die Testseiten aus

Wenn nun der Express.js Server gestartet wird, ist es möglich über den Browser die Komponenten, die uns als Testbasis dienen, aufzurufen:

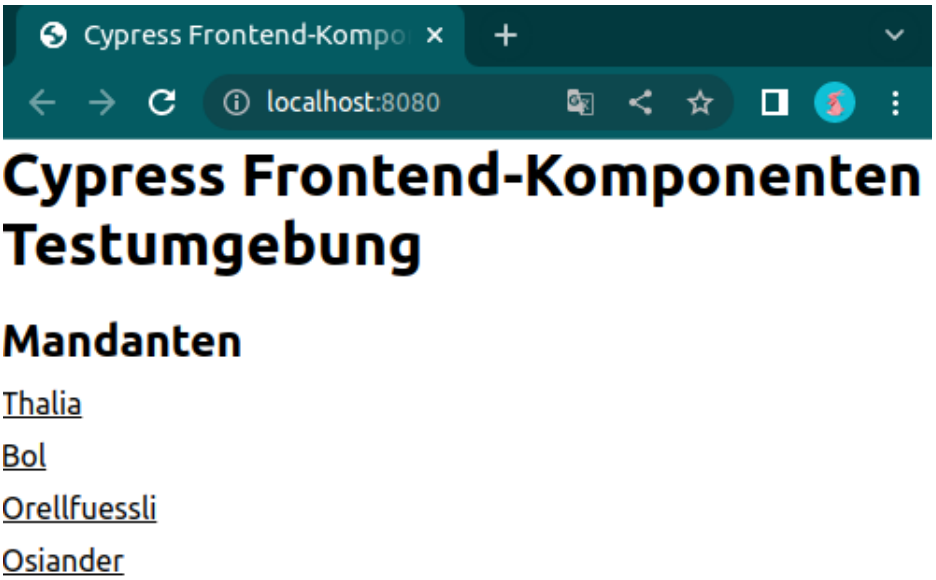

Abb. 5: Die Mandanten in der Testumgebung (Quelle: Eigene Darstellung)

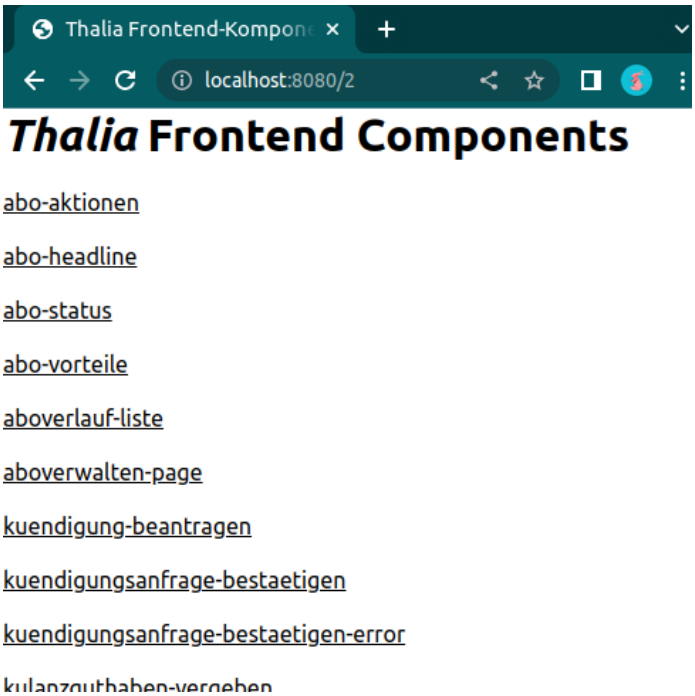

Abb. 6: Komponenten in der Testumgebung (Quelle: Eigene Darstellung)

Abbildung 5 zeigt die Mandanten, für die unsere Testseiten gebaut wurden. Auf Abbildung 6 hingegen sind die einzelnen Frontend-Komponenten des Services zusehen. Dahinter verbirgt sich dann die Seite, gegen die Cypress im Anschluss unsere Tests laufen lässt. Dies geschieht in der Pipeline natürlich *headless*, aber während der Implementierung haben Entwickler*innen die Möglichkeit, die Testausführung im Browser zu betrachten. Mit einem *watcher* werden die statischen Komponenten bei jeder Änderung automatisch neu gebaut und die Tests direkt neu gestartet, sodass Entwickler*innen die Möglichkeit haben bequem Test-Driven oder Test-First zu arbeiten.

