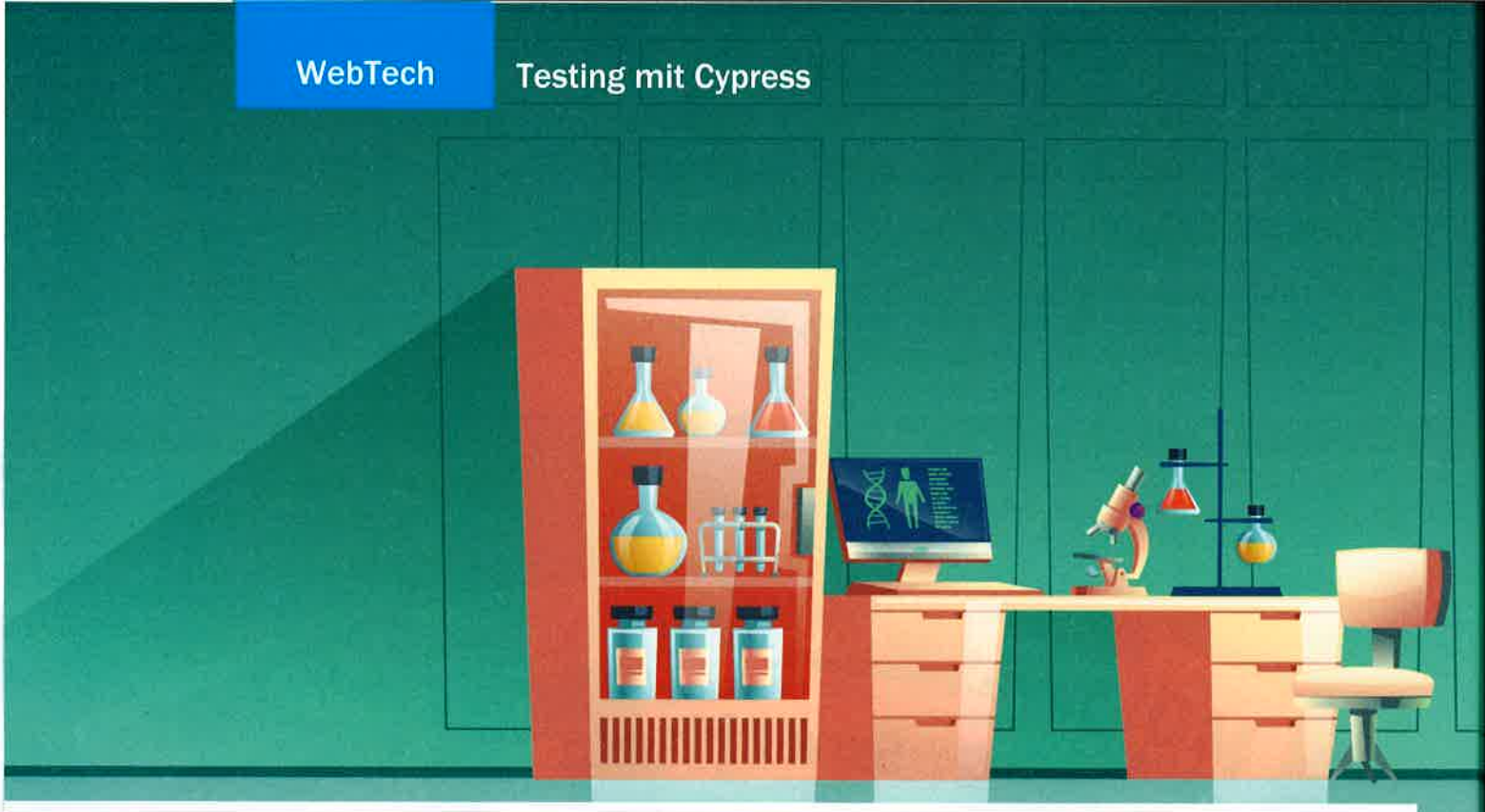


Das Testing Framework für React-, Angular- und Vue-Applikationen

End-to-End-Tests mit Cypress

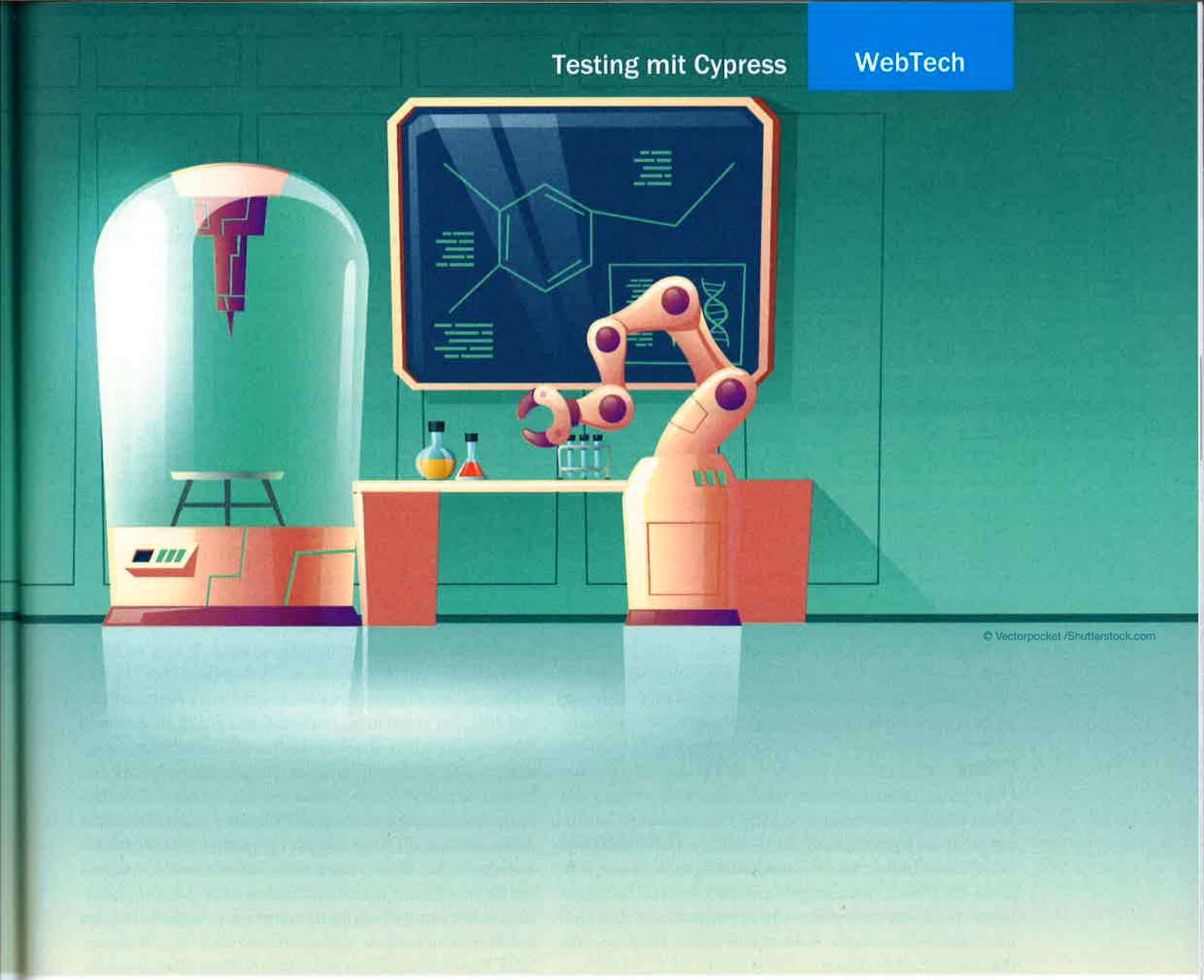
Bei Cypress handelt es sich um ein umfassendes Tool zur Implementierung von E2E-Tests für Webapplikationen, das unabhängig von der Architektur und den verwendeten Technologien zum Einsatz kommen kann. Es basiert auf einer ähnlichen Infrastruktur, wie sie auch bei Selenium zu finden ist. Mit diesem Werkzeug lassen sich sowohl React- als auch Angular- und Vue-Applikationen testen.

von Sebastian Springer

Sie kennen bestimmt das Modell der Testpyramide. Sie dient als Visualisierung der verschiedenen Testarten für eine Applikation. Auf der untersten Ebene liegen die Unit-Tests. Sie laufen schnell ab, verfügen über eine möglichst hohe Testabdeckung und sind in der Regel einfach zu schreiben. Auf der obersten Ebene der Pyramide stehen die UI-Tests. Sie werden häufig auch als End-to-End- oder kurz E2E-Tests bezeichnet. Die Hauptmerkmale dieser Tests sind, dass sie verglichen mit Unit-Tests eine sehr lange Laufzeit haben, aufwendig umzusetzen und deshalb verhältnismäßig teuer sind. In diesem Artikel lernen Sie mit Cypress ein Werkzeug für die Implementierung von E2E-Tests kennen, das Sie in einer Webapplikation unabhängig von der Architektur und den verwendeten Technologien einsetzen können.

Was sind E2E-Tests und wofür werden sie verwendet?

