Web-Anwendungen mit Cypress testen

Achim Heynen 25.02.2025

Umfrage:
Welche Tests kennt ihr?
Wie setzt ihr sie ein?

https://www.cypress.io

- Testframework für Web-Anwendungen
- Unterstützt E2E-Tests für jegliche Web-Anwendungen
- Unterstützt Komponententests für React, Angular, Vue & Svelte
- Kostenlos bei lokal ausgeführten Tests oder selbst gehostetem Service
- Kostenpflichtige Cloud-Lösung mit weiteren Funktionen verfügbar

Demo

Kurzer Überblick über die Cypress App

Warum Cypress?

- Verschiedene Arten von Tests möglich
- Ausführbar per Command Line oder interaktiver UI
- „Time Travel“ - jeder Test kann schrittweise nachvollzogen werden
- Automatisches Warten auf Zustände und Ereignisse
- Manipulation von Netzwerkkommunikation
- Tests lassen sich in verschiedenen Browsern und Display-Größen starten
- Tests und Anwendung lassen sich debuggen
- Tests/Fails können als Bild und als Video gespeichert werden
- Gute Einbindung in Continuous Integration

Cypress installieren

RTFM!!! 😄

- Quasi One-Click-Installation
- Sehr gute Dokumentation auf der Webseite
- Ausführliche Best-Practices und Beispiele mit einer „Real World App“
- Verschiedene Installationsarten für E2E- und Komponententests
- Verschiedene unterstützte Browser (Chrome/Chromium, Firefox, Electron)

Core-Concepts

Queries, Assertions & Actions

- Mit Query-Befehlen lässt sich nach DOM-Elementen und Daten des Browsers suchen
- Syntax zur Suche nach DOM-Elementen ist jQuery sehr ähnlich
- Assertions prüfen Daten aus Query-Ergebnissen
- Zentraler Befehl ist should(...) mit Chainer-Angaben in Syntax von Chai
- Actions führen eine Aktion auf einem Query-Ergebnis aus

```
<input />
<input class="bar" />
<input class="foo bar" />
```

```
cy.get('input.foo')
  .should('have.class', 'bar')
  .type('jou')
```

Sehr verständliche Doku unter https://docs.cypress.io/api

Core-Concepts

Async Execution & Chaining

- Cypress-Commands werden asynchron ausgeführt!

```
it('hides the thing when it is clicked', () => {
  cy.mount(MyComponent) // Hier passiert nichts

  cy.get('.hides-when-clicked') // und hier auch nicht
    .should('be.visible') // immer noch nichts
    .click() // Nein, nichts

  cy.get('.hides-when-clicked') // gääähn
    .should('not.be.visible') // auch nicht
})
```

- Die Testfunktion packt alle cy-Commands in eine Queue
- Die Queue wird erst nach der Testfunktion abgearbeitet

Core-Concepts

Async Execution & Chaining

- Abfolgen werden durch Chaining/Closures erreicht

```
const button = cy.get('button')
const form = cy.get('form')

button.click() // wird zu früh aufgerufen
```

```
cy.get('button').then(($btn) => {
  // wenn get erfolgreich war,
  // wird mit $btn das gefundene
  // Objekt übergeben
})
```

```
const button = cy.get('button')
...
cy.then(() => {
  button.click()
})
```

Core-Concepts

Async Execution & Chaining

```
it('does not work', () => {
  let username = undefined

  cy.visit('https://example.cypress.io')
  cy.get('.user-name')
    .then(($el) => {
      username = $el.text()
    })

  if (username) {
    cy.contains(username).click()
  } else {
    cy.contains('My Profile').click()
  }
})
```

Ist direkt ‚undefined'

Wird unmittelbar beim Einlesen des Tests ausgewertet und ist damit immer ‚undefined'

Wird deshalb immer ins ‚else' laufen

Core-Concepts

Async Execution & Chaining

```
it('does work', () => {
  let username = undefined

  cy.visit('https://example.cypress.io')
  cy.get('.user-name')
    .then(($el) => {
      username = $el.text()

      if (username) {
        cy.contains(username).click()
      } else {
        cy.get('My Profile').click()
      }
    })
})
```

Ist direkt ‚undefined'

Wird erst ausgeführt, wenn das ‚get' auflöst und schreibt dann den Inhalt in ‚username'

Klickt auf den Benutzernamen oder wenn nicht gesetzt auf ‚My Profile'

Eigene cy-Commands können ergänzt werden

Core-Concepts

Intercepts

- Netzwerkverkehr kann simuliert oder verändert werden
  - ➡ sowohl Anfragen als auch Antworten
  - ➡ mit Assertions kombinierbar

```
// spying
cy.intercept('/users/**')
cy.intercept('GET', '/users*')
cy.intercept({
  method: 'GET',
  url: '/users*',
  hostname: 'localhost'
})
```

```
// spying and response stubbing
cy.intercept('POST', '/users*', {
  statusCode: 201,
  body: {
    name: 'Peter Pan',
  },
})
```

```
// spying, dynamic stubbing, request modification, etc.
cy.intercept(
  { method: 'GET', url: '/users*' },
  (request) => {
    /* do something with request and/or response  */
  }
)
```

