Wie, ich soll jetzt was in JavaScript schreiben?

Vor Einführung der Produktorganisation Ende 2016 gab es ein dediziertes Team, dass sich mit Frontendtechnologien wie JavaScript und Cascading Style Sheets (CSS) beschäftigt hat. Die einzelnen Entwicklungsteams fokussierten sich auf die Backendentwicklung in Java. Seit Einführung der Produktteams gibt es kein zentrales Team mehr, das die Frontendentwicklung für alle übernimmt. Stattdessen ist jedes Produktteam selbst dafür verantwortlich, ganz im Sinne von Self-Contained Systems (SCS) bzw. Micro Frontends.

Im Team Kaufen haben wir vor eineinhalb Jahre mit relativ einfachen Mitteln angefangen, JavaScript und CSS zu schreiben, und seitdem unser Vorgehen und Tooling stetig weiterentwickelt.

Die ersten Gehversuche

Das erste eigene Frontend haben wir im Team Kaufen Ende 2017 angefangen zu entwickeln. Der einfachste Weg zur Auslieferung von JavaScript und CSS war die Integration von Inline-Skript in unseren HTML-Code. Die Skripte landeten dadurch eins-zu-eins so im Browser, wie wir sie geschrieben hatten.

Durch dieses Vorgehen benötigten wir kein besonderes Tooling, sondern konnten schnell in die Entwicklung der benötigten Funktionalitäten einsteigen. Gleichzeitig ergeben sich bei dieser Vorgehensweise auch verschiedene Nachteile:

- Aufgeblähter Code im Browser
- Codekommentare im Browser
- kein Caching des Codes
- keine Tests des JavaScripts
- globale Variablen in JavaScript
- kein Sicherstellen, dass JavaScript in allen gängigen Browsern ausführbar ist

Um diese Probleme zumindest teilweise zu adressieren, haben wir nach einer

Weile begonnen, den CSS-Code in eigene Dateien auszulagern und durch Maven-Plugins zu verkleinern. So konnten wir die CSS-Dateien dann über Link-Tags im HTML einbinden und vom Caching durch den Browser profitieren. Die Verarbeitung über Maven fügte sich sehr gut in unseren bestehenden Buildprozess. Uns wurde allerdings schnell klar, dass die Verarbeitung über Maven nicht zukunftsfähig ist. Da wir nach und nach insbesondere immer mehr JavaScript geschrieben haben, erkannten wir, dass wir uns mit den Werkzeugen aus dem Node Package Manager (NPM)-Kosmos beschäftigen müssen, um uns hier zukunftsfähig aufstellen zu können.

Auf in neue Gefilde

Im ersten Schritt wollten wir die folgenden Punkte verbessern:

- Kein Inline-Skipt mehr. Stattdessen sollen JavaScript- und CSS in eigenen Dateien entwickelt und auch als solche ausgeliefert werden.
- Verarbeitung von JavaScript und CSS, wie Minify und Uglify, damit der Code kleiner wird, keine Kommentare mehr im Browser sichtbar sind und der ausgelieferte Code schlechter lesbar ist.
- Sicherstellen, dass JavaScript in allen gängigen Browsern ausführbar ist, durch Linting, dass prüft, ob JavaScript in Version ECMA Script 5 (ES5) geschrieben ist.
- JavaScript-Abhängigkeiten sollen über NPM bezogen werden können, anstatt über WebJars, die wir bisher über Maven gezogen hatten.

Resultat der Verbesserung unseres Buildprozesses war der Einsatz von

- Yarn als Package Manager und Alternative zu NPM, der aber dennoch die NPM Repositories nutzen kann
- Gulp zur Automatisierung der verschiedenen notwendigen Buildschritte inkl. zahlreicher Plugins für Gulp zur Verarbeitung des JavaScript- und CSS-Codes
- Linting mit ESLint
- Bezug externer JavaScript-Abhängigkeiten über NPM

Unsere gesetzten Ziele konnten wir so erreichen. Trotzdem hatten wir noch ein paar wesentliche Aspekte zu verbessern.