Der Begriff End-to-End bezieht sich auf die Tatsache, dass sich E2E-Tests, im Gegensatz zu Unit-Tests, nur auf spezielle Units of Code beziehen und diese unabhängig von ihrer Umgebung überprüfen. Das bedeutet im Umkehrschluss, dass solche Tests eine Applikation wirklich von einem Ende, also der grafischen Oberfläche im Browser, bis zum anderen, dem Server beziehungswei-



se noch einen Schritt weiter bis zur Datenbank, testen. Doch keine Sorge, um solche Tests zu schreiben, müssen Sie kein datenbankaffiner UI-Experte sein. Ganz im Gegenteil: Für den Einsatz einiger Test-Frameworks müssen Sie noch nicht einmal programmieren können. Mit der Selenium IDE können Sie beispielsweise Ihre E2E-Tests aufzeichnen lassen, indem Sie sich durch die Workflows Ihrer Applikation klicken. Der Nachteil dieser Herangehensweise ist, dass die Tests sehr fragil sind. Gerade das Warten auf bestimmte Elemente, bevor Sie mit ihnen interagieren, stellt ein großes Problem dar. Auch sind solche Tests nur schwer anpassbar und werden deshalb bei umfangreicheren Änderungen an der Applikation einfach gelöscht und neu aufgezeichnet. Aus diesem Grund gehen mittlerweile wieder viele Entwicklungsteams dazu über, ihre Tests selbst zu formulieren. Dabei greifen sie auf moderne Test-Frameworks, bestimmte Architekturformen und Best Practices zurück, die die Tests robuster, wartbarer und verständlicher machen. Im Idealfall können Sie einen E2E-Test wie einen ganz normalen Text lesen und so verstehen, wie der getestete Workflow aussieht.

