#JAVAPRO #Framework #Desktop #Testing

Web- und Desktop-Anwendung aus einer Code-Base

Viele sehen die Zukunft von Java im Web, gleichzeitig sollen aber bestehende Desktop-Anwendungen weiterhin stand-alone funktionieren. Die Lösung: Einen Application-Server transparent in eine Electron-Anwendung einbetten und damit das bestehende Nutzererlebnis erhalten. Beispielhaft wird in diesem Artikel eine Swing-Altanwendung in eine Web-Anwendung transformiert und deren Qualität durch hochwertigen UI-Test gesichert.

Autor:

Daniel Rieth steht am Anfang seiner Karriere als Software-Eentwickler und -Beraterberater. Nach den ersten Schritten in Informatik und Pädagogik an der LMU München vertieft er nun seine softwaretechnischen Kenntnisse durch ein duales Studium mit QFS als Praxispartner.

Dr. Pascal Bihler forschte in Deutschland, Frankreich und den USA zu Internet -of -Things, Aagiler Software-Eentwicklung und User -Experience -Design, lehrte als Softwaretechnik-Dozent für Softwaretechnik an der Uni- Bonn, arbeitete als freier Software-Bberater für UI-Design und entwickelte die ersten erfolgreichen kooperativen mobilen Spiele. Bei QFS arbeitet er nun an Tools zur Qualitätssicherung von Desktop- und Web-Anwendungen.

Quality First Software GmbH - www.qfs.de
https://twitter.com/qftest_qfs
https://www.xing.com/companies/qualityfirstsoftwaregmbh
https://www.linkedin.com/company/quality-first-software-gmbh-qfs
https://gitlab.com/qfs
[daniel.rieth,pascal.biehler]@qfs.de

Anfang 2014 kamen die Entwickler von GitHub auf eine wegweisende Idee: Für ihren hauseigenen Editor Atom[1] verheirateten sie das auf browserlose JavaScript-Ausführung ausgelegte Node.js Framework, dessen zentraler Bestandteil (die V8-Engine) aus Chromium herausgelöst worden war, wieder mit einer rahmenlosen Chrome-Rendering-Engine. Damit schufen sie die Möglichkeit, Anwendungen, die mit Web-Technologien entwickelt waren und üblicherweise auf eine Server-Client-Infrastruktur aufsetzen, als Desktop-Anwendungen auszuführen, ohne dass nach dem Download bzw. der Installation der Anwendung zur Programmausführung eine Netzwerkverbindung notwendig ist. Ende April ist das daraus hervorgegangene quelloffene Electron-Framework[2] bei Version 5 angekommen und das Entwickler-Team verspricht Updates im Dreimonatstakt, bei denen jeweils die eingebettete Node.js- und Chromium-Version aktualisiert wird. Und neben GitHub mit dem eingangs genannten Atom-Editor sind auch viele weitere Softwareschmieden auf den Zug aufgesprungen: So arbeitet das Framework unter anderem in den bekannten Tools Microsoft-Visual-Studio-Code und Skype.

Für Java-Entwickler, die mit Swing-Applikationen, insbesondere mit Java-Webstart in eine ungewisse Zukunft sehen, bietet sich durch Electron die Möglichkeit, populäre Web-Frameworks für ein Redesign der Benutzerschnittstelle zu verwenden und dennoch weiterhin komfortable und kundenfreundliche Stand-Alone-Desktop-Clients ausliefern zu können. In diesem Artikel möchten wir diesen Prozess am Beispiel der technischen Aktualisierung einer Swing-Anwendung durchspielen.

Mit dem Skalpell UI und Controller trennen

Das MVC- (Model-View-Controller) bzw. das MVVM-Pattern (Model-View-ViewModel), bei dem an die Stelle des zentralen Controllers ein ViewModel getreten ist, das über eine Datenbindung die Anbindung an das User-Interface realisiert, mag manchem Entwickler eingestaubt vorkommen. Für den ersten Schritt der technischen Anwendungsaktualisierung bietet es aber ein hervorragendes Werkzeug: In der Theorie sind Datenhaltung und Datenpräsentation über Schnittstellen klar voneinander getrennt und technisch austauschbar, die Realität sieht leider häufig anders aus. So gilt es also zunächst klar festzulegen welche Daten im Model verwaltet werden sollen und welche Elemente sich auf das verwendete UI-Framework beziehen – der Rest verbleibt in der Controller- bzw. ViewModel-Schicht. Am Ende dieses Schrittes, der häufig an eine Mischung von Archäologie und Chirurgie mit virtuellem Pinsel und Skalpell erinnert, steht eine Anwendung, die optisch identisch zum Ursprungsprodukt ist, intern aber die einfache Möglichkeit zum Austausch der Präsentationsebene bietet. Das erfolgreiche Anwendungs-Refactoring verifiziert man am einfachsten über eine möglichst breite Abdeckung aller Anwendungsfälle mit automatisierten UI-Tests: Tools wie QF-Test[3] erlauben es, schnell mit der bestehenden Anwendung Testfälle aufzuzeichnen. Diese können dann während des Refactoring-Prozesses immer wieder gegen die überarbeitete Anwendung abgespielt werden, so dass Abweichungen früh im Prozess erkannt und korrigiert werden können.

