End-to-End-Tests (E2E) überprüfen, ob ein Programm aus Sicht des Nutzers wie vorgesehen funktioniert.

Die Automatisierung ermöglicht es, die Tests für jeden neuen Stand zu wiederholen. Smart geschriebener Testcode findet Fehler, bevor ein Anwender beim Zugriff auf das Echtsystem darauf stößt.

In diesem Beitrag grenzen wir E2E-Tests theoretisch von anderen Testtypen ab und überlegen, weshalb sie sinnvoll sind. Die praktischen Beispiele bauen auf Joomla! [1] auf. Das CMS ermöglicht es, komplexere Problemstellungen aufzuzeigen. Die Verwendung einer fertigen Anwendung hat Vor- und Nachteile. Ein Minus ist, dass Joomla! installiert werden muss, um dem Beispielcode [2] praktisch zu folgen. Wer den Aufwand scheut, findet in den Abbildungen Orientierungshilfen zum Mitdenken.

Ich bin überzeugt, dass Tests, die

- möglichst zeitnah zur Programmierung,
- automatisch und
- häufig (idealerweise nach jeder Programmänderung)

durchgeführt werden, mehr bringen, als sie kosten. In diesem Beitrag möchte ich diejenigen Entwickler motivieren, die schon immer Tests für ihre Software schreiben wollten – es aber aus verschiedenen Gründen nie getan haben. Unter Umständen räumt Cypress Hindernisse aus dem Weg. Beginnen wir mit etwas Theorie.

Das magische Dreieck

Lohnt es sich, Zeit und Geld in das Programmieren von Tests zu investieren? Das magische Dreieck [3] beschreibt den Zusammenhang zwischen Kosten, Zeit und Qualität. Das Spannungsverhältnis zwischen diesen Faktoren wurde ursprünglich im Projektmanagement beschrieben. Dort stellte man fest, dass ein höherer Kostenaufwand positive Auswirkungen auf die Qualität und/oder den Fertigstellungstermin – die Zeit – hat (**Abb. 1**). Umgekehrt wird eine Kosteneinsparung die Qualität mindern und/oder die Fertigstellung verzögern (**Abb. 2**).

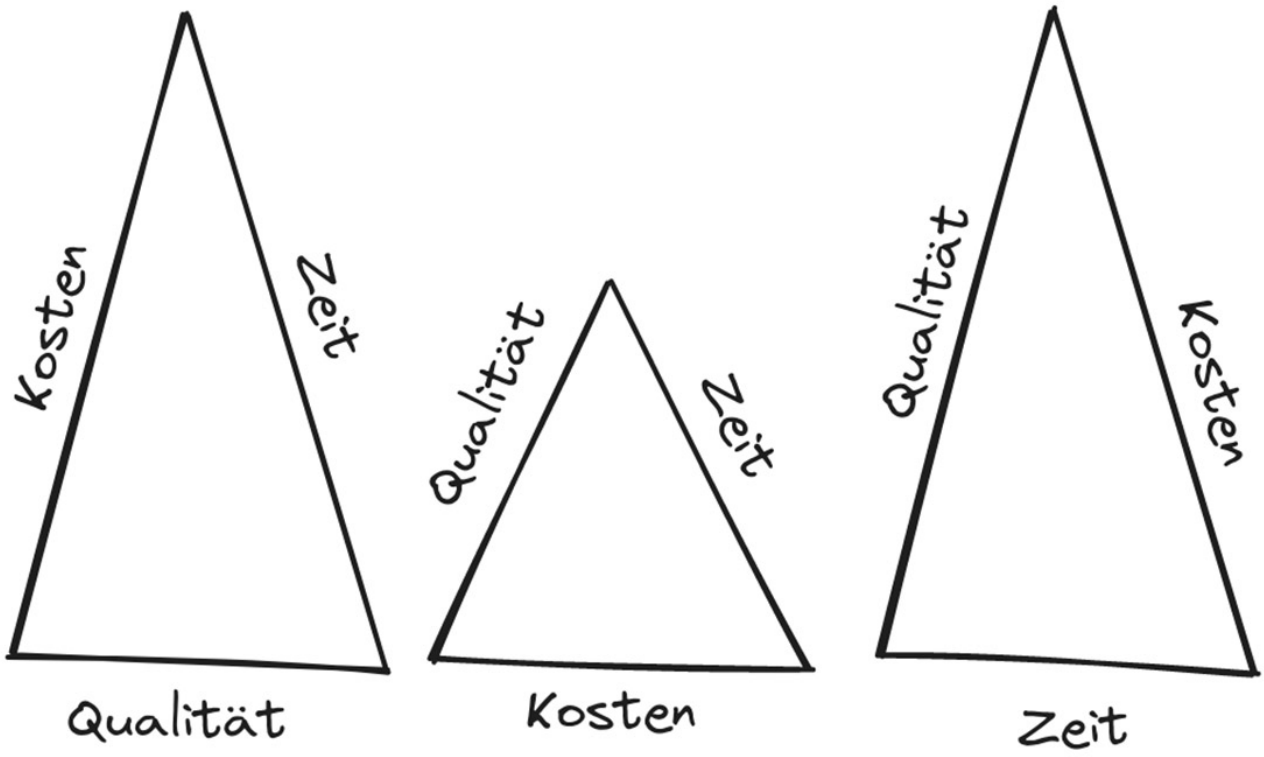

Abb.1: Das magische Dreieck: Mehr Geld wirkt sich positiv auf Qualität/Zeit aus

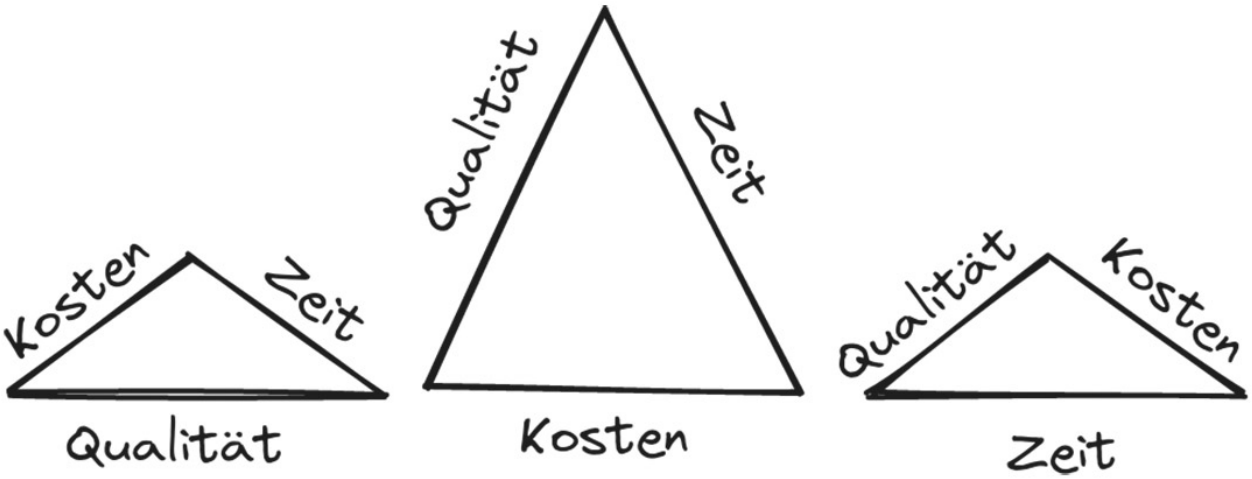

Abb. 2: Das magische Dreieck: Geringere Investitionen wirken sich negativ auf Qualität/Zeit aus

Jetzt kommt die Magie ins Spiel: Auf lange Sicht wird dieses Spannungsverhältnis überwunden. Unter Umständen haben Sie selbst einmal erlebt, dass das Sparen an der Qualität langfristig keine Kosten mindert. Die technische Schuld führt oft zu Mehraufwand (**Abb. 3**).