Da E2E-Tests eine lange Laufzeit haben und sie, wenn sie während des Build-Prozesses einer Applikation ausgeführt werden, diesen spürbar verzögern können, weist eine Applikation meist wesentlich weniger E2E-Tests als Unit-Tests auf. Um herauszufinden, welche Teile Ihrer Applikation Sie testen sollten, identifizieren Sie die wichtigsten Workflows Ihrer Applikation. Stellen Sie sich und Ihren Kollegen die Fragen: Welche Features sind für den Erfolg der Applikation entscheidend? Womit verdienen wir unser Geld? Was muss unbedingt funktionieren, damit ein Benutzer die Applikation nutzen kann? Die Antworten auf diese Fragen geben an, was Sie testen sollten. E2E-Tests kommen auch für Routinen in Ihrer Applikation zum Einsatz, die sich mit anderen Tests nicht oder nur mit großem Aufwand überprüfen

Abb. 1: Cypress GUI

lassen. E2E-Tests sind außerdem ein Instrument zur Verifikation Ihrer Unit-Tests. Schlägt ein E2E-Test fehl und die Unit-Tests laufen gleichzeitig fehlerfrei durch, ist das ein Indiz dafür, dass ein bestimmter Aspekt noch nicht ausreichend getestet wird. In einem solchen Fall sollten Sie nicht nur dafür sorgen, dass der E2E-Test wieder erfolgreich läuft, sondern auch die Unit-Tests für den entsprechenden Bereich der Applikation auf den Prüfstand stellen. Auf diese Weise erzeugen Sie Sicherheit für die Entwickler, da es zusätzlich zu den Unit-Tests noch eine weitere Absicherung der Applikation gibt.

Cypress

Über viele Jahre hat sich Selenium als Werkzeug der Wahl für die Umsetzung von E2E-Tests etabliert. Selenium stellt die Infrastruktur für E2E-Tests zur Verfügung. Die Tests können in verschiedenen Sprachen wie C#, Java, PHP oder JavaScript verfasst werden. Selenium führt die Tests auf echten Browserinstanzen aus und nutzt dafür bestimmte Schnittstellen und Bindings, die den Zugriff abstrahieren.

Im Vergleich zu Selenium ist Cypress noch relativ jung. Der erste Commit in das Repository erfolgte im Juni 2014. Seit dieser Zeit hat sich viel getan und Cypress wurde um zahlreiche Features ergänzt, sodass es sich mittlerweile um eine vollwertige Lösung für E2E-Tests handelt, die Sie guten Gewissens für Ihre Applikation einsetzen können. Eine Besonderheit von Cypress ist, dass Tests und Applikation gemeinsam und nicht isoliert voneinander ausgeführt werden. Die grundlegende Infrastruktur ist ähnlich wie bei Selenium, da es auch bei Cypress eine Server- und eine Clientkomponente gibt, die kontinuierlich miteinander kommunizieren. Alle beteiligten Komponenten sind jedoch im Gegensatz zu Selenium in JavaScript geschrieben. Die Basis für den Serverprozess bildet ein Node.js-Server. Die Entscheidung für JavaScript als einzige unterstützte Sprache senkt zwar die Hürde für Frontend-Entwickler, erhöht sie jedoch für alle, die einen anderen Sprachhintergrund haben.

Cypress versucht, Ihnen eine Komplettlösung für Ihre E2E-Tests zu bieten, indem es alle Features mitbringt, die Sie zur Formulierung von Tests benötigen. Als zugrunde liegendes Test-Framework kommt Mocha in Kombination mit Chai zum Einsatz. Beide Bibliotheken haben sich vor allem im Node.js-Umfeld mittlerweile seit Jahren bewährt. Da es sich bei Cypress um eine eigenständige Open-Source-Implementierung handelt, sind Sie auch bei den Technologien in Ihrem Zielsystem nicht eingeschränkt. Cypress funktioniert sowohl mit klassischen Multi-Page- als auch modernen Single-Page-Applikationen. Dabei spielt es hier keine Rolle, mit welchem Framework oder welcher Bibliothek die Applikation umgesetzt wurde. Mit Cypress lassen sich sowohl React- als auch Angular- und Vue-Applikationen testen.

Installation und erste Schritte

Die Installation von Cypress erfolgt lokal in Ihrer Applikation mit dem JavaScript-Paketmanager Ihrer Wahl. Nutzen Sie beispielsweise Yarn, installieren Sie Cypress mit dem Kommando *yarn add -D cypress*. Bevor Sie dieses Kommando ausführen, sollten Sie sicherstellen, dass Ihre Applikation über eine gültige *package.json*-Datei verfügt. Eine solche können Sie sich mit dem Befehl *yarn init -y* erzeugen. Die Option *-D* bei der Installation von Cypress sorgt dafür, dass das Paket als *devDependency* in Ihre *package.json* eingetragen wird. Da es sich bei Cypress um ein Test-Framework handelt, das Sie nur während der Entwicklung und nicht zum Betrieb Ihrer Applikation verwenden, sollten Sie es nicht als normale Abhängigkeit installieren. Der Installationsbefehl sorgt dafür, dass alle erforderlichen Abhängigkeiten für den Betrieb von Cypress heruntergeladen werden. Eine Initialisierung erfolgt zu diesem Zeitpunkt noch nicht. Mit dem Kommando *yarn cypress open* starten Sie Cypress. Yarn sucht bei diesem Kommando nach einer Skriptdatei mit dem Namen *cypress* im lokalen *node_modules/.bin*-Verzeichnis und übergibt dieser bei der Ausführung das Subkommando *open*. *cypress open* öffnet das Desktop-GUI. Hierbei handelt es sich um eine React-Applikation, die mit Electron gerendert wird. Mit diesem Werkzeug erhalten Sie eine Übersicht über die Tests Ihrer Applikation und haben vielfältige Kontrollmöglichkeiten.