User-Interface mit Web-Technologien darstellen

Zur Erstellung der UI verwenden wir den GUI-Builder von RapidClipse[4]. Dieses Tool ermöglicht es innerhalb von kurzer Zeit, eine voll funktionstüchtige, auf Vaadin[5] basierende Benutzeroberfläche zu erstellen und bei Bedarf auch für mehrere unterschiedliche Endgeräte und Plattformen auszuliefern. Nach dem Anlegen eines neuen RapidClipse-Projektes ist bereits der Rahmen für unsere Applikation gegeben. Die bereitgestellte Project-Management-View bietet eine gute Sortierung für unsere Datenbanken, UI-Elemente, Ressourcen und Themes, die zum Stylen unserer Anwendung definiert werden können. Wir benutzen hier das vorgegebene Runo-Theme, sodass ein einfacher Aufruf von **`setTheme("runo")`** in der **`init`** Methode unsere gesamte Interaktion mit diesem Aspekt der Frontendprogrammierung darstellt. Alle Änderungen an der View, vom Platzieren von UI-Komponenten bis zur Einstellung ihrer Eigenschaften, können in der praktischen GUI-Builder-Ansicht durchgeführt werden. Der zugehörige Code wird automatisch generiert bzw. angepasst, kann aber bei Bedarf auch selbst geschrieben und eingebunden werden. Event-Handler zur Interaktion der Oberfläche mit unserer Business-Logic werden mit einem simplen Rechtsklick auf die gewünschte Komponente erzeugt.

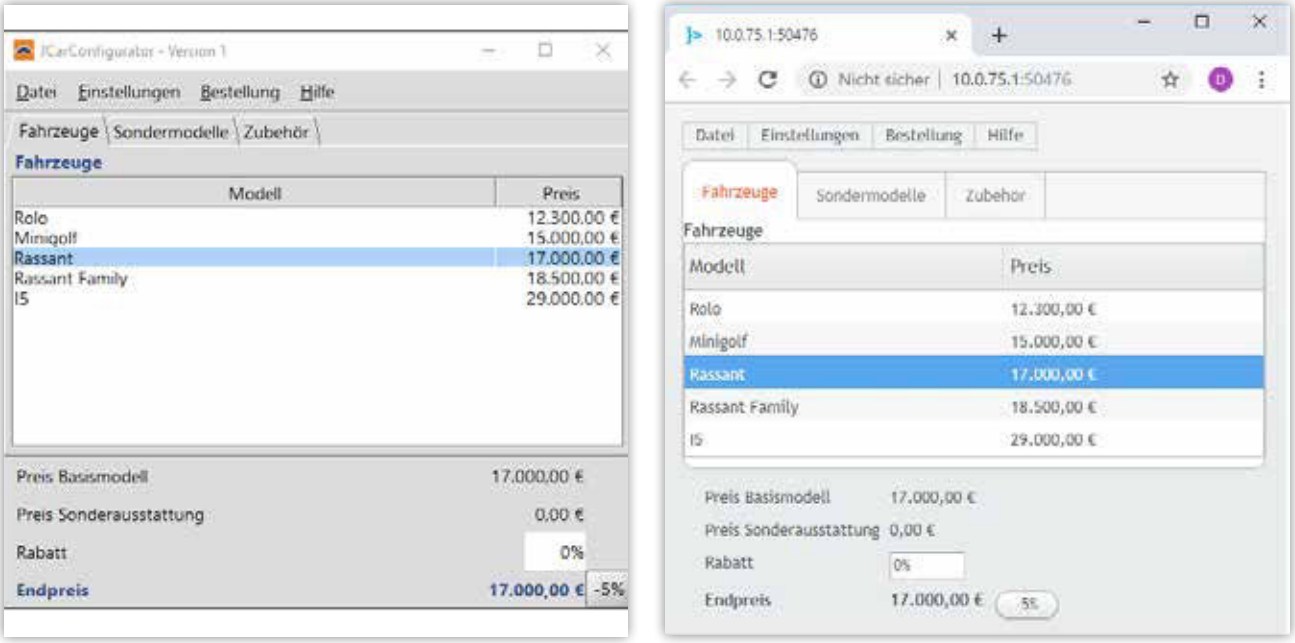
Einige Menüelemente in unserer ursprünglichen Swing-Anwendung öffnen neue Fenster, beispielsweise Dialoge, um Tabellendaten zu bearbeiten oder Informationen über die ausgewählten Elemente anzuzeigen. Um in Vaadin ein neues Fenster zu erstellen, bedienen wir uns des Navigators. Der bei Projekterzeugung von RapidClipse automatisch generierten **`DesktopUI.java`** wird ebenfalls über den GUI-Builder, ein Navigator hinzugefügt. Im Navigator können in dessen Properties mehrere Pfade mit zugehörigen Views registriert werden, welche dann im Programm mit der **`navigate`** Methode angesteuert werden können **(Listing 1)**. Die MainView hat logischerweise einen leeren String als Pfadnamen.