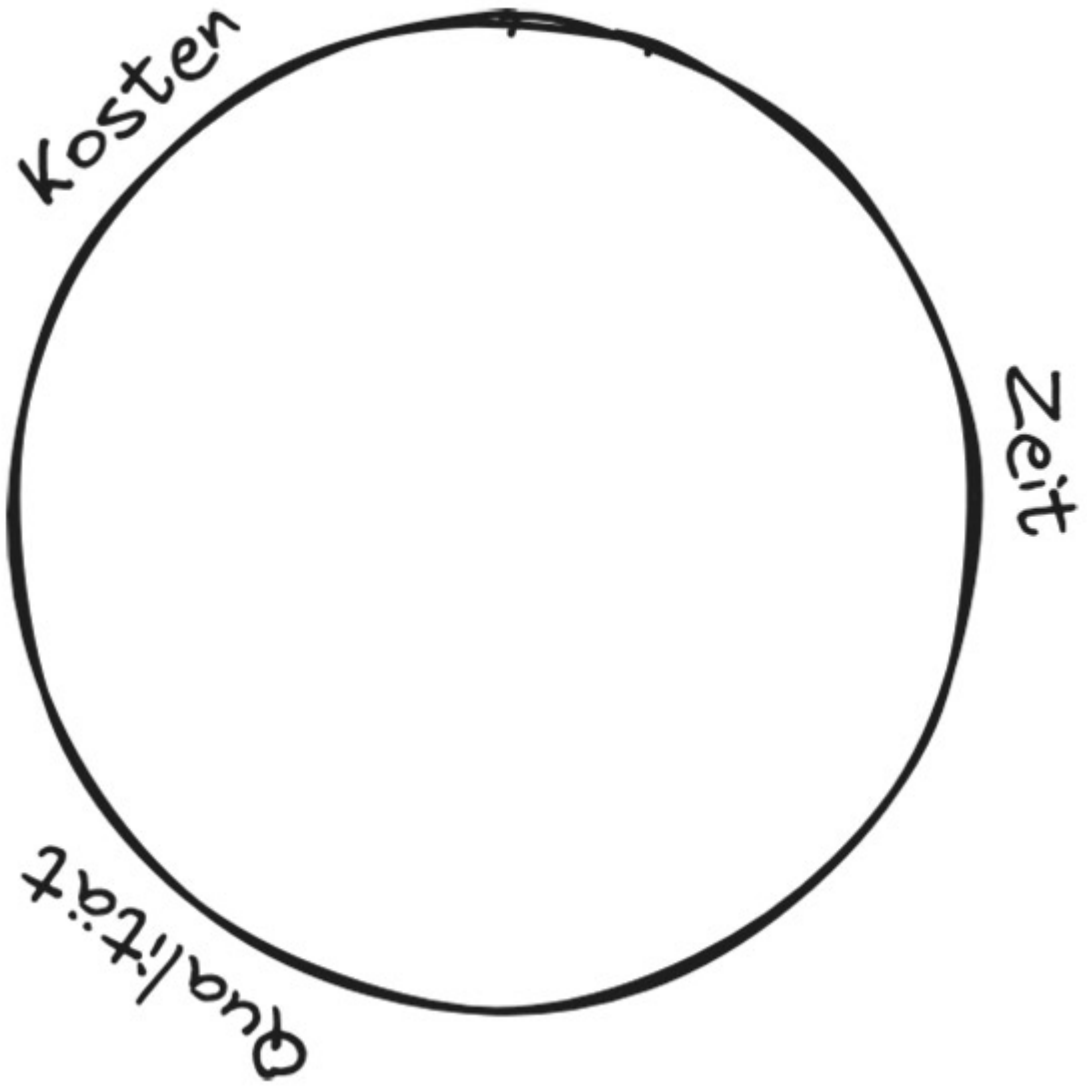

Abb. 3: Auf lange Sicht kann der Zusammenhang zwischen Kosten, Zeit und Qualität überwunden werden

Unter technischer Schuld versteht man den Aufwand, der bei der Änderung oder Erweiterung von minderwertig programmierter Software entsteht. Martin Fowler [4] unterscheidet vier Arten von technischen Schulden: solche, die man bewusst und solche, die man ungewollt eingegangen ist. Außerdem differenziert er zwischen vorsichtigen und risikofreudigen technischen Schulden (**Abb. 4**).

https://software-architecture-summit.de/session-qualification/sas-ber23-infoblock/?layout=contentareafeed&widgetversion=0&seriesId=aXkrJMRqaY6TETENr

Abb. 4: Technische Schuld

In der Literatur findet man immer wieder niederschmetternde Statistiken über die Erfolgsaussichten von Softwareprojekten. In den 1990er Jahren zeigte eine Studie von A. W. Feyhl [5], dass 70 Prozent der Projekte eine Kostenabweichung von mindestens 50 Prozent aufweisen. Sollte man also auf Kostenschätzungen verzichten und der Argumentation der #NoEstimates-Bewegung [6] folgen? Je mehr Erfahrungen ich sammle, desto mehr komme ich zu dem Schluss, dass extreme Ansichten nicht weiterhelfen. Die Lösung liegt meistens in der Mitte.

Wann ist nun der beste Zeitpunkt, Tests in ein Projekt zu integrieren? Werfen wir einen Blick auf die Kosten für die Behebung eines Fehlers in den verschiedenen Projektphasen (**Abb. 5**). Je früher Sie einen Fehler finden, desto geringer sind die Kosten für dessen Behebung. Zu Ende gedacht bedeutet das, dass es sinnvoll ist, Tests so früh wie möglich zu integrieren.

- Tests finden unbekannte Fehler während der Entwicklung. Das Auffinden der Fehlfunktion ist teuer. Lokalisierung und Korrektur sind billig.
- Debugger klären Fehler, die nach der Fertigstellung auftauchen. Das Auffinden der Fehlfunktion ist kostenlos. Lokalisierung und Behebung sind in der Regel teuer.

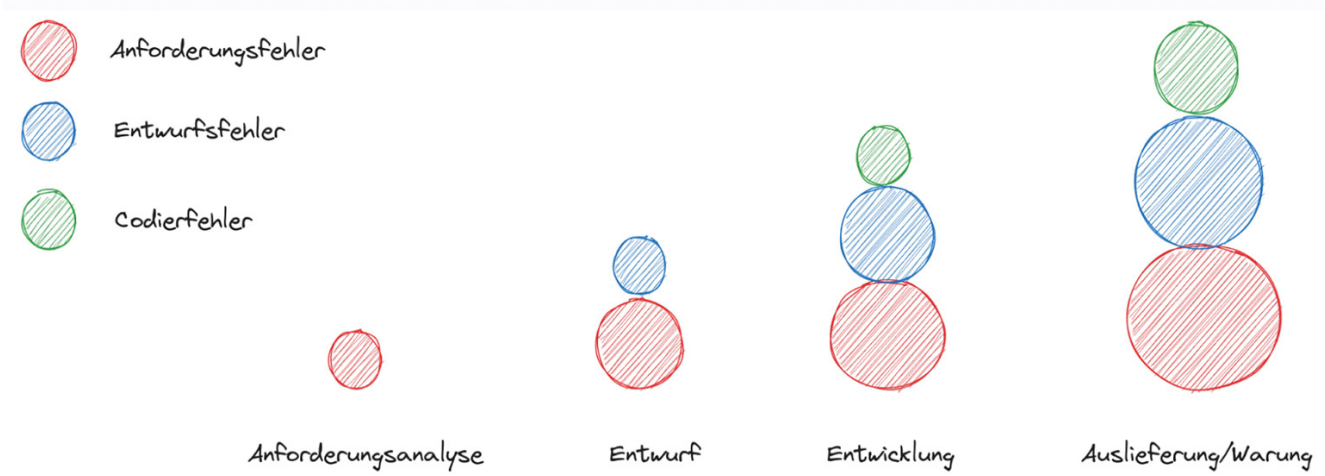

Abb. 5: Relative Kosten für die Fehlerbehebung in den unterschiedlichen Projektphasen

Kontinuierliche Integration (Continuous Integration, CI)

Es ist wichtig, jederzeit zu wissen, in welchem Zustand ein Softwareprojekt sich gerade befindet. Neuentwicklungen, die nicht in die bestehende Software passen, sollten erst dann integriert werden, wenn sichergestellt ist, dass sie keine negativen Auswirkungen auf das Gesamtsystem haben. In Zeiten, in denen immer häufiger Sicherheitsprobleme gefunden werden, sollte in einem Projekt jederzeit eine neue Version erstellt werden können. Und hier kommt die kontinuierliche Integration ins Spiel. CI integriert den neuen Code permanent. Die Software wird in kleinen Zyklen erstellt und getestet. Auf diese Weise stößt man frühzeitig auf mögliche Probleme und die Fehlersuche ist müheloser, da Fehler zeitnah zur

Programmierung entdeckt werden. Cypress bietet eine ausführliche Dokumentation zur Verwendung von CI mit verschiedenen CI-Anbietern [7].

Um sicherzustellen, dass man jederzeit Tests für alle Programmteile zur Verfügung hat, bietet sich die testgetriebene Entwicklung (Test-driven Development, TDD) an. Dabei handelt es sich um eine Programmiertechnik, bei der in kleinen Schritten entwickelt wird. Zuerst schreibt man den Testcode. Erst dann erstellt man den zu testenden Programmcode. Neue Tests schlagen zunächst fehl, weil die gewünschte Funktion nicht im Programm implementiert ist. Erst im zweiten Schritt erstellt man den Code, der den Test erfüllt. Der Merksatz dazu lautet: TDD-Tests helfen, das Programm richtig zu schreiben.

Wenn Sie zum ersten Mal von dieser Technik hören, werden Sie sich mit dem Konzept vielleicht nicht wohlfühlen. Viel lieber möchte man mit dem produktiven Teil beginnen. Probieren Sie es aus. Manchmal freundet man sich mit einer neuen Technik an, nachdem man sie kennengelernt hat. In Projekten mit hoher Testabdeckung fühle ich mich deutlich wohler.

Idealerweise wird zusammen mit TDD die verhaltensgesteuerte Entwicklung (Behaviour-driven Development, BDD) eingesetzt. Das ist eine Art Best Practice, bei der nicht die Implementierung des Programmcodes im Vordergrund steht, sondern das Verhalten des Programms. Ein Test prüft, ob die Anforderung des Anwenders erfüllt wird. Der Merksatz: BDD-Tests helfen, das richtige Programm zu schreiben.