Initial erzeugt Cypress eine Reihe von Beispielen für Sie, die Sie zur Orientierung verwenden können. Bei der ersten Ausführung wird im Wurzelverzeichnis Ihrer Applikation ein neues Verzeichnis mit dem Namen *cypress* angelegt, das vier Unterverzeichnisse aufweist:

- *fixtures:* Das *fixtures*-Verzeichnis enthält Strukturen, die Sie für Ihre Tests laden können. Sie kommen zum Einsatz, um die Umgebung für einen Test vorzubereiten.
- *integration:* Das *integration*-Verzeichnis enthält wiederum ein Unterverzeichnis mit dem Namen *examples*. Hier finden Sie Beispieltests. Im *integration*-Verzeichnis sollten Sie Ihre eigenen Tests speichern, damit Sie von Cypress gefunden werden können.
- *plugins:* In diesem Verzeichnis können Sie Plug-ins speichern, mit denen Cypress erweitert werden kann.
- *support:* Die *index.js*-Datei im *support*-Verzeichnis wird vor allen Tests ausgeführt und kann verwendet werden, um beispielsweise eine globale Konfiguration zu laden.

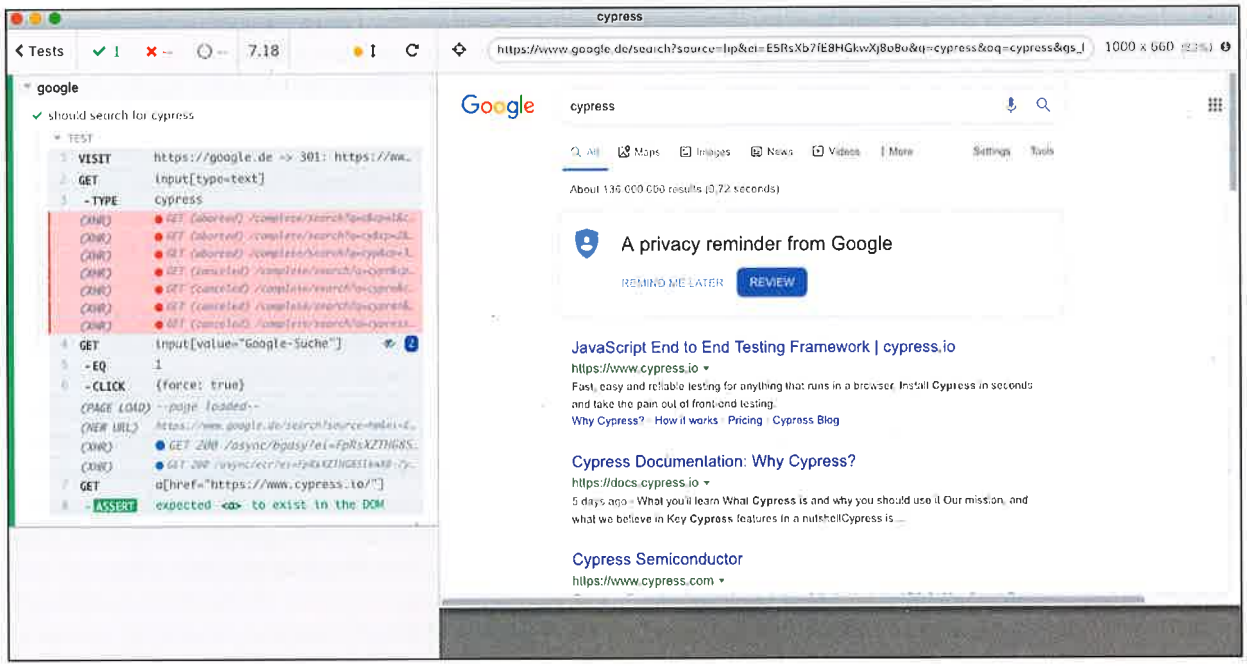


Abb. 2: Browserfenster mit Tests

Im Desktop-GUI (**Abb. 1**) sehen Sie drei Tabs, wobei der TESTS-Tab der wichtigste ist. Hier finden Sie die Liste der verfügbaren Tests. Haben Sie Ihre Tests in Unterverzeichnissen gruppiert, werden Ihnen diese als zusätzliche Strukturierungsebene angezeigt. Über das Suchfeld können Sie die Liste der Tests eingrenzen. Der Button RUN ALL SPECS sorgt dafür, dass alle Tests ausgeführt werden. Alternativ können Sie auch einen einzelnen Test durch einen Klick ausführen. Daraufhin öffnet sich ein zusätzliches Browserfenster und die Tests werden dort ausgeführt. Haben Sie einen oder mehrere Tests einmal gestartet, überprüft Cypress, ob der Quellcode des Tests modifiziert wurde und führt den Test bei Änderungen automatisch erneut aus. Möchten Sie diesen Prozess beenden, können Sie das über den STOP-Button in der rechten oberen Ecke des GUI tun. Ebenfalls rechts oben ist ein Drop-down-Menü mit einer Browserauswahl angesiedelt. Hier können Sie zwischen verschiedenen verfügbaren Browsern wählen, in denen die Tests ausgeführt werden sollen.