(Listing 1 - Wechsel der View per Klick auf leaveViewButton)

```
private void leaveViewButton_buttonClick(final Button.Click event){
   Navigation.to("pathname").navigate();
}
```

Ein weiteres praktisches Werkzeug von RapidClipse ist die Quick-Launch-Ansicht. Hier kann das gesamte UI, oder einzelne Fenster simultan zur Entwicklung im Browser angezeigt werden. Änderungen im GUI-Builder werden hier in Echtzeit reflektiert, wobei einige Änderungen, beispielsweise Namensänderungen der Elemente, hiervon natürlich ausgeschlossen sind.

Das Nachbilden der ursprünglichen Swing-Oberfläche erfolgt auf diese Art sehr unkompliziert. Dies liegt zum einen

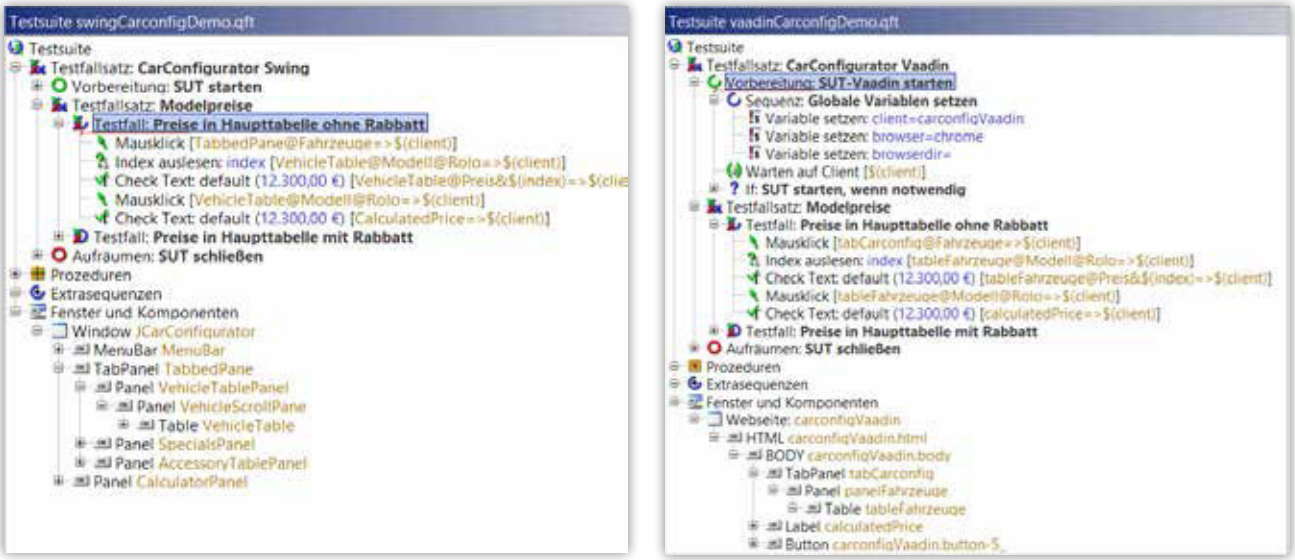


Die GUIs im Vergleich: links Swing, rechts Vaadin. (Abb. 1)

an dem einfachen GUI-Builder, zum anderen aber am verwendeten Vaadin-Framework, welches viele Aufgaben wie etwa die Client-Server-Kommunikation, transparent realisiert. Im Ergebnis steht die neu entwickelte Oberfläche in seiner Funktion und seinem Aussehen dem ursprünglichen Swing-Programm in nichts nach **(Abb. 1)**