Was meine ich damit? Es kommt vor, dass Anwender Dinge anders sehen als Entwickler. Der Arbeitsablauf beim Löschen eines Artikels im CMS Joomla! ist so ein Beispiel. Immer wieder treffe ich Anwender, die im Papierkorb auf das Statusicon klicken und sich wundern, dass der Beitrag danach wieder aktiv ist. Der Anwender geht intuitiv davon aus, dass das Element dauerhaft gelöscht wird. Für den Entwickler ist ein Klick auf

das Statussymbol ein Toggeln des Status. Aus Entwicklersicht ist die Funktion fehlerfrei implementiert. Stelle ich mich auf die Seite eines Benutzers, merke ich, dass das an dieser Stelle nicht die richtige Funktion ist.

Bei der verhaltensgesteuerten Entwicklung werden die Anforderungen an die Software durch Beispiele beschrieben, die als Szenarien oder User Stories bezeichnet werden. Merkmale der verhaltensgesteuerten Entwicklung sind

- die Einbindung des Anwenders in den Entwicklungsprozess,
- die Dokumentation mit Fallbeispielen in Textform,
- das automatische Testen dieser Beispiele und
- die sukzessive Implementierung.

Das Joomla!-Projekt hat BDD in einem Google-Summer-of-Code-Projekt [8] eingeführt. Man hoffte, dass Nutzer ohne Programmierkenntnisse mit Gherkin [9] leichter in die Entwicklung einbezogen werden könnten. Der Ansatz wurde nicht weiterverfolgt. Zu dieser Zeit verwendete Joomla! Codeception als Testwerkzeug. Mit Cypress ist auch die BDD-Entwicklung [10] möglich.

Planung

Es gibt verschiedene Arten von Tests:

- Ein Unit-Test prüft kleine Programmeinheiten unabhängig voneinander.
- Ein Integrationstest testet das Zusammenspiel der einzelnen Einheiten.
- E2E-Tests stellen sicher, dass ein Programm die definierte Aufgabe erfüllt.

Ebenso gibt es unterschiedliche Strategien: Top-down und Bottom-up sind zwei verschiedene Ansätze zur Darstellung

komplexer Sachverhalte (**Abb. 6**). Top-down geht Schritt für Schritt vom Abstrakten/Allgemeinen zum Konkreten/Speziellen. Beispiel: Ein CMS wie Joomla! stellt Websites im Allgemeinen in einem Browser dar. Konkret gibt es in diesem Prozess Teilaufgaben, beispielsweise das Löschen eines Elements aufgrund eines Klicks auf eine Schaltfläche. Bottom-up beschreibt die umgekehrte Richtung.

Nun kommt BDD ins Spiel. Es beinhaltet die Beschreibung des Verhaltens der Software in Form von Szenarien. Diese helfen bei der Erstellung von E2E-Tests.

Der übliche Ansatz für die Erstellung von Tests erfolgt von unten nach oben. Wer die verhaltensgesteuerte Softwareentwicklung bevorzugt, verwendet idealerweise die Top-down-Strategie und erkennt so Missverständnisse zwischen Endanwendern und Entwicklern frühzeitig.

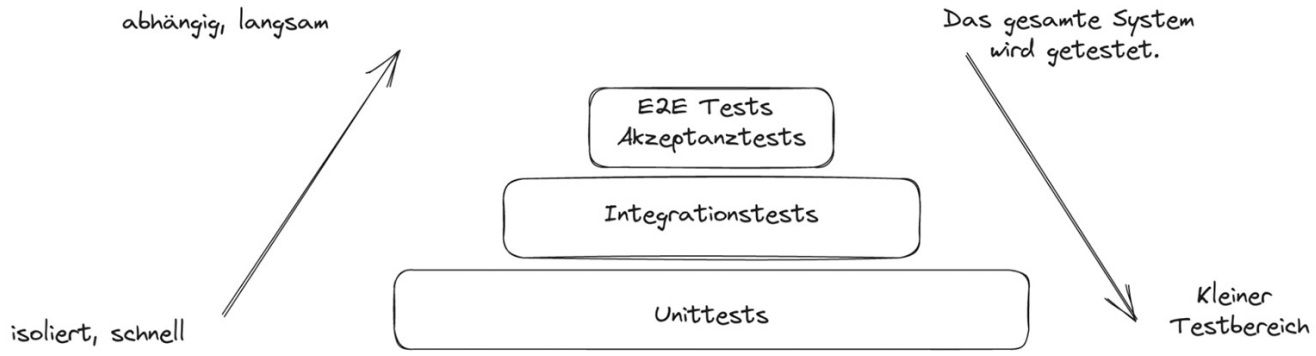

Abb. 6: Teststrategien – Top-down und Bottom-up

- Bei der Anwendung der Top-down-Strategie beginnt man mit den E2E-Tests. Der Schwerpunkt liegt darauf, zu testen, wie ein Benutzer mit dem System interagiert.
- Bei der Bottom-up-Strategie beginnt man mit Unit-Tests. Das Problem des Bottom-up-Ansatzes ist, dass es schwierig ist, zu testen, wie eine Komponente später in realen Umgebungen eingesetzt wird.

Aber wie viele Tests von welchem Typ wendet man nun sinnvollerweise an? Mike Cohns Testpyramide empfiehlt viele

Unit-Tests bei der Entwicklung von Softwareanwendungen. Integrations- und E2E-Tests sollten einen geringeren Anteil ausmachen. Auf diese Weise werden Fehler innerhalb einer Unit schnell aufgedeckt. Auf der mittleren Ebene befinden sich Integrationstests, die gezielt kritische Schnittstellen prüfen. An der Spitze der Pyramide stehen die langsamen E2E-Tests, die das Verhalten der Anwendung als Ganzes testen. Nach dieser theoretischen Einführung legen wir jetzt mit der Praxis los.

Cypress und Joomla! einrichten

Unser Beispiel baut bewusst auf dem CMS Joomla! auf. Die Entwicklerversion des CMS ist für die Arbeit mit Cypress konfiguriert. Außerdem gibt es implementierte Tests, an denen man sich orientieren kann.

Wir gehen folgende Schritte zur Einrichtung der lokalen Umgebung:

1. Klonen Sie das Repository in das Stammverzeichnis Ihres lokalen Webservers:

```
$ git clone https://github.com/joomla/joomla-cms.git
$ cd joomla-cms
```

1. Laut der Joomla!-Roadmap [11] wird die Major-Version 5.0 [12] im Oktober 2023 erscheinen. Ich baue auf dieser Entwicklungsversion auf. Wechseln Sie zum Branch *5.0-dev*:

```
$ git checkout 5.0-dev
```

1. Installieren Sie alle nötigen composer-Pakete:

```
$ composer install
```

1. Installieren Sie alle nötigen npm-Pakete:

```
$ npm install
```

Wenn Sie diese Schritte gegangen sind, ist alles fertig konfiguriert. Nur die individuellen Daten sind in der Datei *joomla-cms/cypress.config.js* anzupassen. Orientierung bietet die Vorlage *joomla-cms/cypress.config.dist.js*. Weitere Informationen stellen Joomla! [13] und Cypress [14] auf ihren Websites bereit.

Cypress verwenden

Rufen Sie *npm run cypress:open* via CLI im Joomla!-Stammverzeichnis auf. Kurze Zeit später öffnet sich die Cypress-App (**Abb. 7**). Dass Cypress die zuvor erstellte Datei *joomla-cms/cypress.config.dist.js* lädt, erkennt man daran, dass E2E Testing als konfiguriert markiert ist (**Abb. 7**).