Das Browserfenster, in dem die Tests ausgeführt werden, ist wiederum in mehrere Sektionen unterteilt (**Abb. 2**). In der Kopfzeile sehen Sie die Anzahl der erfolgreichen beziehungsweise fehlgeschlagenen Tests und deren Dauer. Außerdem können Sie das automatische Scrollen deaktivieren und alle Tests erneut ausführen. Die danebenliegende Adressleiste gibt an, welche URL gerade vom Test überprüft wird. Der linke Teil des Browserfensters enthält die Liste der Tests. Auf der obersten Ebene werden die Test-Suites angezeigt, darunter liegen die einzelnen Testfälle sowie die einzelnen Optionen, die während des Tests durchgeführt wurden. Hovern Sie mit der Maus über einen bestimmten Eintrag in der Liste, wird im rechten Teil des Browserfensters der Stand der Applikation zu diesem Zeitpunkt angezeigt. So lassen sich das Verhalten der Applikation und potenzielle Fehler einfacher lokalisieren. Öffnen Sie die Entwicklerwerkzeuge des Browsers, können Sie den jeweiligen Stand auch inspizieren. Schlägt eine Operation eines Tests fehl, erhalten Sie auf der linken Sektion des Browserfensters zusätzliche Informationen.

Ein erster Test

Durch die Verwendung von Mocha und Chai folgt ein Cypress-Test den üblichen Regeln eines JavaScript-Unit-Tests. Auch die Syntax sollte Ihnen bekannt vorkommen, wenn Sie schon einmal einen Test mit einem der etablierten Test-Frameworks wie eben Mocha, Jasmine oder Jest geschrieben haben. Test-Suites werden mit der *describe*-Funktion erzeugt. Alternativ können Sie hier auch die *context*-Funktion verwenden. Beide sind äquivalent. Zur besseren Strukturierung Ihrer Tests können Sie auch mehrere *describe*-Blöcke ineinander schachteln. Die *describe*-Funktion akzeptiert als erstes Argument eine kurze Beschreibung der Test-Suite. Sie wird in der linken Spalte des Test Runners angezeigt. Als zweites Argument definieren Sie eine *Callback*-Funktion, die die Test-Suite enthält. Die Cypress-Tests selbst werden durch die *it*-Funktion eingeleitet. Auch sie erhält als erstes Argument eine Zeichenkette, die den Test beschreibt, und als zweites die eigentliche Testfunktion. Für die Vorbereitung vor den Tests können Sie mit den *beforeAll*- und *beforeEach*-Funktionen auf die Set-up-Routinen zurückgreifen. Das Aufräumen nach den Tests erfolgt mit den *afterAll*- und *afterEach*-Funktionen.

Damit Sie einen besseren Eindruck von den Möglichkeiten von Cypress bekommen, sehen wir uns einen konkreten Testfall an: Getestet werden soll die Eingabe des Suchbegriffs „cypress“ bei Google. Anschließend soll das Formular abgesendet und geprüft werden, ob ein Link zur Webseite von Cypress vorhanden ist. Listing 1 enthält den Quellcode dieses Tests.

Der Test beginnt mit der Erzeugung einer Test-Suite mit dem Label *google*. Darin finden Sie den Test mit der Beschreibung *should search for cypress*. Das globale *cy*-Objekt stammt von Cypress und bietet Ihnen alles, was Sie zur Formulierung eines E2E-Tests benötigen. Die *visit*-Methode weist Cypress beispielsweise an, den Browser zum angegebenen URL zu navigieren.

Mit der *get*-Methode lokalisieren Sie ein bestimmtes Element auf der aktuellen Seite. Die Besonderheit hierbei liegt darin, dass Cypress automatisch auf das gesuchte Element wartet. Es spielt also keine Rolle, ob das Element schon vorhanden ist oder erst durch einen Page Load oder JavaScript erzeugt werden muss. Cypress wartet bis der konfigurierte Time-out von standardmäßig vier Sekunden erreicht ist. Ist das Element dann immer noch nicht vorhanden, meldet Cypress einen Fehler und der Test wird abgebrochen.

Mit den Elementen, die Sie mit der *get*-Methode lokalisiert haben, können Sie verschiedenste Operationen durchführen. So ist es beispielsweise möglich, dass Sie das Element auf bestimmte Aspekte prüfen. Den einfachsten Fall sehen Sie im letzten Statement des Tests. Hier prüfen Sie, ob ein A-Element mit der Eigenschaft *href* und dem Wert *https://www.cypress.io/* existiert. Doch bevor Sie prüfen können, ob das Element existiert, müssen Sie zunächst das Eingabeformular bedienen. Zu diesem Zweck suchen Sie im ersten Schritt nach dem Eingabefeld und nutzen die *type*-Methode, um die Zeichenkette *cypress* einzugeben. Anschließend feuern Sie ein *click*-Event auf dem SUBMIT-Button. An dieser Stelle gibt es zwei Schwierigkeiten. Auf der Google-Webseite gibt es zwei Elemente mit dem *value*-Attribut und dem Wert *Google-Suche*. Cypress unterstützt jedoch standardmäßig nicht, dass ein *click*-Event auf mehreren Elementen ausgelöst wird. Dieses Verhalten können Sie zwar über Konfigurationsobjekte beeinflussen, was Sie in diesem Test jedoch eigentlich erreichen wollen, ist, dass auf den korrekten LOGIN-Button geklickt wird. Mit der *eq*-Methode können Sie ein Element aus einer Liste von Elementen auswählen. Im Test ist das das zweite gefundene Element. Mit der *click*-Methode lösen Sie dann das Event aus. Cypress interagiert standardmäßig nur mit Elementen, die auch für einen Benutzer erreichbar sind. Nutzen Sie die Google-Suche, präsentiert Ihnen Google eine Liste von Vorschlägen, die den SUBMIT-Button überlagert und diesen unbenutzbar macht. An dieser Stelle haben Sie zwei Möglichkeiten: Entweder Sie sorgen dafür, dass die Vorschlagsliste versteckt wird, indem Sie auf das gesuchte Element klicken, oder Sie nutzen, wie im Beispiel aus Listing 1, ein Konfigurationsobjekt, das Sie der *click*-Methode übergeben. Mit dem Schlüssel-Wert-Paar *force: true* erzwingen Sie den Klick, obwohl Cypress das Element nicht direkt verwenden kann.