Die Beibehaltung der UI-Struktur ermöglicht es uns nun auch, beim Testen der Oberfläche auf die bestehende Teststruktur aufzusetzen. Unterstützt das verwendete Tool gleichermaßen plattformübergreifende Tests für Java- und Web-Anwendungen, wie zum Beispiel das bereits erwähnte QF-Test, so genügt es häufig, die Komponentendefinition für die neue Engine anzupassen sowie die Start-Prozedur für das SUT (System-Under-Test – die zu testende Anwendung) neu zu generieren. Lag vorher für die Swing-Oberfläche bereits eine flächendeckende Teststruktur vor, kann diese so auf die neue Technologie angepasst werden und es können so schon im Entwicklungsprozess Fehler schnell erkannt werden. Aber auch, wenn die Testabdeckung bisher gering war, können durch den Record-Replay-Ansatz schnell weitere Tests generiert werden.

Um das generierte Vaadin-GUI mit QF-Test automatisiert zu testen, gibt es mehrere Optionen. Zum einen ist es möglich, die generierte WAR-Datei auf einem Application-Server zu deployen und sich über die URL zu verbinden. Allerdings kann es mühsam und zeitaufwändig sein, diese Datei häufig zu erzeugen, vor allem bei größeren Anwendungen. In unserem Fall lässt sich aber auch



Die dazugehörigen Tests: links Swing, rechts Vaadin. (Abb. 2)

der Preview-Modus in RapidClipse benutzen, der aus der Quick-Launch-Ansicht gestartet werden kann. Da der von RapidClipse vorinstallierte Servlet-Container allerdings die Preview jedes Mal auf unterschiedlichen Ports startet, muss die URL in der Startsequenz des SUT entsprechend angepasst werden. Hierbei hilft es, die Adresse bei allen Vorkommnissen in der Testsuite durch eine Variable zu ersetzen, insbesondere bei Verwendung des Schnellstart-Assistenten in der Sequenz „Browser-Fenster öffnen". Einfacher ist es, einen eigenen Server für die Previews festzulegen und dort den Port einzustellen. Hierfür muss im Menü Quick-Launch die Option Start Servlet ausgewählt und dort der entsprechende Port definiert werden. Für die Einbettung in eine Desktop-Anwendung ist es notwendig, die Web-Anwendung über einen lokalen, dedizierten Prozess zur Verfügung zu stellen. Für unser Beispiel bietet sich Jetty-Runner[6] an, der eine direkte Darstellung der Anwendung aus der WAR-Datei ohne zusätzliches Deployment erlaubt.

Web-Archive in Electron-App einbetten

Für den Einstieg in die Entwicklung von Electron-Anwendungen bietet das zugehörige Quick-Start-Projekt[7] eine gute Basis. Nachdem dieses geklont und über npm install eingerichtet wurde, können wir die zuvor entwickelte Vaadin-Anwendung in den Electron-Container einbinden. Dazu erstellen wir im Projektverzeichnis einen Ordner mit dem Namen lib und kopieren unsere WAR-Datei sowie die Datei **`jetty-runner.jar`**[8] dorthin. Um den Jetty-Runner zusammen mit der Anwendung starten und auch sauber wieder stoppen zu können, benötigen wir darüber hinaus noch die beiden npm-Module get-port und tree-kill, die wir über den Befehl npm install get-port tree-kill installieren können. Wenn wir das Electron-Programm über **`npm start`** starten, sieht man eine einfache Seite, die einige Informationen über die laufenden Prozesse anzeigt. Diese Seite wird aus der Datei **`index.html`** geladen und stellt die Standard-Benutzerschnittstelle für die Demoanwendung dar. Da die Informationen in dieser Datei statisch sind, können sie vergleichsweise schnell geladen werden. Mit ein paar optischen Anpassungen können wir sie so in unserem Projekt gut als Splash-Screen verwenden.