- Zur besseren Wartbarkeit des JavaScript-Codes hatten wir mehrere JavaScript-Dateien, die dann auch einzeln ausgeliefert wurden. Dadurch gab es Abhängigkeiten zwischen den Dateien, die beim Einbinden in den HTML-Code zu berücksichtigen waren.
- Die JavaScript-Dateien waren relativ groß. Dadurch war der Code schwer lesbar und Refactorings waren sehr aufwändig und fehleranfällig.
- Der JavaScript-Code wurde nach wie vor durch globale Variablen angesteuert, bspw. durch Click-Events im HTML-Code.
- Fehler im JavaScript-Code fielen erst relativ spät in unserer Build & Deployment Pipeline auf, da wir mit der vorhandenen Struktur keine Unittests schreiben konnten.

Modularisierung des JavaScript-Codes

Zur Adressierung der genannten Schwachstellen wollten wir unseren JavaScript-Code in erster Linie in kleinere Module aufteilen, die dann durch den Buildprozess in einer Datei gebündelt werden.

Zunächst haben wir analysiert, ob sich für unsere Zwecke der Einsatz eines der gängigen JavaScript-Frameworks wie React, Angular oder Vue.js eignet. Entschieden haben wir uns allerdings für einen Model-View-Controller (MVC)-Ansatzes mit Vanilla JavaScript. Dieser Ansatz ist bspw. leichtgewichtiger und besser mit unserem Einsatz der Template-Engine Handlebars, die wir für die HTML-Generierung aktuell einsetzen, zu vereinbaren.

Zur konsequenten Umsetzung eines MVC-Patterns haben wir unseren JavaScript-Code in viele kleine JavaScript-Dateien unterteilt, je eine für Model, View und Controller. Das Bündeln der einzelnen kleinen Dateien zu einer großen Datei, die dann auch über ein Script-Tag im HTML eingebunden wird, erfolgt im Buildprozess mit Webpack. Damit Webpack die einzelnen Dateien zu einer zusammenfügen kann, definieren die einzelnen Dateien Node.js-Module. Das Resultat dieses Buildprozesses ist eine anonyme JavaScript-Datei, die keine globalen Variablen definiert, sich selbst auf HTML DOM Events, wie etwa Click-Events, registriert und keine Abhängigkeiten zu anderen Dateien hat.

Im ersten Schritt wollten wir auf ECMA-Script 6 (ES6)-Funktionalitäten verzichten und uns stattdessen strikt an die JavaScript ES5-Version halten. So

mussten wir in unseren Buildprozess nicht auch direkt eine Konvertierung von ES6 zu ES5 einbauen, sondern können diesen möglichst simpel halten.

Durch die kleinen JavaScript-Module waren wir auch in der Lage, einfache Unittests für unseren Code zu schreiben. Nach einer kurzen Sondierung der zahlreichen Frameworks haben wir uns für die Nutzung von Jest entschieden. Durch diese Tests konnten wir unsere Codequalität maßgeblich steigern.

Kommt da noch mehr?

Der aktuelle Aufbau unseres JavaScript-Codes funktioniert so für uns weitestgehend schon sehr gut. Trotzdem gibt es Punkte, die wir in der Zukunft erneut evaluieren und verbessern werden:

- Einsatz von ES6 statt ES5: ES6 bietet einige Möglichkeiten, den Code lesbarer zu gestalten. Zudem sind wir von der Strukturierung über Node.js-Module nicht überzeugt und würden stattdessen lieber auf ES6-Module bzw. Klassen setzen.
- Ausführbarkeit in allen gängigen Browsern sicherstellen durch Browser-spezifische Tests oder Polyfills, bspw. durch den Einsatz von Babel
- Schnelleres Feedback und bessere Qualität durch Browser-spezifische Tests von zusammenhängenden JavaScript-Module als Erweiterung zu den Unittests mit Jest