Aufbau und Organisation der Tests

UI-Component Tests mit Cypress werden mithilfe von sogenannten Test-Suites organisiert, die für jeden Mandanten nach Notwendigkeit angelegt werden können. Eine Test-Suite ist eine Sammlung von Tests, die alle ein bestimmtes Ziel verfolgen. Jede Test-Suite besteht aus einem oder mehreren Test-Cases, die wiederum aus einer Reihe von Test-Schritten bestehen.

Test-Suites können für jeden benötigten Mandanten in einem separaten Test-File angelegt werden. Ob dies nötig ist, entscheidet die Test-Strategie und schlussendlich die Entwickler*innen. Bei simplen Komponenten, die wenig mandantenspezifisches Verhalten aufweisen, ist es häufig sinnvoll, um Redundanz zu vermeiden, in weiteren Test-Suites nur genau diese zu testen oder ganz auf diese zu verzichten.

```
describe( description: "Tests für <abo-aktionen> auf Thalia.de", specDefinitions: () => {

    beforeEach( action: () => {
        cy.visit("2/abo-aktionen");
    });
    it( expectation: "Hier definieren was der Test prüft", assertion: () => {
        cy.get('[data-test="HelloWorld"]') ...
            .should( chainer: 'be.visible') Chainable<Subject>
            .and( chainer: 'contain.text', value: 'HelloWorld')
    });
```

Abb. 7: Eine Test-Suite mit einem simplen Test-Case als Beispiel (Quelle: Eigene Darstellung)

Eine Test-Suite beginnt mit der Beschreibung über ein *describe(),* das beispielsweise die getestete Komponente und den Mandanten enthalten sollte (s. Abb. 7): Hier die Komponente *abo-aktionen* und der Mandant Thalia.de. Vor jedem Lauf wird die Testumgebung der Komponente aufgerufen, das passiert

über

```
cy.visit("2/abo-aktionen")
```

im *beforeEach()*-Block. Die zwei im Pfad steht dabei für den Mandanten Thalia.de. Das *it()* kennzeichnet den Beginn eines Test-Case. Dabei haben Entwickler*innen die Möglichkeit zu beschreiben, was genau der Test prüft. Innerhalb der geschweiften Klammern sind die Test-Steps definiert. Im vereinfachten Beispiel wird das HTML-Object mit dem Selektor *data-test“HelloWorld“* aufgerufen und dann geprüft, ob dieser sichtbar ist und einen bestimmten Text enthält. Hier sind Entwickler*innen fast keine Grenzen gesetzt, wie die offizielle Dokumentation zeigt.