Jetzt, da das Fenster angezeigt wird, soll der Jetty-Runner im Hintergrund unsere Web-Application starten und die Anzeige, nachdem alles geladen ist, zur Anwendungs-UI wechseln. Dazu sind einige Anpassungen in der Datei **`main.js`** erforderlich. Diese Datei kontrolliert die Electron-Anwendung primär und wird dazu in einem Main-Prozess innerhalb einer Node.js-Umgebung ausgeführt. Von dort aus können Browser-Fenster geöffnet werden, welche ihrerseits JavaScript-Code ausführen können. im Beispiel wird die Datei **`renderer.js`** geladen, später dann der durch Vaadin erzeugte Code. Dieser Code wird im Kontext des Browser-Fensters im sogenannten Renderer-Prozess ausgeführt und kann mit dem Main-Prozess über eine Inter-Prozess-Kommunikation (IPC) weitgehend transparent Daten austauschen. Um dies zu vereinfachen, ergänzen wir in der **`main.js`** die Eigenschaften des Browser-Fensters (Listing 2). Da wir im Fenster nur lokalen Code laden wollen, steht der Zugriff von einem Prozess auf den anderen immer unter unserer Kontrolle. Ein Beispiel für die Interprozesskommunikation ist das Beenden der Anwendung aus dem Java-Code heraus mittels P**`age.getCurrent().getJavaScript().execute("require ('electron').remote.app.quit()")`**.

(Listing 2 - Vereinfachter IPC-Zugriff in der main.js)

```
// Create the browser window.
mainWindow = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
        enableRemoteModule: true,
        contextIsolation: false,
        nodeIntegration: true
    }
})
```

Den Java-Prozess zur Ausführung unserer Web-Application möchten wir direkt im Anschluss an den **`createWindow`** Aufruf starten. Dazu ergänzen wir den bestehenden **`ready`** Handler der Anwendung (Listing 3).

(Listing 3 - Erweiterter Ready-Handler)

```
app.on('ready', function() {
    createWindow()
    startJavaProcess('carconfig.war') // Name der WAR-Datei
})
```

In (Listing 4) sieht man, wie der eigentliche Jetty-Prozess gestartet wird: Zunächst wird mit dem getPort-Modul ein freier TCP-Port ermittelt. Standardmäßig startet Jetty immer auf Port 8080, der in unserem Fall dann aber schnell belegt ist. Im Anschluss daran werden der Jetty-Runner mit diesem Port und dem WAR-File als Parameter, als neuer Prozess gestartet und die Ausgaben des Prozesses abgefangen.

(Listing 4 - Start des Jetty-Prozesses)

```
async function startJavaProcess(warFileName, showOutput = false) {
    const getPort = require('get-port')
    const child_process = require('child_process')
    const javaPort = await getPort()

    let resourcesDir = app.getAppPath()
    let javaBin = 'java'

    const javaProcess = child_process.spawn(javaBin,
        ['-jar', `${resourcesDir}/lib/jetty-runner.jar`,
        '--host', '127.0.0.1',
        '--port', javaPort,
        `${resourcesDir}/lib/${warFileName}`],
        {
             stdio: ['ignore', 'pipe', 'pipe'],
             windowsHide: true,
```

```
            detached: true
        }
    )

    javaProcess.stdout.on('data', (data) => {
      if (showOutput) process.stdout.write(`jetty-out: ${data}`)
    })

    javaProcess.stderr.on('data', (data) => {
      if (showOutput) process.stdout.write(`jetty-err: ${data}`)
    })

    const pid = javaProcess.pid

    console.log(`Starting java server on port ${javaPort},
    process ${pid}`)
}
```

Nachdem der Prozess gestartet wurde, muss nun auf den Start der Web-Application gewartet und dann der Inhalt des Browser-Fensters entsprechend gewechselt werden. Dazu ist es möglich, regelmäßig den Ziel-Port abzufragen und nachzusehen, ob dort die Anwendung bereits zur Verfügung steht. Einfacher und robuster ist es aber, die Ausgaben des Jetty-Servers auszuwerten und nach der finalen Ausgabe die Umleitung durchzuführen. Dazu können wir den STDERR-Handler wie in (Listing 5) dargestellt erweitern.

(Listing 5 - Automatischer Redirect auf die Startseite)

```
javaProcess.stderr.on('data', (data) => {
    if (showOutput) process.stdout.write(`jetty-err: ${data}`)
    if (data.includes('INFO:oejs.Server:main: Started @')) {
        mainWindow.loadURL('http://127.0.0.1:' + javaPort)
    }
})
```

Alternativ zum Standard-Log des Jetty-Servers wäre es auch denkbar, nach dem Start der Web-Anwendung aus dieser heraus eine eindeutige Ausgabe zu erzeugen, auf die dann in der **`main.js`** reagiert wird. Abschließend soll der Java-Prozess beim Beenden der Electron-Anwendung auch zuverlässig mit allen Kind-Prozessen geschlossen werden. Dazu bietet sich auf Linux und Mac-OS-Systemen der Befehl **`process.kill(-pid)`** an. Durch den **detached** Parameter beim Start des Prozesses wurde eine neue Prozessgruppe erstellt, die nun durch die Angabe des Minuszeichens komplett beendet wird. Unter Windows muss man auf die Hilfe des Tree-kill-Moduls zurückgreifen. Der vollständige Quit-Handler, welchen wir noch am Ende der **`start JavaProcess`** Methode ergänzen, wird in (Listing 6) gezeigt.