Das Absenden des Formulars führt dazu, dass die Informationen zum Server gesendet werden und die Seite mit den Ergebnissen geladen wird. Da Cypress die Kontrolle sowohl über die Testausführung als auch die Applikation hat, stellt auch ein solcher Page Load kein Problem dar. Wie erwähnt wartet der anschließende *get*-Aufruf auf das Erscheinen des gesuchten Elements. Existiert es, wird der Test als erfolgreich betrachtet.

Listing 1: Cypress-Test

```
describe('google', () => {
  it('should search for cypress', () => {
    cy.visit('https://google.de');
    cy.get('input[type=text]').type('cypress');
    cy.get('input[value="Google-Suche"]')
      .eq(1)
      .click({ force: true });

    cy.get('a[href="https://www.cypress.io/"]').should('exist');
  });
});
```

Elemente finden

Das Auffinden von Elementen in einer Applikation ist wohl die häufigste Operation, die Sie in Ihren Cypress-Tests ausführen werden. Sie benötigen die Elementreferenzen sowohl für Interaktionen als auch zur Überprüfung. Gerade die Lokalisierung von Elementen ist ein Aspekt, der den Unterschied zwischen einem guten, stabilen und verlässlichen Test und einem unzuverlässigen ausmacht. Für die Lokalisierung eines Elements können Sie auf die verschiedenen Selektoren zurückgreifen, die Sie auch in der *querySelectorAll*-Methode nutzen können. Je eindeutiger ein solcher Selektor ist, desto stabiler wird Ihr Test. Nutzen Sie beispielsweise einen Pfad von Selektoren wie beispielsweise *div.parent > div.child > input*, besteht das Risiko, dass er sich mit der Zeit ändert. In diesem Fall reicht es schon aus, wenn ein zusätzlicher Container eingefügt wird. Greift der Selektor nicht mehr, schlägt der Test fehl und zwar nicht, weil die Funktionalität nicht mehr funktioniert, sondern weil sich die Struktur der Seite geändert hat. Eine Lösung für dieses Problem ist die Verwendung eindeutiger Attribute in den gesuchten Elementen. Andere Testbibliotheken wie beispielsweise die React Testing Library empfehlen das Attribut *data-testid*. Das Präfix *data-* stellt sicher, dass es keine Namenskonflikte mit den Standardattributen des Browsers gibt. Außerdem gelten keine Einschränkungen wie beispielsweise, dass es nur ein Element mit dieser Attribut-Werte-Kombination auf einer Seite geben darf, wie das beim *id*-Attribut der Fall ist.

Debugging in Cypress

Bei der Entwicklung von E2E-Tests kommt es immer wieder vor, dass Sie sich die Umgebung in der Applikation zu einem bestimmten Zeitpunkt während des Tests ansehen müssen. Das kann entweder der Fall bei der Fehlersuche sein oder wenn Sie an der Weiterentwicklung eines Tests arbeiten. Da Cypress im Browser läuft, können Sie auch auf die Entwicklerwerkzeuge des Browsers zugreifen und dort den DOM-Inspektor und

den JavaScript-Debugger verwenden. Hierbei sollten Sie jedoch beachten, dass die Kommandos von Cypress asynchron sind. Es reicht also nicht, aus, einen einfachen Breakpoint im Code zu setzen. Sie müssen warten, bis eine bestimmte Aktion abgeschlossen ist. Das können Sie auf zwei verschiedene Arten lösen. Entweder Sie nutzen die Tatsache aus, dass die Methoden, die Ihnen Cypress bereitstellt, *Promise*-Objekte zurückgeben, oder Sie nutzen die *debug*-Methode von Cypress. Soll der Test beispielsweise nach dem Klick auf einen Button angehalten werden, hängen Sie direkt an den Aufruf der *click*-Methode ein *.then*. In der *Callback*-Funktion, die Sie der *then*-Methode übergeben, können Sie dann sowohl mit dem *debugger*-Statement einen manuellen Breakpoint setzen als auch beliebige Logausgaben auf der Konsole tätigen. Bei der zweiten Variante nutzen Sie statt der *then*- die *debug*-Methode (Listing 2). Sie sorgt ebenfalls dafür, dass die Testausführung an dieser Stelle angehalten wird und Sie die Entwicklerwerkzeuge des Browsers verwenden können. Damit die Breakpoints funktionieren, müssen Sie die Entwicklertools zuvor geöffnet haben. Das erreichen Sie entweder über ein Tastaturkürzel oder mit einem Rechtsklick auf ein beliebiges Element und dann den Menüpunkt UNTERSUCHEN.

Responsive Layouts testen

Ein wichtiger Aspekt bei modernen Webapplikationen ist, dass sie sich an verschiedene Umgebungen anpassen können. So ist es üblich, dass auf einem Smartphone mit einem kleineren Display manche Elemente ausgeblendet und die bestehenden anders angeordnet werden als auf einem Desktoprechner. Mit Cypress haben Sie die Möglichkeit, auch dieses Verhalten zu testen, indem Sie Cypress mitteilen, welche Abmessungen der Browser überprüfen soll. Zum Einstellen des gewünschten Viewports nutzen Sie die *viewport*-Methode von Cypress (Listing 3). Dieser können Sie Breite und Höhe als Zahlen übergeben. Außerdem definiert Cypress eine Reihe von Geräten als sogenannte Presets, die Sie als Zeichenkette übergeben. Ein Aufruf von *cy.viewport(‚iphone-6‘)* setzt beispielsweise die Breite auf 375 px und die Höhe auf 667 px. Als zweites Argument können Sie außerdem die Ausrichtung des Geräts mit den Zeichenketten *portrait*, was der Standardwert ist, und *landscape* angeben.