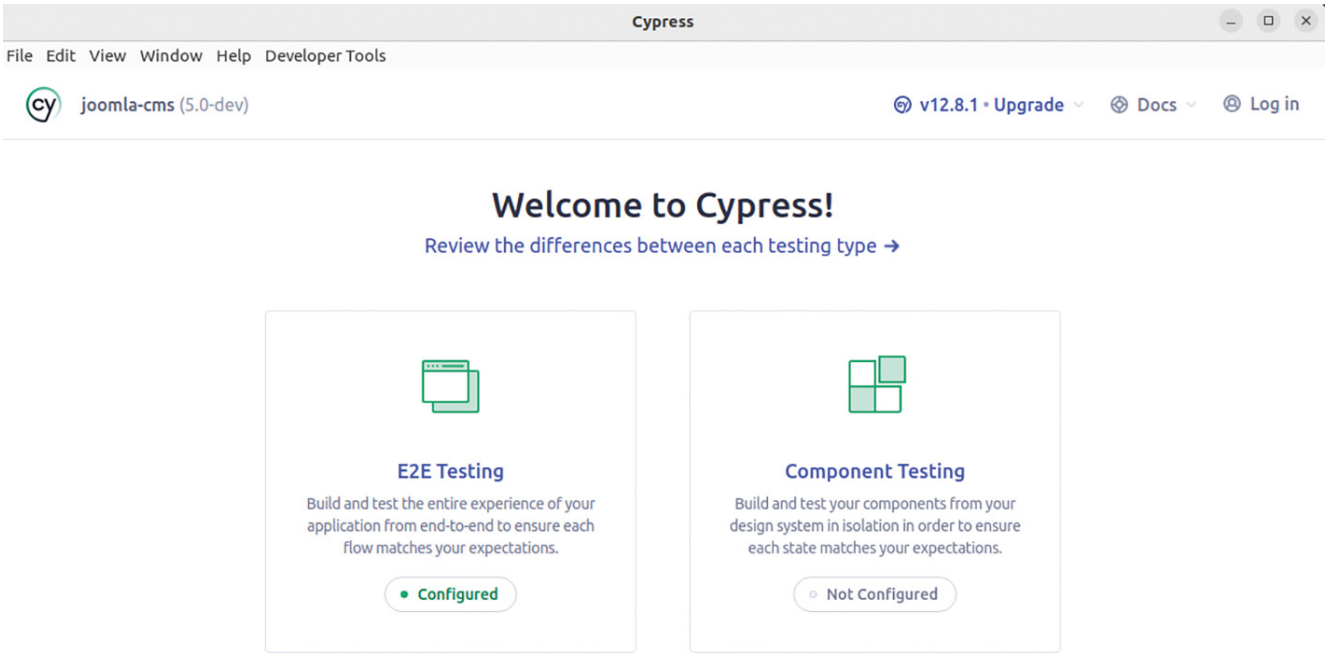

Abb. 7: Cypress-App öffnet sich nach dem Aufruf von npm run cypress:open

Nach dem Klick auf E2E Testing selektiert man den bevorzugten Browser. Für dieses Beispiel habe ich die Option Start Testing

in Firefox gewählt (**Abb. 8**).

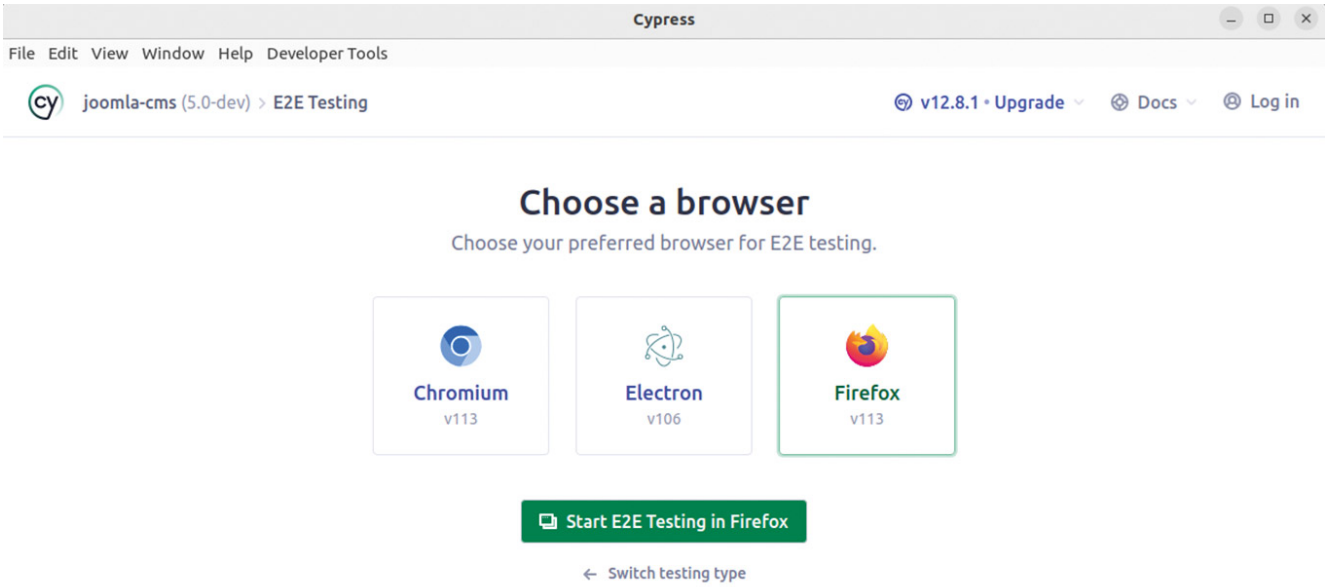

Abb. 8: E2E-Tests in der Cypress-App: Wähle den zu verwendenden Browser

Die nächste Ansicht listet alle implementierten Tests auf (**Abb. 9**). Wenn man eine Datei per Mausklick auswählt, werden die darin enthaltenen Tests aufgerufen. Der Ablauf ist im Browser live sichtbar.

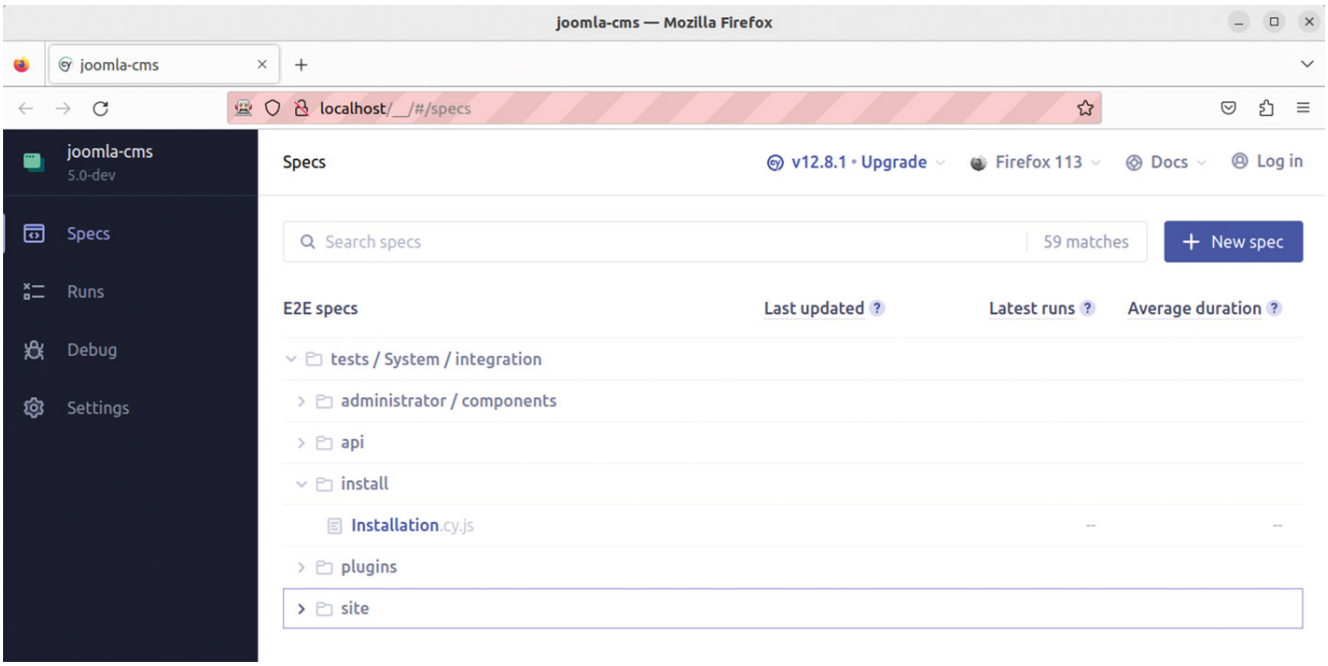

Abb. 9: Joomla!-Testreihe in Firefox über Cypress-App

Während die Tests ablaufen, sieht man links den Testcode und rechts das visuelle Ergebnis (**Abb. 10**). Klickt man später in

der Historie zurück, kann man den HTML-Code zu diesem Zeitpunkt analysieren.