(Listing 6 - Java-Prozess mit der Anwendung beenden)

```
app.on('quit', function(event) {
    if (pid) {
        console.log(`Stopping java server with process ${pid}`)
        if (process.platform == 'win32') {
            const treeKill = require('tree-kill')
            treeKill(pid, 'SIGTERM')
        } else {
            process.kill(-pid)
        }
    }
})
```

Electron-App für die Distribution vorbereiten

Bisher läuft die Electron-App nur innerhalb der Entwicklungsumgebung. Um eine Electron-App zu verpacken und für die Auslieferung vorzubereiten, stehen verschiedene Module bereit. Eines davon ist der electron-builder[9], welcher mit **`npm install --save-dev electron-builder`** dem Projekt hinzugefügt wird. Um diesen Auszuführen, fügt man in den Scripts-Bereich der **`package.json`** die Zeile **`dist`**, **`electron-builder`** hinzu und ruft dann **`npm run dist`** auf der Kommandozeile auf. Damit auch alles wie erwartet funktioniert, sind aber noch ein paar Anpassungen vorzunehmen: Zunächst muss dem Builder mitgeteilt werden, dass er die JAR- und WAR-Dateien im **`lib`** Verzeichnis als Resourcen aufzufassen hat. Dazu ergänzen wir die **`package.json`** wie in (Listing 7) dargestellt um einen **`build`** Abschnitt.

(Listing 7 - Build-Informationen in der package.json)

```
"build": {
    "appId": "our.examples.JavaElectron",
    "files": [
        "main.js",
        "index.html"
    ],
    "extraResources":[
        "lib"
    ]
}
```

Damit werden die Bibliotheken in das **`ressources`** Verzeichnis gelegt, welches während der Ausführung über **`process.resourcesPath`** zur Verfügung steht. Leider divergiert hier die Ausführung während der Entwicklungszeit von der fertig gebauten Version, so dass wir die **`startJavaProcess`** Methode wie in (Listing 8) modifizieren müssen – existiert das **`lib`** Verzeichnis im Ressourcen-Ordner, so gehen wir von einer Laufzeitausführung aus, andernfalls vom Entwicklungszeitpunkt.

(Listing 8 - Dynamische Festlegung des Ressourcen-Pfades)

```
let resourcesDir = process.resourcesPath
try {
    await require('fs').promises.access(resourcesDir+'/lib')
} catch (ex) {
    resourcesDir = app.getAppPath()
}
```

Darüber hinaus nimmt die Anwendung aktuell an, dass auf dem System eine Java-Ausführungsumgebung vorhanden ist. Einfacher wird die Verbreitung der Anwendung aber, wenn die JRE direkt mit in die Anwendung eingebettet wird. Dazu legen wir die entpackten JREs (die man zum Beispiel von adoptopenjdk.net[10] direkt herunterladen kann) für alle von uns unterstützen Betriebssysteme in den Projektordner und benennen die Verzeichnisse entsprechend der Architektur um in **`jre-mac-x64`**: **`jre-win-ia32`** usw. In der **`package.json`** ergänzen wir im Bereich **`extraResources`** den Eintrag **`jre-${os}-${arch}`**, damit die jeweils passende JRE zur Anwendung gepackt wird, und in der **`main.js`** bestimmen wir den Pfad zum Java-Binary wie in **(Listing 9)** dargestellt.

(Listing 9 - Pfad zum Java-Binary in der main.js bestimmen)

```
const os = (process.platform == "darwin") ? "mac" :
           (process.platform == "win32")  ? "win" : "linux"
let javaBin = `${resourcesDir}/jre-${os}-${process.arch}/bin/
java`
if (os == "mac") { // Mac-Java has a different directory struc-
ture
    javaBin = javaBin.replace('bin/','Contents/Home/bin/')
}
```

Anwendung testen mit Spectron und QF-Test

Zum Electron-Projekt gehört das Spectron-Testframework[11], welches auf Chromedriver und dem WebdriverIO-Projekt aufsetzt und zur Testautomatisierung über ein Application-Objekt Zugriff auf die Electron-Anwendung bietet. In Kombination mit einer Javascript-Testumgebung (zum Beispiel den Node.js-Modulen mocha und chai) können so Testskripte erstellt werden. Dazu installiert man die notwendigen Module über **`npm install spectron mocha chai chai-as-promised --save-dev`**, ergänzt in der **`package.json`** das Skript **`test`** : **`mocha test`** und erstellt dann eine Testdatei **`test.js`** wie in **(Listing 10)** dargestellt.