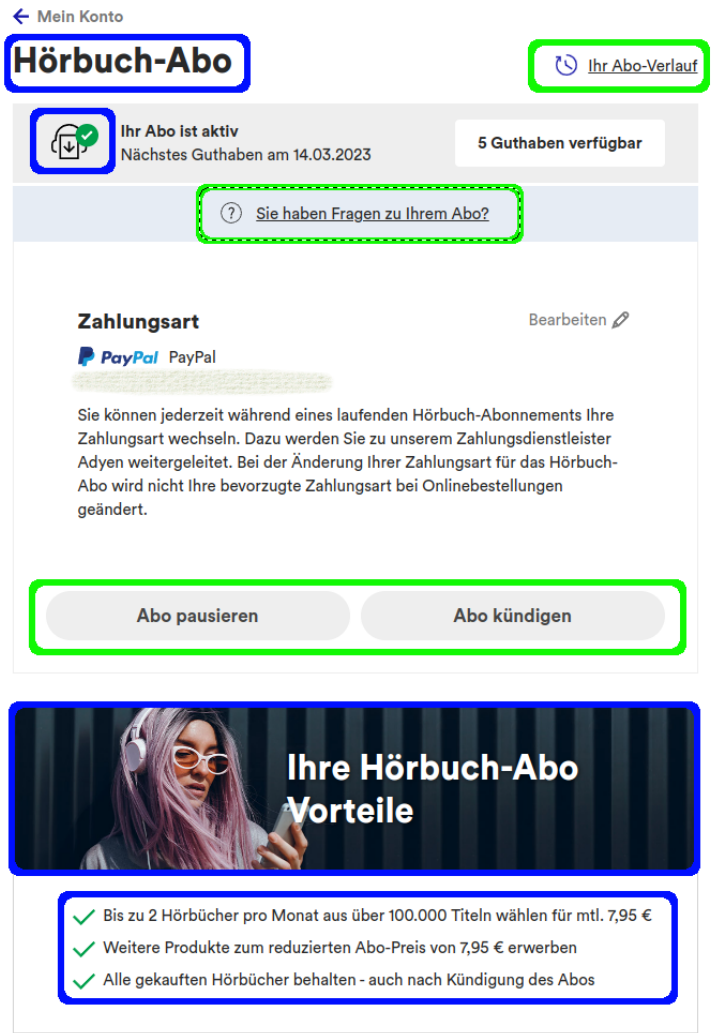

Abb. 8: Persönliche Hörbuch-Abo Seite (Quelle: Eigene Darstellung)

Was bedeutet das jetzt konkret? Beispiel!

Zu sehen (s. Abb. 8) ist die Seite in unserem Shop, auf der Kund*innen den aktuellen Status ihres Hörbuch-Abonnements einsehen und verwalten können. Die Darstellung wurde bisher vollständig durch deutlich langsamere E2E-Tests geprüft. Da es Kund*innen mit Abonnements in fünf verschiedenen Status (aktiv, pausiert, gekündigt, beendet oder ohne Abonnement) geben kann erhöht sich die Laufzeit stark, denn die funktionalen Tests mussten zusätzlich die korrekte Darstellung sicherstellen. Dadurch wurden die Tests massiv aufgebläht, was wiederum zu einer verdoppelten Laufzeit und mehr unzuverlässigen Tests führte. Wartbarkeit wurde zu einem Problem, das die Feature-Entwicklung verzögerte oder sogar dazu führte, dass Tests vorübergehend deaktiviert wurden. Durch die Einführung der UI-Component Tests konnten wir die blauen markierten Flächen in den E2E-Tests komplett ignorieren. Die grünen Flächen spielen weiterhin eine Rolle, allerdings nur funktional. Das heißt wir prüfen hier nicht auf Darstellung und/oder Wording, sondern gehen nur den Kundenprozess durch und schauen nicht abseits des Weges.

Im Rückspiegel

UI-Component Tests können ein wichtiger Bestandteil der Teststrategie sein. Für uns ist dabei Cypress das richtige Tool. Es ermöglicht eine schnelle und einfache Testdurchführung und lässt sich problemlos in CI/CD-Pipelines einbauen und automatisieren. Ein wichtiger Faktor bei der Tooleinführung in unserem Team war ein niedrigschwelliger Einstieg, der durch die Verwendung von Cypress als E2E-Testing Tool gegeben war. Aber es sollte eine kritische Frontend-Masse gegeben sein, damit sich eine Einführung lohnt, denn es erfordert schon ein Anpassen des Arbeitsablaufs bei der Frontend-Entwicklung. Das muss aber nicht unbedingt negativ sein, denn es bietet, mit dem richtigen Setup, auch die Chance, Test-Driven oder Test-First Development zu betreiben. Schlussendlich wird zunächst zusätzliche Arbeitskraft durch das Etablieren einer zusätzlichen Testebene investiert, wodurch aber ein Vielfaches eingespart wird, indem weniger Fehlermeldungen aufkommen und effektiver entwickelt wird.