Umgang mit Netzwerkverbindungen

Typischerweise überprüfen Sie bei einem E2E-Test nicht nur das Frontend, sondern die gesamte Applikation. Das bedeutet auch, dass Sie die Netzwerkanfragen vom Frontend zum Backend nicht abfangen, sondern durchlassen. Um die Umgebung korrekt testen zu können, benötigen Sie verlässliche und konsistente Testdaten. Die dürfen sich zwischen den einzelnen Testläufen auch nicht ändern, da Ihr Test ansonsten keine stabile Umgebung vorfindet. Diese Problemstellung lässt sich auf mehrere Arten lösen. So können Sie die Umgebung für Ihren Test jeweils zur Laufzeit vorbereiten. Das bedeutet, Sie legen die Datensätze manuell an. Testen Sie beispielsweise die Anmeldung Ihrer Applikation, benötigen Sie einen Benutzer. Diesen können Sie sich über die Registrierung anlegen. Das bedeutet allerdings, dass Sie im Zuge der Anmeldung auch die Registrierung testen. Ein solches Vorgehen ist jedoch nicht erstrebenswert, da ein so umfangreicher Test aufwendig zu erstellen und außerdem fehleranfällig ist. Eine bessere Lösung besteht darin, dass Sie serverseitig Testdatensätze vorbereiten, die Sie während des Tests verwenden können. Ein weiterer Lösungsansatz ist, dass Sie die Anfragen an den Server mit Hilfe von Cypress abfangen und sie innerhalb des Tests beantworten. Das Problem bei diesem Vorgehen ist, dass es sich hierbei um keinen klassischen E2E-Test handelt, da Sie den Server umgehen.

Mit einem Aufruf der *server*-Methode von Cypress aktivieren Sie zunächst die Möglichkeit, Anfragen abzufangen. Anschließend können Sie mit Hilfe der *route*-Methode angeben, welche Anfragen konkret abgefangen und wie sie beantwortet werden sollen. Die Antwort können Sie entweder direkt als JavaScript-Objekt angeben oder Sie nutzen das *fixture*-Feature von Cypress. Dabei legen Sie die gewünschte Datenstruktur im *fixtures*-Verzeichnis an und verweisen dann mit dem dritten Argument, das Sie der *route*-Methode übergeben, auf diese Fixtures. Fixtures referenzieren Sie, indem Sie zunächst die Zeichenkette *fixture:* gefolgt vom Namen der Datei angeben. Für eine bessere Verwaltung der Fixtures können Sie außerdem die *fixture*-Methode verwenden. Dieser übergeben Sie den Pfad zur Fixture-Datei relativ zum *fixtures*-Verzeichnis. Das Resultat ist der Base64-encodete Inhalt der Fixture-Datei.

Screenshots mit Cypress

Cypress bietet Ihnen die Möglichkeit, Screenshots und Videos vom Verlauf der Testausführung aufzuzeich-

Listing 2: Debugging

```
describe('google', () => {
  it('should&nbspsearch&nbspfor&nbspcypress', () => {
    cy.visit('https://google.de');
    cy.get('input[type=text]').type('cypress');

    cy.debug();

    ...
  });
});
```

Listing 3: Viewport

```
describe('google', () => {
  it('should&nbspsearch&nbspfor&nbspcypress', () => {
    cy.viewport('iphone-6');
    ...
  });
});
```