Core-Concepts

Aliases

- Schneller Zugriff auf diverse Daten:
  - Query-Ergebnisse
  - Fixtures
  - Intercepts
  - Alles was sich wrappen lässt

```
beforeEach(() => {
  cy.fixture('users.json').as('users')
})

it('utilizes users in some way', function () {
  cy.get('@users').then((users) => {
    const user = users[0]

    cy.get('header').should('contain', user.name)
  })
})
```

Core-Concepts

Aliases

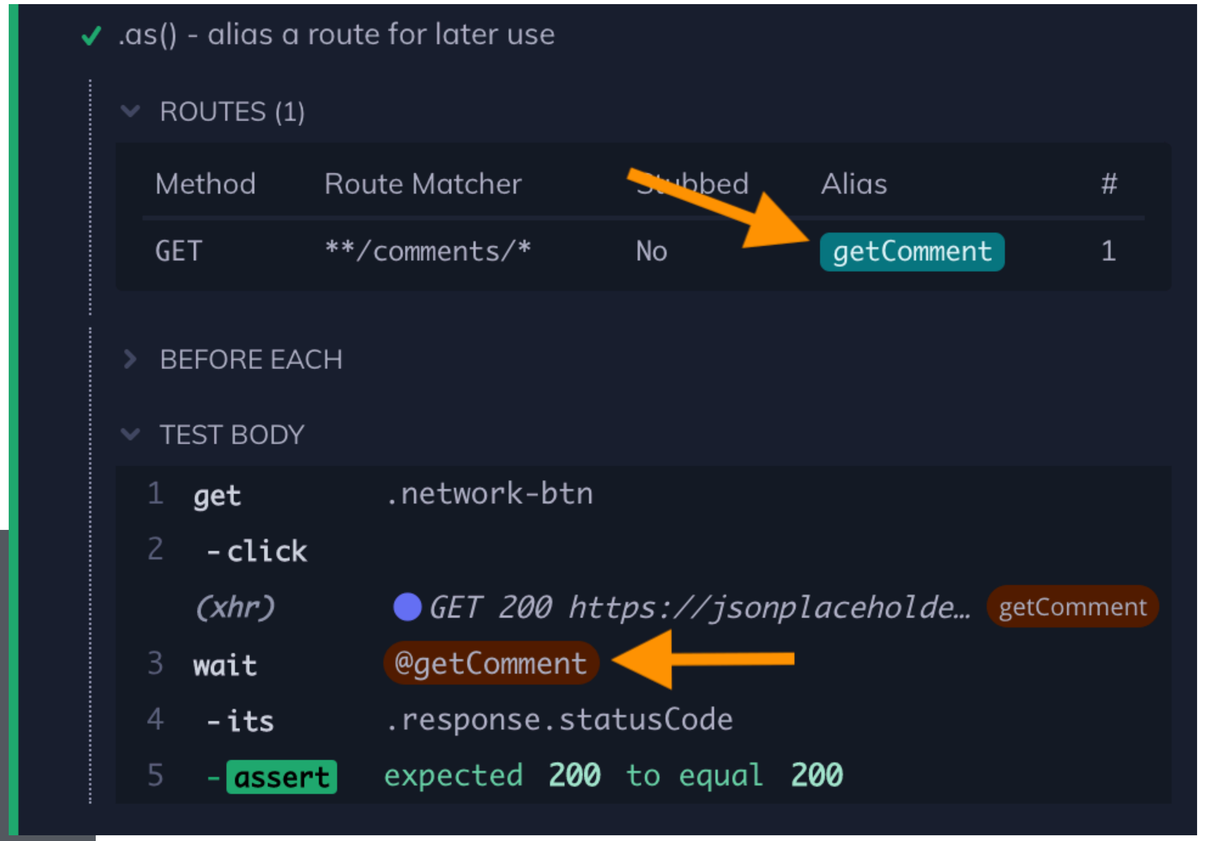
- Vorteile:
  - Shortcut für komplexe Commands
  - Bei Queries: Retryability
  - Bei Intercepts: Warten auf Requests

```
cy.intercept('POST', '/users', { id: 123 })
  .as('postUser')

cy.get('form').submit()

cy.wait('@postUser').then(({ request }) => {
  expect(request.body)
    .to.have.property(,name', 'Brian')
})

cy.contains('Successfully created user: Brian')
```



Core-Concepts

Stub/Spy/Clock

- Programmatische Manipulation
  - Stubs können Methoden von JS-Objekten ersetzen
  - Spies können Aufrufe von Methoden überwachen
  - Mit Clock kann die Systemuhr verändert werden (simuliert)