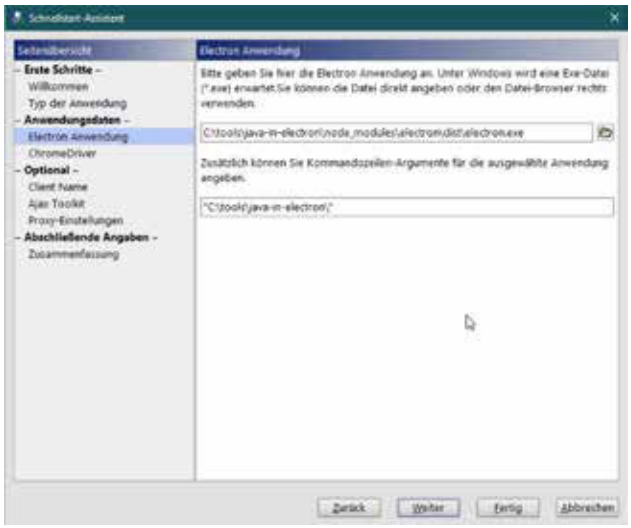
(Listing 10) Typischer Test mit Mocha, Chai und Spectron

```
var Application = require('spectron').Application
var chai = require('chai')
chai.should()
chai.use(require('chai-as-promised'))

let app

before(() => {
    app = new Application({
      path: './node_modules/electron/dist/electron.exe',
      args: ['./']
    })
    return app.start()
})

describe('The app', () => {
    it('loads splash screenproperly', function() {
        return app.client.waitUntilWindowLoaded()
    })

    it('loads initial form', () => {
        return app.client.waitUntil(() => {
            return app.client.element('.v-generated-body').
            isExisting()
        },9999)
    }).timeout(10000)

    it('loads splash screenproperly', function() {
        return app.client.waitUntilWindowLoaded()
    })


    it('loads initial form', () => {
        return app.client.waitUntil(() => {
            return app.client.element('.v-generated-body').
            isExisting()
        },9999)
    }).timeout(10000)

    it ('presents correct data', () => {
        return app.client
            .getText('.v-table-table .v-table-row:nth-of-type(1)
            ' +
                     '.v-table-cell-content:nth-of-type(2)')
            .should.eventually.equal('12.300,00 €')
    })

    it ('calculates price correctly', () => {
        return app.client
            .click('.v-table-table .v-table-row:nth-of-type(1)')
            .getText('.v-gridlayout-slot:nth-of-type(9)')
            .should.eventually.equal('12.300,00 €')
    })

})

after(() => {
    if (app && app.isRunning()) return app.stop()
})
```

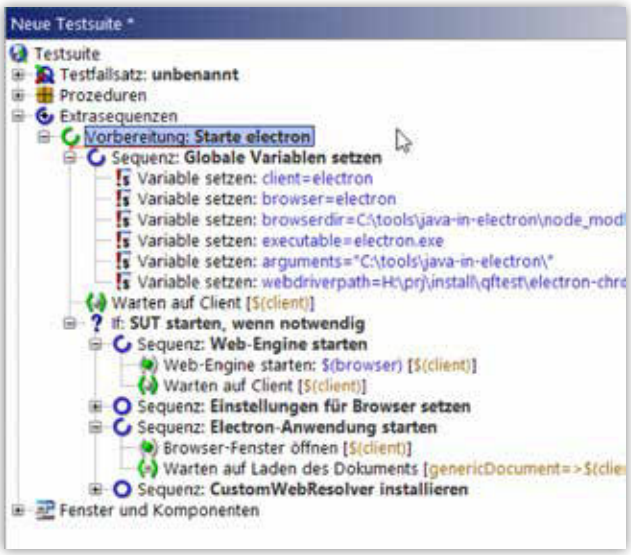
Im **`before`** Teil wird die Electron-App gestartet. Dazu wird zur Entwicklungszeit das Electron-Binary mit dem Pfad des Projektes als erster Parameter verwendet. Da die Testdefinition auf WebdriverIO aufsetzt, stellen sich auch bei Tests die typischen Herausforderungen, denen man auch zum Beispiel bei Tests mit Selenium begegnet, wie zum Beispiel die aufwendige Adressierung der Elemente.