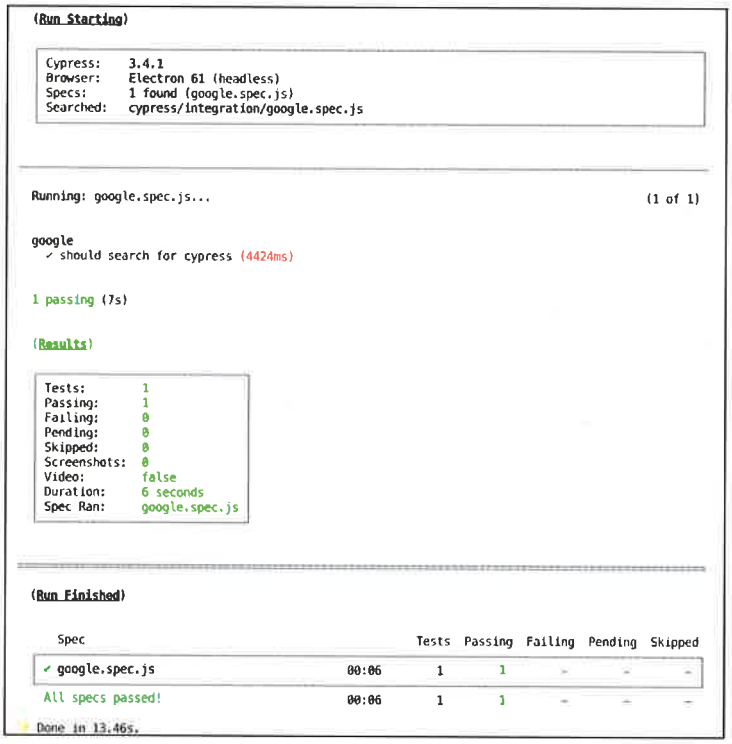


Abb. 3: Ausgabe der Headless-Ausführung von Cypress auf der Konsole

nen. Dieses Feature ist nützlich, wenn Sie einen Fehler suchen müssen oder wissen möchten, wie die Anzeige Ihrer Applikation zu einem bestimmten Zeitpunkt aussieht. Um einen Screenshot aufzuzeichnen, rufen Sie die *screenshot*-Methode innerhalb Ihres Tests auf. Cypress erzeugt daraufhin innerhalb des *cypress*-Verzeichnisses ein Unterverzeichnis mit dem Namen *screenshots*. Innerhalb dieses Verzeichnisses wird für jede Testdatei ein Verzeichnis erzeugt, das den gleichen Namen trägt, wie die Testdatei. Haben Sie also beispielsweise den Test in unserem initialen Beispiel in der Datei *google.spec.js* gespeichert und zeichnen in diesem Test nun einen Screenshot auf, wird im *screenshots*-Verzeichnis das Verzeichnis *google.spec.js* angelegt. In diesem Verzeichnis werden dann PNG-Dateien mit den Screenshots gespeichert. Die Namenskonvention dieser Dateien ist zunächst die Beschreibung aus dem *describe*-Block, gefolgt von der Beschreibung aus dem *it*-Block. Zeichnen Sie mehrere Screenshots in einem Test auf, werden mehrere Dateien mit dem gleichen Namen, gefolgt von einer Zahl, angelegt. Die Screenshots werden also nicht überschrieben. Im Headless-Modus, den Sie gleich noch näher kennenlernen werden, werden vor jedem Lauf die Screenshots gelöscht. Mit der Option *trashAssetsBeforeRuns* können Sie dieses Verhalten beeinflussen. Im Headless-Modus werden außerdem automatisch Screenshots beim Fehlschlag eines Tests aufgezeichnet.

Der Headless-Modus

Die Ausführung der E2E-Tests während der Entwicklung einer Applikation kann teilweise hilfreich sein. Aber gerade, wenn Sie eine größere Anzahl von Tests haben, die eine Laufzeit von mehr als ein paar Sekunden haben, ist es nicht mehr praktikabel, die Tests noch häufig auszuführen. Das regelmäßige Ausführen der E2E-Tests ist jedoch erforderlich, um eine Rückmeldung zum Zustand Ihrer Applikation und Ihrer Tests zu erhalten. Aus diesem Grund empfiehlt es sich, die Tests automatisiert auszuführen. In vielen Umgebungen steht Ihnen jedoch keine grafische Umgebung zur Verfügung. Cypress kann jedoch auch in einer solchen Umgebung Tests ausführen. Möglich wird das durch den von Cypress verwendeten Electron-Browser. Dieser ist in der Lage, die Tests headless, also ohne grafische Ausgabe auszuführen (**Abb. 3**). Einen solchen Headless-Testlauf starten Sie mit dem Kommando *yarn cypress run*. Die Tests werden in diesem Fall komplett ausgeführt und Cypress anschließend beendet. Schlägt ein Test fehl, nimmt Cypress wie schon erwähnt einen Screenshot auf. Außerdem werden die Testläufe standardmäßig als Video aufgezeichnet. Bei den Videos verfährt Cypress wie bei den Screenshots und leert das *videos*-Verzeichnis vor einem Testlauf. Falls Sie die Testaufzeichnung aus Speicherplatz- und Zeitgründen nicht wünschen, können Sie das in der *cypress.json*-Datei über den Schlüssel *video* mit dem Wert *false* deaktivieren. Alternativ dazu geben Sie beim *run*-Kommando die Option *--config video=false* an.

Standardmäßig schreibt Cypress die Ausgabe eines Testlaufs auf die Konsole. Zur besseren Integration in eine Continuous-Integration-Umgebung können Sie auch einen entsprechenden Reporter benutzen. Cypress beinhaltet die Teamcity und JUunit Reporter, sodass Sie diese lediglich aktivieren müssen.

Zusammenfassung

Cypress ist ein umfassendes Werkzeug zur Erzeugung von E2E-Tests für Webapplikationen. Im Kern besteht Cypress aus einem Node.js-Server und einer React-/Electron-Applikation. Die Macher von Cypress bieten außerdem einen teilweise kostenpflichtigen Dashboarddienst, der zusätzliche Features für Cypress bietet. Der Kern von Cypress, also der Test-Runner und das Desktop-GUI, ist jedoch als Open-Source-Projekt verfügbar und kann ohne das Dashboard verwendet werden.

Cypress integriert Mocha und Chai zur Formulierung der Tests und folgt damit der in JavaScript üblichen Syntax. Neben den Basisfunktionen zur Strukturierung der Tests und zahlreicher Funktionen zum Auffinden und Interagieren mit Elemente bietet Cypress noch viele Komfortfunktionen, wie beispielsweise Spying, Stubbing und Mocking von Funktionen und Teilen der Laufzeitumgebung. Damit können Sie die Stabilität der Tests verbessern und die Laufzeit verkürzen.

Cypress lässt sich sowohl auf einem Entwicklungsrechner als auch auf einem Server betreiben und ist damit ein hilfreiches Werkzeug für die Absicherung Ihrer Applikation.

Sebastian Springer ist JavaScript-Entwickler bei MaibornWolff in München und beschäftigt sich vor allem mit der Architektur von client- und serverseitigem JavaScript. Er ist Berater und Dozent für JavaScript und vermittelt sein Wissen regelmäßig auf nationalen und internationalen Konferenzen.