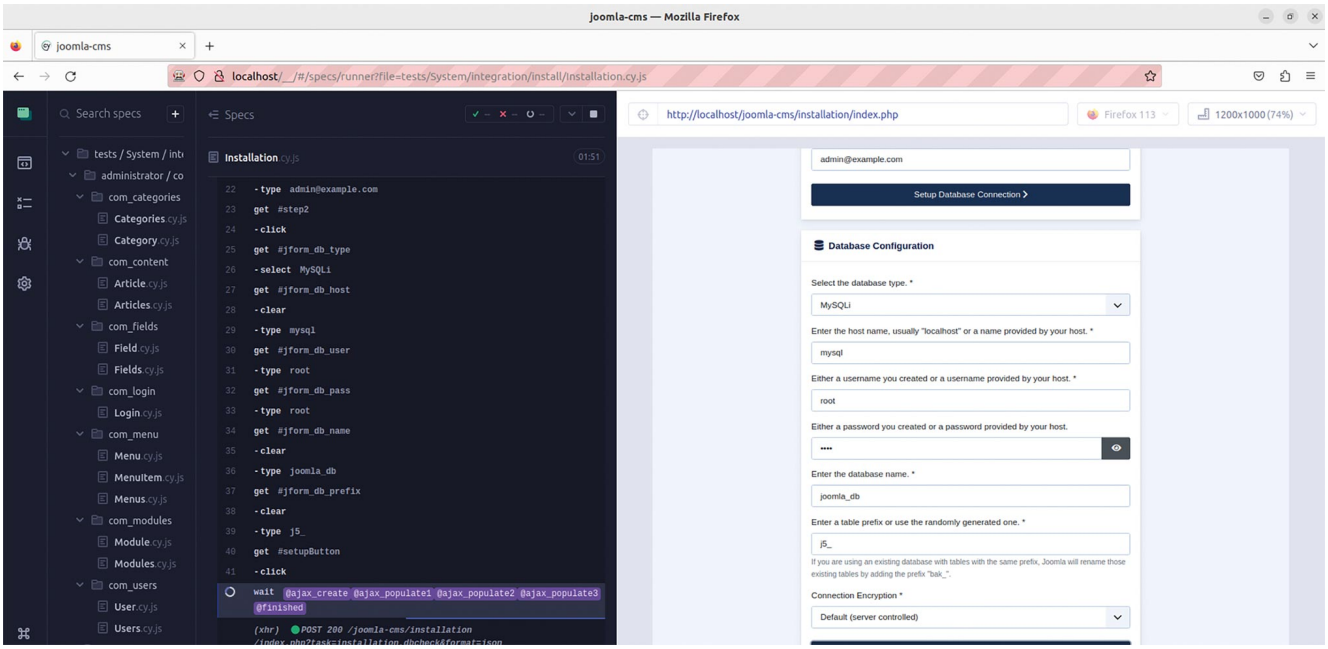

Abb. 10: Der Test, der die korrekte Installation von Joomla! sicherstellt, während der Ausführung

Probieren Sie es aus. Starten Sie den Test, der die korrekte Installation sicherstellt. Als Ergebnis verfügen Sie über ein System, auf dem wir aufbauen können.

Falls Sie als *db_host* eine externe Ressource verwenden, funktioniert der Test nicht. Installieren Sie in diesem Fall Joomla! manuell mit den Angaben, die in der Datei *joomla-cms/cypress.config.js* eingetragen sind.

Standardmäßig ruft *cypress run* alle Tests headless [15] auf. Der Befehl *$ npm run cypress:runin* führt die implementierten Tests aus und speichert im Fehlerfall Screenshots im Verzeichnis */joomla-cms/tests/cypress/output/screenshots*.

Weitere CLI-Befehle

Es gibt weitere hilfreiche Befehle, die nicht in der *package.json* des Joomla!-Projektes als Skript implementiert sind. Ich führe diese via npx [16] aus.

Der Befehl *cypress verify* überprüft, ob Cypress korrekt installiert ist:

```
$ npx cypress verify

✔  Verified Cypress! /.../Cypress/12.8.1/Cypress
```

Der Befehl *cypress info* gibt Informationen über Cypress und die aktuelle Umgebung aus (Listing 1).

Listing 1

```
$ npx cypress info
Displaying Cypress info...
Detected 2 browsers installed:
1. Chromium
 - Name: chromium
 - Channel: stable
 - Version: 113.0.5672.126
...
```

Die Dokumentation [17] bietet zu jedem Befehl ausführlichere Informationen.

Der erste Test

In der Entwicklungsversion des Joomla!-CMS gibt es fertige Cypress-Tests im Verzeichnis */tests/System/integration*. Diejenigen, die gern am Beispiel lernen, finden hier einen Einstieg.

Das Joomla!-Projekt stellt mit joomla-cypress [18] auch Code für gängige Testfälle bereit. Diese werden bei der Installation der Entwicklungsversion des CMS via *npm install* per *package.json* (Listing 2) und während der Entwicklung der Tests per Support-Datei */tests/System/support/index.js* importiert. Der Speicherort der Support-Datei ist in der Datei *cypress.config.js* festgelegt.

Listing 2

```
// package.json

{
  "name": "joomla",
  "version": "5.0.0",
  ...
  "devDependencies": {
    ...
    "joomla-cypress": "^0.0.16",
    ...
  }
}
```

Ein Beispiel für eine Funktion, die via joomla-cypress bereitgestellt wird, ist der Klick auf eine Schaltfläche in der Toolbar. Der Aufruf *Cypress.Commands.add(‚clickToolbarButton‘, clickToolbarButton)* bewirkt, dass in den eigenen Tests der Befehl *clickToolbarButton()* zur Verfügung steht. *cy.clickToolbarButton(’new‘)* simuliert einen Mausklick auf den Button New. Der hierfür erforderliche Code ist in Listing 3 auszugsweise dargestellt. Listing 4 zeigt ein weiteres Beispiel, die Anmeldung im Administrationsbereich.

Listing 3

```
// /node_modules/joomla-cypress/src/common.js
...
const clickToolbarButton = (button, subselector = null) => {
  switch (button.toLowerCase())
  {
    case "new":
      cy.get("#toolbar-new").click()
      break
      ...
  }
}
Cypress.Commands.add('clickToolbarButton', clickToolbarButton)
...
```

Listing 4

```
// /node_modules/joomla-cypress/src/user.js
...
const doAdministratorLogin = (user, password, useSnapshot =
true) => {
  cy.visit('administrator/index.php')
  cy.get('#mod-login-username').type(user)
  cy.get('#mod-login-password').type(password)
  cy.get('#btn-login-submit').click()
  cy.get('h1.page-title').should('contain', 'Home Dashboard')
}

Cypress.Commands.add('doAdministratorLogin',
doAdministratorLogin)
...
```

Aufgaben, die in der individuellen Umgebung häufig vorkommen, befinden sich im Verzeichnis */tests/System/support*. Sie werden über die Support-Datei */tests/System/support/index.js* importiert. Ein Beispiel ist die Anmeldung im Administrationsbereich mit den konkreten Log-in-Daten. Die zuvor behandelte Funktion *doAdministratorLogin* im Paket joomla-cypress erfordert die Angabe der Log-in-Daten beim Aufrufen. Die Datei */tests/System/support/commands.js* bietet mit der Funktion *doAdministratorLogin* die Möglichkeit, auf die Angabe der Daten zu verzichten. Listing 5 zeigt, wie die Log-in-Daten aus der Konfiguration *cypress.config.js* genutzt werden. *Cypress.env(‚username')* wird mit dem Wert der Eigenschaft *username* in der Gruppe *env* belegt. Außerdem sehen wir, wie man Befehle überschreibt. *Cypress.Commands.overwrite(‚doAdministratorLogin' …)* tritt an Stelle des Codes im Paket joomla-cypress.

Listing 5

```
// /tests/System/support/commands.js
...
Cypress.Commands.overwrite('doAdministratorLogin',
(originalFn, username, password, useSnapshot = true) => {
  const user = username ?? Cypress.env('username');
  const pw = password ?? Cypress.env('password');
```

```
  if (!useSnapshot) {
    Cypress.session.clearAllSavedSessions();
    return originalFn(user, pw);
  }
  return cy.session([user, pw, 'back'], () => originalFn(user,
pw), { cacheAcrossSpecs: true });
});
...
```

Damit wir über eigenen Code zum Testen verfügen, installieren wir eine simple Beispielkomponente [19] über das Joomla!-Backend (**Abb. 11**).

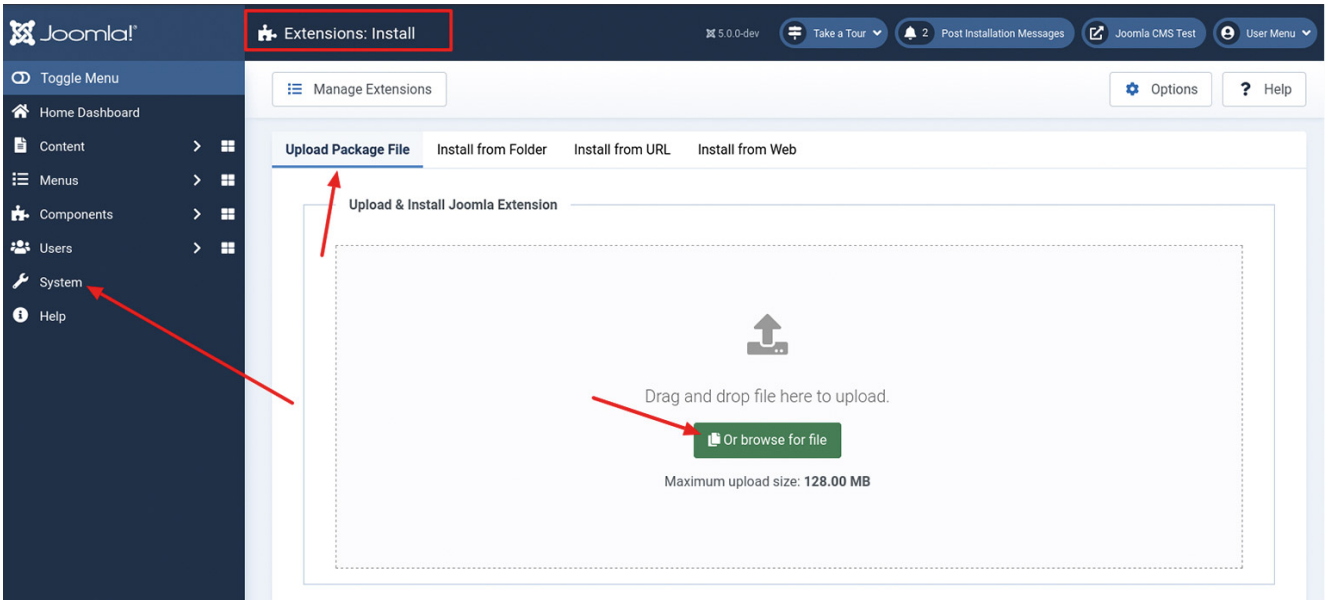

Abb. 11: Installation einer eigenen Joomla!-Erweiterung

Nach der Installation ist die Ansicht zum Verwalten der Beispielkomponente in der linken Seitenleiste des Joomla!-Backends (**Abb. 12**) integriert.