Während sich Spectron also tendenziell direkt an Entwickler richtet, erlaubt QF-Test auch Testern ohne Programmierkenntnissen einen einfachen Zugriff auf die Electron-Anwendung. Die Tests in QF-Test können zwar mit Hilfe von Skripten beinahe beliebig ausgebaut werden, doch gleichzeitig bleibt es durch Funktionen wie etwa Capture-Replay und dem Komfort einer grafischen Benutzeroberfläche leicht zugänglich.

Durch die Unterstützung verschiedener UI-Technologien können bestehende Tests jetzt direkt für die Electron-Anwendung



Test der Electron-Anwendung zur Entwicklungszeit. (Abb. 3)

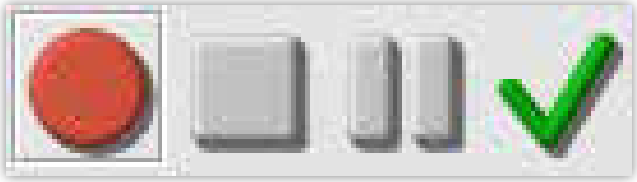


Eine Startsequenz für den Test einer Electron-Anwendung. (Abb. 4)

wiederverwendet werden, lediglich die Startsequenz benötigt Anpassungen. Dazu wählt man im **Extras** Menü den Schnellstart-Assistenten und dort die Option **Eine Electron Anwendung testen**. Im zweiten Schritt muss die zu testende Electron-Anwendung angegeben werden. Dies kann entweder die Anwendungsdatei sein, welche beim Build erzeugt wurde, oder man gibt hier die Electron-Binärdatei aus dem Ordner **node_modules/electron/dist** an und spezifiziert als Argument das Projektverzeichnis (Abb. 3).

Im nächsten Schritt frägt QF-Test nach der Electron-Version, die bei der Anwendungsentwicklung verwendet wurde und dem passenden ChromeDriver. In den meisten Fällen kann QF-Test automatisch die korrekte Version erkennen und den entsprechenden Treiber herunterladen. Weitere Voreinstellungen sind optional und im Assistenten selbst beschrieben, für unsere Anwendung aber nicht nötig. Der entstandene Vorbereitungsknoten stellt nun automatisch die Verbindung zu unserer Anwendung her (Abb. 4).

Sobald die Verbindung zum SUT aktiv ist, können Testbausteine aufgenommen und Tests abgespielt werden. Die Aufnahme von User-Inputs und Checks ist eine große Hilfe, möchte man schnell zu einigen ersten Tests gelangen. Hierfür genügt ein Klick auf den Aufnahmeknopf (Abb. 5), und folgende Interaktionen mit Komponenten der GUI werden aufgezeichnet. Es ist hilfreich, bestimmte Aktionssequenzen als Prozeduren im Prozedurenpaket abzuspeichern. Auf diese Weise lassen sich Tests schneller zusammenbauen und bleiben übersichtlicher. Zudem erlaubt diese Methode den Einsatz von Keyword-Driven-Testing, einem Testverfahren mit dessen Hilfe Tester mit wenig Erfahrung zum benutzten Test-Tool und ohne Programmierkenntnisse Tests aus vorher erstellten Bausteinen zusammensetzen können.



Aufnahme & Checks in der Toolbar. (Abb. 5)

Im Vergleich mit den für die Vaadin-Version erstellten Tests ergeben sich nur wenige Unterschiede. Die Komponenten unterscheiden sich zwischen beiden Formen nicht, die Startsequenzen sind sehr ähnlich. Zusätzlich ist aber hier ein Zugriff auf die nativen Menüs möglich: Er erfolgt über sogenannte Auswahlknoten, die auch entsprechend aufgenommen werden können.

Fazit:

Electron bietet Entwicklern die Möglichkeit, Web-Anwendung schnell und unkompliziert desktoptauglich zu machen und an eine breite Kundenbasis zu verteilen. Mit Tool-Unterstützung lassen sich Java-Programme schnell dorthin transferieren, Fehler bei der Migration zu vermeiden. Die in diesem Artikel dargestellten Schritte bieten eine erste Grundlage, auf die individuell durch weitere Optimierung in der Konfiguration der verwendeten Komponenten **aufgebaut** werden kann.

Quellen:

1 https://atom.io
2 https://electronjs.org
3 https://www.qf-test.com
4 https://java-pro.de/rapidclipse4
5 https://vaadin.com
6 https://www.eclipse.org/jetty/documentation/9.4.x/runner.html
7 https://github.com/electron/electron-quick-start
8 Von https://central.maven.org/maven2/org/eclipse/jetty/jetty-runner/9.4.18.v20190429/
9 https://www.electron.build
10 Am besten direkt die „zip“ bzw. „tar.gz“ des JREs anstelle des Installers auswählen
11 https://electronjs.org/spectron