Ähnlich wie Intercepts für JS-Methoden

```
cy.stub(obj, 'method')
  .withArgs(,bar')
  .returns(,foo')
```

```
cy.spy(util, 'addListeners')
App.start()
expect(util.addListeners)
  .to.be.called
```

```
cy.clock()
cy.visit('/index.html')
cy.tick(1000)
cy.get(,#seconds-elapsed')
  .should('have.text', '1 seconds')
cy.tick(1000)
cy.get(,#seconds-elapsed')
  .should(,have.text', '2 seconds')
```

Verschiedene Testtypen

- Unit-Tests
- Component-Tests
- E2E-Tests
- Integration-Tests

Verschiedene Testtypen

Unit-Tests

- Prüfen eine abgeschlossene Code-Einheit
- Viele kleine Tests, jeder davon mit einem konkreten Ziel
- Lassen sich sehr schnell ausführen
- Können Funktionen testen, die in der Praxis nicht/schwer erreicht werden
- Asynchrone Interaktion durch z.B. Http-Requests möglich, aber eher aufwändig

Verschiedene Testtypen

Component-Tests

- Spezielle Form von Unit-Tests, testen eine Komponente
- Unterstützung in Cypress für bestimmte JS-Frameworks
- Komponenten werden im Browser sichtbar gerendert
- Alle interaktiven Funktionen von Cypress verfügbar

Vorteile

- Kleine, schnelle Tests
- Gute Abgrenzung/Kapselung
- Visuell nachvollziehbar

Nachteile

- Auf bestimmte Frameworks begrenzt
- Einschränkungen, z.B. keine Navigation, keine Downloads, kein Seitentitel

Verschiedene Testtypen

E2E-Tests

- Testet eine (Web-)Anwendung in einem realen Umfeld
- Unterstützung in Cypress für Webseiten allgemein
- Tests laufen sichtbar im Browser
- Alle interaktiven Funktionen von Cypress verfügbar

Vorteile

- Keine Begrenzung auf bestimmte Frameworks
- Testet auch komplexe Abläufe
- Testet Interaktion mehrerer Systeme
- Visuell nachvollziehbar

Nachteile

- Langsamer als Component-Tests
- Testumgebung komplex
- Tests zur Entwicklungszeit schwierig

Verschiedene Testtypen

Integration-Tests

- Wie E2E-Tests mit simulierten Systemen

Vorteile

- Selbe Testmöglichkeiten wie E2E-Tests
- Testumgebung kann bei Bedarf simuliert werden
- Test zur Entwicklungszeit kein Problem

Nachteile

- Keine Garantie, dass APIs der Realität entsprechen

Verschiedene Testtypen

Unterschiedliche Ziele

Component-Tests

- Funktionen in minimalen Szenarien sicherstellen
- Tests von hypothetischen Zuständen ermöglichen
- Unmittelbare Unterstützung bei der Entwicklung

Integration-Tests

- Tatsächliche Abläufe einer Anwendung simulieren
- Zusammenspiel der Komponenten sicherstellen
- Unmittelbare Unterstützung bei der Entwicklung

E2E-Tests

- Abläufe unter realen Bedingungen sicherstellen
- Nachgelagerte Unterstützung

Software-Qualität!

Tests in Continuous Integration

- Cypress kann mit diversen CI-Tools genutzt werden
- Neue Funktionen und Änderungen werden direkt mit Tests belegt
- Features möglichst Component-Tests, sonst Integration-Tests
- Typische Abläufe/Use-Cases als E2E-Tests
- Component- und Integration-Tests werden unmittelbar auf neue Commits ausgeführt
- Pull/Merge-Requests benötigen „grüne Builds“, um gemerged zu werden
- E2E-Tests werden je nach Infrastruktur regelmäßig auf bestimmten Branches ausgeführt

E2E-Testumgebung

Problem:

- Externe Systeme/APIs müssen passen
- Datenstände/strukturen müssen passen
- Verschiedene Branches benutzen verschiedene Versionen

- Deutliche Vorteile bei Container-basierten Deployments der Systeme
- Eigene Software per Inhouse-Paketmanager bereitstellen
- Datenstand pflegen und immer neu einspielen ist aufwändig
- Stattdessen Test-Runs z.B. Daten mit Timestamps erstellen lassen
- „Unabhängige“ Tests schreiben, die nicht mit anderen Daten kollidieren

Was kann Cypress noch?

- Cypress Cloud bietet CI-Funktionen out-of-the-box
- Smart Orchestration: Parallele Ausführung, Test Priorisierung, Load-Balancing etc.

  - Cypress Cloud kostet (recht viel)
  - Die meisten Funktionen lassen sich mit eigener CI ebenfalls nutzen

- Accessibility-Tests
- UI-Coverage

  - Nur in Cypress Cloud verfügbar
  - Zusätzlich kostenpflichtig

Schreibt Tests!

Vielen Dank für die Aufmerksamkeit