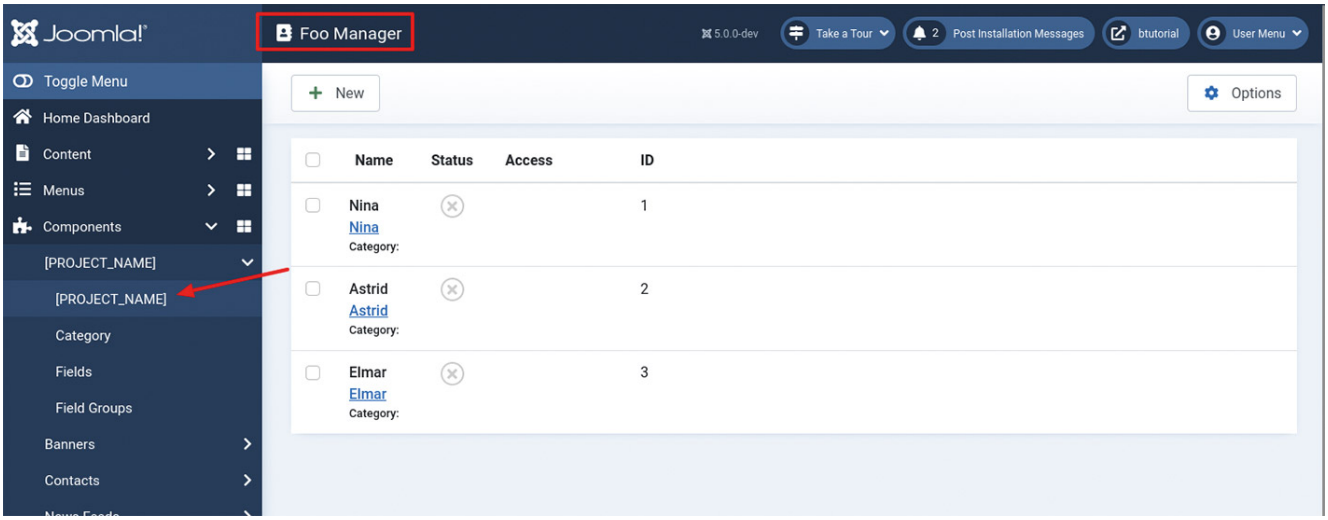

Abb 12: Ansicht der Beispielkomponente im Joomla!-Backend

Der erste eigene Test

Beim Testen des Joomla!-Backends beginnt jeder Test mit einem Log-in. Redundanten Code verhindern wir, indem wir die Funktion *beforeEach()* verwenden. Der Hook führt Code vor jedem Test aus. Daher der Name *beforeEach().*

Cypress bietet verschiedene Arten von Hooks [20], darunter *before* und *after* Hooks, die vor oder nach dem Test in einer Testgruppe ausgeführt werden, sowie *beforeEach* und *afterEach* Hooks, die vor oder nach jedem einzelnen Test in der Gruppe ablaufen. Hooks können global oder innerhalb eines bestimmten *described*-Blocks definiert werden. Der Code aus Listing 6 bewirkt, dass eine Anmeldung im Backend vor jedem Test innerhalb des *described*-Blocks *Test com_foos features* durchgeführt wird.

Listing 6

```
// tests/System/integration/administrator/components/com_foos/FoosList.cy.js

describe('Test com_foos features', () => {
  beforeEach(() => {
    cy.doAdministratorLogin()
  })
```

```
  ...
})
```

In der Datei */tests/System/support/index.js* sind Hooks implementiert, die global für jede Testdatei und jeden Test gelten. Die Komponente, die wir zum Testen installierten, beinhaltet drei Elemente. Zuerst testen wir, ob diese Elemente erfolgreich angelegt wurden (Listing 7).

Listing 7

// tests/System/integration/administrator/components/com_foos/FoosList.cy.js

```
describe('Test com_foos features', () => {
  beforeEach(() => {
    cy.doAdministratorLogin()
  })

  it('list view shows items', function () {
    cy.visit('administrator/index.php?...')

    cy.get('main').should('contain.text', 'Astrid')
    cy.get('main').should('contain.text', 'Nina')
    cy.get('main').should('contain.text', 'Elmar')

    cy.checkForPhpNoticesOrWarnings()
  })
})
```

Um sicherzustellen, dass ein Element angelegt wurde, suchen wir das DOM-Element *main* mit dem Cypress-Befehl *get* [21] und stellen via *should(‚contain.text', ‚…')* sicher, dass das Element sich in der Übersichtsliste befindet.

Neu ist auch die Funktion *checkForPhpNoticesOrWarnings().* Sie stellt sicher, dass keine Warnungen ausgegeben werden. Die Implementierung der Funktion befindet sich in der Datei */node_modules/joomla-cypress/src/support.js*.

Die gerade erstellte Testdatei *FooList.cy.js* sollte nach dem Speichern und Neuladen in der Liste der verfügbaren Tests in Cypress Browseroberfläche angeboten werden (**Abb. 13**). Klicken Sie auf den Namen FooList, um den Test auszuführen (**Abb. 14**).

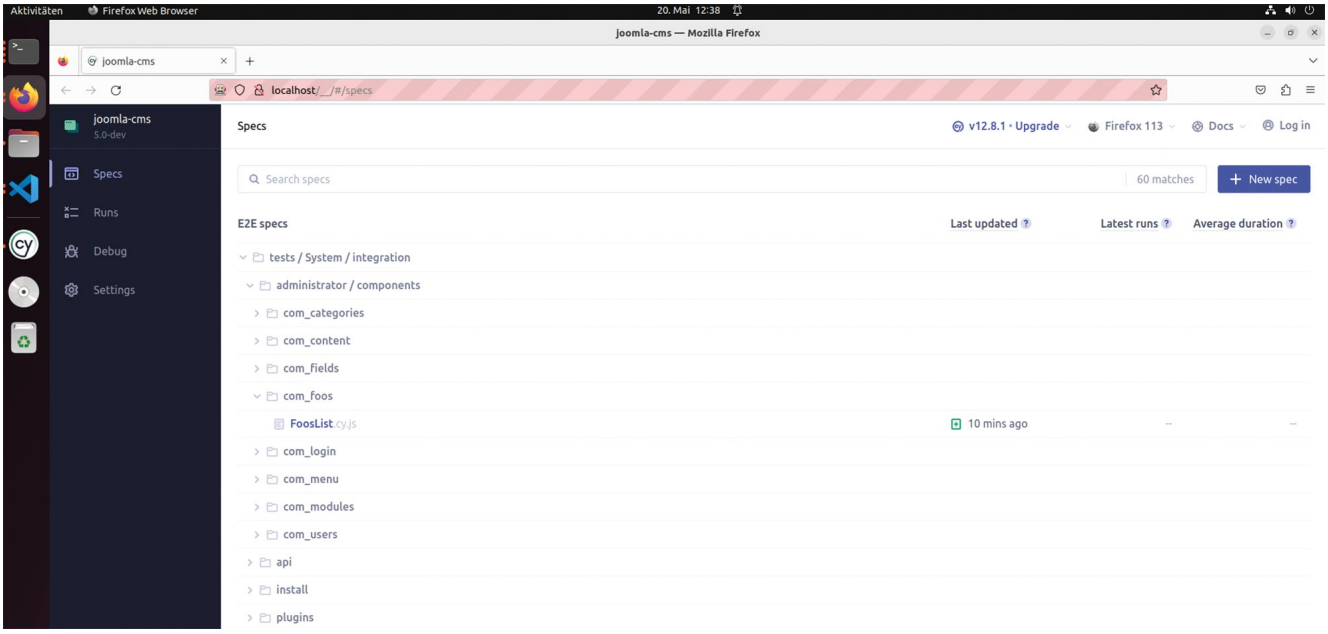

Abb. 13: Joomla!-Test für die eigene Erweiterung ausführen

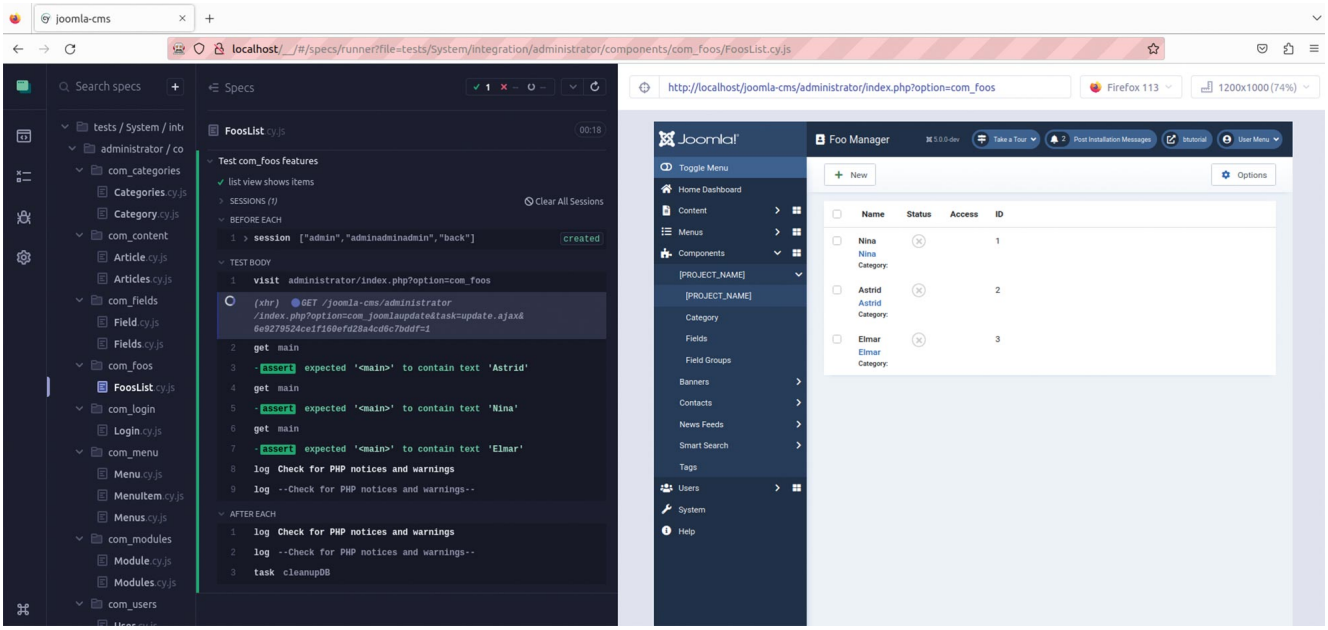

Abb. 14: Ansicht, nachdem der Test erfolgreich durchgelaufen ist

Fügen Sie jetzt die Zeile *cy.get(‚main‘).should(‚contain.text‘, ‚Sami‘)* in den Testcode ein, sodass der Lauf fehlschlägt. Nach dem Speichern der Testdatei erkennt Cypress die Änderung und führt alle Tests in

der Testdatei erneut aus. Wie erwartet schlägt der Test fehl (**Abb. 15**). Sie können jeden Testschritt in der linken Seitenleiste sehen. Für jeden Schritt gibt es einen Snapshot, sodass man das Markup jederzeit überprüfen kann.

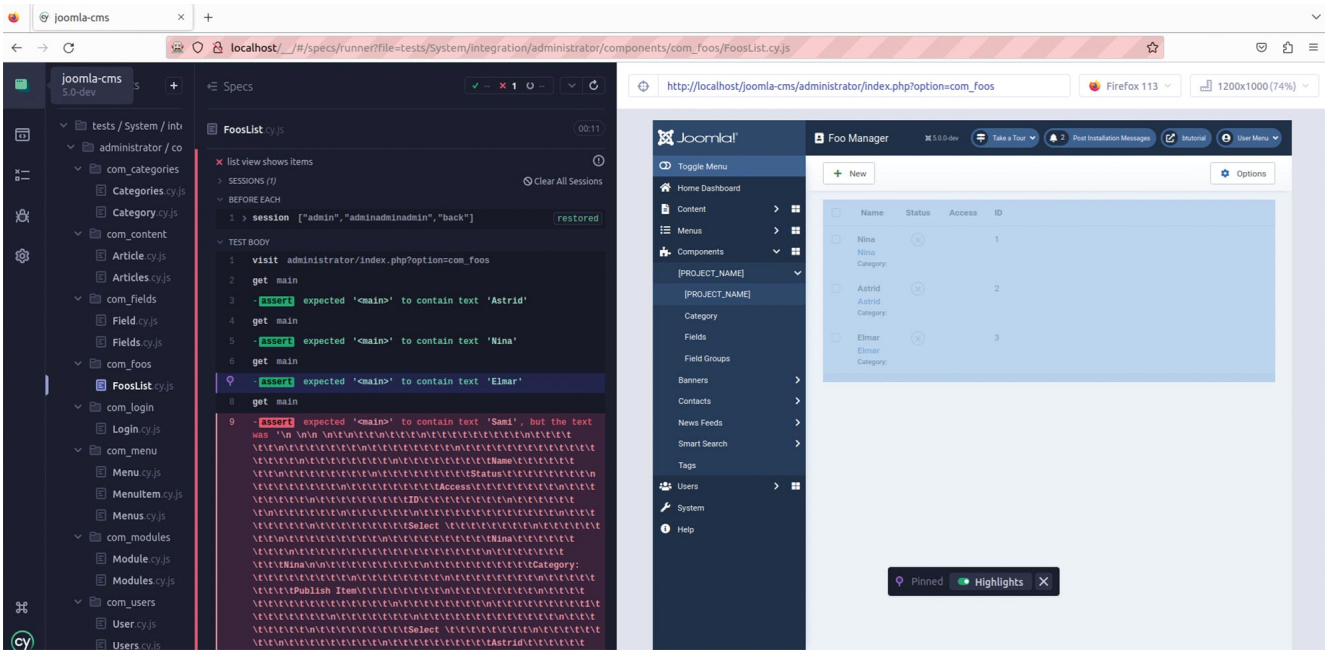

Abb. 15: Ansicht, nachdem der Test fehlgeschlagen ist

Testausführung organisieren

Im nächsten Schritt fügen wir einen Test zum Überprüfen des Empty-State-Layouts des Beispielcodes hinzu (Listing 8). Da wir nun zwei Tests in dieser Datei haben, wird Cypress bei jedem Speichern immer beide Tests ausführen. Via *.only()* ist es möglich, sich während der Entwicklung auf einen Test zu konzentrieren.

Listing 8

// tests/System/integration/administrator/components/com_foos/FoosList.cy.js

```
describe('Test com_foos features', () => {
  beforeEach(() => {
    cy.doAdministratorLogin()
  })
```

```
  it('list view shows items', function () {
    ...
  })

  it.only('emptystate layout', function () {
    ...
  })
})
```

Das Frontend der Beispielerweiterung testen wir mit der Datei */tests/System/integration/site/components/com_foos/FooItem.cy.js*. Bisher verwendeten wir CSS-Klassen, um ein HTML-Element zu finden. Das funktioniert, wird aber nicht empfohlen. Warum nicht? Wenn man CSS-Klassen verwendet, bindet man die Tests an etwas, das sich in der produktiven Umgebung ändern kann. Um die Tests stabiler zu implementieren, empfiehlt Cypress die Verwendung spezieller Testattribute. Ich werde das Attribut *data-test* für die Elemente verwenden. Zuerst füge ich das Attribut *data-test=“foo-main“* zum Produktionscode hinzu (Listing 9). Danach verwende ich das Attribut *[data-test=“foo-main“]*, um Elemente im Test zu finden (Listing 10).

Listing 9

```
// /components/com_foos/tmpl/foo/default.php

<?php
  \defined('_JEXEC') or die;
?>
<div data-test="foo-main">
  Hello Foos
</div>
```

Listing 10

```
//
tests/System/integration/site/components/com_foos/FooItem.cy.j
s
describe('Test com_foo frontend', () => {
  it('Show frondend via query in url', function () {
```

```
    cy.visit('index.php?...')
      cy.get('[data-test="foo-main"]').should('contain.text',
'Hello Foos')
    cy.checkForPhpNoticesOrWarnings()
  })
})
```

Jetzt testen wir die Erstellung eines Menüpunkts innerhalb von Joomla! für die Testkomponente. Das tun wir in der Datei */tests/System/integration/administrator/components/com_foos/MenuItem.cy.js*. Der Code ist komplex und weist eine Reihe von Besonderheiten auf.

Zunächst habe ich eine Konstante definiert, in der ich alle wichtigen Eigenschaften des Menüpunkts einstelle. Das hat den Vorteil, dass ich bei Änderungen der Menüpunktdaten nur an einer Stelle etwas anpassen muss:

```
const testMenuItem = {
  'title': 'Test MenuItem',
  'menuitemtype_title': 'COM_FOOS',
  'menuitemtype_entry': 'COM_FOOS_FOO_VIEW_DEFAULT_TITLE'
}
```

Listing 11 zeigt den relevanten Code der Datei *MenuItem.cy.js.*

Listing 11

*//
tests/System/integration/administrator/components/com_foos/MenuItem.cy.js*

```
describe('Test menu item', () => {
  it('creates a new menu item', function () {
    cy.intercept('index.php?...').as('item_list')
    cy.clickToolbarButton('Save & Close')
    cy.wait('@item_list')
                                          cy.get('#system-
message').contains('...saved').should('exist')
    ...
    cy.visit('administrator/index.php?...')
    cy.setFilter('published', 'Trashed')
```

```
    cy.searchForItem(testMenuItem.title)
    ...
    cy.on("window:confirm", (s) => {
      return true;
    });
    ...
                                          cy.get('#system-
message').contains('...deleted').should('exist')
  })
})
```

Zum einen zeigt dieses Beispiel, wie man einen Test so aufbaut, dass am Ende wieder der Ausgangszustand hergestellt ist. Auf diese Art und Weise können die Tests unbegrenzt oft wiederholt werden. Ohne das Wiederherstellen des Ausgangszustands würde der zweite Testlauf fehlschlagen, weil Joomla! das Speichern von zwei Elementen mit dem gleichen Namen ablehnt. Jeder Test sollte

- wiederholbar sein;
- einfach gehalten sein. Konkret bedeutet das, dass man eine begrenzte Problemstellung testen sollte und der Code hierzu sollte nicht zu umfangreich sein;
- unabhängig von anderen Tests sein.

Der Test zeigt ebenfalls, wie man eine mit *cy.intercept()* [22] definierte Route als Alias verwendet und dann via *cy.wait()* [23] auf diese wartet. Beim Schreiben von Tests ist man versucht, im Befehl *cy.wait* zufällige Zeitwerte wie *cy.wait(2000);* zu verwenden. Das Problem bei diesem Ansatz ist, dass ein reibungsloser Testdurchlauf nicht garantiert ist. Warum? Weil das zugrunde liegende System von Umständen abhängt, die sich kaum vorhersagen lassen. Daher ist es immer besser, genau zu definieren, worauf man wartet.

Der Code zeigt weiterhin, wie man auf einen Alert wartet und diesen via *window:confirm* bestätigt. Nicht zuletzt enthält der Testcode viele Funktionen, die von Entwicklern wiederverwendet

werden können. Beispielsweise *cy.setFilter(‚published‘, ‚Trashed‘)* oder *cy.clickToolbarButton(‚Save & Close‘).*

Asynchroner und synchroner Code

Cypress-Befehle sind asynchron, das heißt, sie geben keinen Wert zurück, sondern „erzeugen“ ihn. Cypress ordnet die Befehle seriell in einer Warteschlange. Wenn man in Tests asynchronen und synchronen Code mischt, erhält man unter Umständen unerwartete Ergebnisse.

Wenn Sie den Code aus Listing 12 aufrufen, wird wider Erwarten ein Fehler auftreten, weil *mainText === ‚Initial‘* am Ende noch gültig ist. Warum ist das so? Cypress führt erst den synchronen Code aus, der am Anfang und am Ende steht. Danach ruft es den asynchronen Teil innerhalb von *then()* auf. Das heißt, die Variable *mainText* wird initialisiert und gleich danach geprüft, ob sie sich geändert hat – was natürlich nicht so ist.

Listing 12

```
let mainText = 'Initial';
cy.visit('administrator/index.php?...')
cy.get("main").then(
  ($main) => (mainText = $main.text())
);

if (mainText === 'Initial') {
   throw new Error('Der Text hat sich nicht geändert. Er lautet: ${mainText}');
}
```

Anschaulich wird das Abarbeiten der Warteschlange, wenn man den folgenden Code in der Konsole des Browsers beobachtet. Der Text *Cypress Test.* erscheint lange bevor der Inhalt des Elements *main* ausgegeben wird, obwohl die Codezeilen in einer anderen Reihenfolge stehen.

```
cy.get('main').then(function(e){
```

```
  console.log(e.text())
})
console.log('Cypress Test.')
```

Spy und Stub

Ein *Stub* bietet die Möglichkeit, das Verhalten einer Funktion zu simulieren. Anstatt die eigentliche Funktion aufzurufen, ersetzt der Stub diese und gibt einen vordefinierten Wert zurück. Ein *Spy* ändert das Verhalten einer Funktion nicht. Vielmehr erfasst es einige Informationen darüber, wie die Funktion aufgerufen wird. Es prüft beispielsweise, wie oft oder mit welchen Parametern ein Aufruf erfolgte.

Listing 13 zeigt *Spy* und *Stub* in Aktion. Via *const stub = cy.stub()* erstellen wir das *stub*-Element und bestimmen im nächsten Schritt, dass beim ersten Aufruf *false* und beim zweiten *true* als Antwort zurückgegeben wird. Mittels *cy.on(‚window:confirm‘, stub)* erreichen wir, dass das Stub anstelle von *‚window:confirm‘* eingesetzt wird. Anschließend erstellen wir via *cy.spy(win, ‚confirm‘).as(‚winConfirmSpy‘)* das *Spy*-Element, das den Aufruf von *‚window:confirm‘* beobachtet. Nun simulieren wir im Testcode eine Situation, in der beim ersten Aufruf das Löschen einer Kategorie abgelehnt und beim zweiten Aufruf bestätigt wird. Dabei sorgt das *Stub* dafür, dass beim ersten Aufruf *false* und danach *true* per *‚window:confirm‘* zurückgegeben wird. *cy.get(‚@winConfirmSpy‘).should(‚be.called…‘)* hilft sicherzustellen, dass die Funktion tatsächlich in der erwarteten Häufigkeit aufgerufen wurde.

Listing 13

```
// tests/System/integration/administrator/components/com_foos/FoosList.cy.js
...
const stub = cy.stub()
```

```
stub.onFirstCall().returns(false)
stub.onSecondCall().returns(true)

cy.on('window:confirm', stub)

cy.window().then(win => {
  cy.spy(win, 'confirm').as('winConfirmSpy')
})

cy.intercept('index.php?...').as('cat_delete')
cy.clickToolbarButton('empty trash');

cy.get('@winConfirmSpy').should('be.calledOnce')
cy.get('main').should('contain.text', testFoo.category)

cy.clickToolbarButton('empty trash');
cy.wait('@cat_delete')

cy.get('@winConfirmSpy').should('be.calledTwice')

cy.get('#system-message-container').contains('Category
deleted.').should('exist')
...
```

Kurzgefasst

In diesem Beitrag haben Sie theoretische Grundlagen und praktische Besonderheiten von E2E-Tests kennengelernt. Das geschah anhand von eingängigen Beispielen. Ich hoffe, dass Sie den Rundgang durch Cypress und Joomla! genossen und Wissen und Anregungen für sich mitgenommen haben.

Seit 2017 programmiert Astrid Günther individuelle Websites und schreibt Bücher für Menschen, denen ihr Auftritt im Web wichtig ist. Am liebsten mit Joomla! und sehr gerne in Kombination mit geografischen Daten. Dabei schaut sie immer gerne über den Tellerrand hin zu anderen Open-Source-Projekten.

Links & Literatur

[1] https://www.joomla.de

[2] https://www.codeberg.org/astrid/entwickler_magazin_beispielcode_cypress_joomla (bit.ly/42qy2Aw)

[3] https://www.wikipedia.org/wiki/Projektmanagement#Stakeholdererwartungen

[4] https://www.martinfowler.com/bliki/TechnicalDebtQuadrant.html

[5] https://www.link.springer.com/chapter/10.1007/978-3-322-92018-8_1

[6] https://www.oikosofyseries.com/no-estimates-book-order

[7] https://www.learn.cypress.io/advanced-cypress-concepts/running-cypress-in-ci

[8] https://www.magazine.joomla.org/all-issues/june-2016/how-to-make-joomla-cms-tests-better-with-gherkin-and-codeception (bit.ly/3IYrouO)

[9] https://www.wikipedia.org/wiki/Cucumber_(Software)

[10] https://www.github.com/badeball/cypress-cucumber-preprocessor

[11] https://www.developer.joomla.org/roadmap#5x

[12] https://www.github.com/joomla/joomla-cms/tree/5.0-dev

[13] https://www.docs.joomla.org/Setting_up_your_workstation_for_Joomla_development/de (bit.ly/3oMWBdg)

[14] https://www.docs.cypress.io/guides/getting-started/installing-cypress (bit.ly/45Qt4js)

[15] https://www.wikipedia.org/wiki/Headless_(Informatik) (bit.ly/43mP8kr)

[16] https://www.docs.npmjs.com/cli/v9/commands/npx

[17] https://www.docs.cypress.io/guides/guides/command-line

[18] https://www.github.com/joomla-projects/joomla-cypress/tree/develop (bit.ly/3Cf8Hz6)

[19] https://www.codeberg.org/astrid/entwickler_magazin_beispielcode_cypress_joomla/releases (bit.ly/3WReofY)

[20] https://www.docs.cypress.io/guides/core-concepts/writing-and-organizing-tests#Hooks (bit.ly/43HJetD)

[21] https://www.docs.cypress.io/api/commands/get

[22] https://www.docs.cypress.io/api/commands/intercept

[23] https://www.docs.cypress.io/api/commands/wait