Bereitstellung von Testdaten einmal smart

Wir, das Team AiM (Artikel im Mittelpunkt), haben mit dem Testdatentool eine Anwendung entwickelt, um anderen Teams Testartikel zur Verfügung zu stellen. Diese Anwendung möchte ich in diesem Blogartikel vorstellen.

Das Problem der Beschaffung und Bereitstellung von Testdaten

In meinen sieben Jahren als QA-ler bei der Thalia ist mir ein Problem immer wieder begegnet: Sobald Testdaten in anderen Teams liegen, nimmt deren Beschaffung viel Zeit und Energie in Anspruch. Nicht zu vergessen die Aufwände auf Seiten der Teams, die Testdaten bereitstellen.

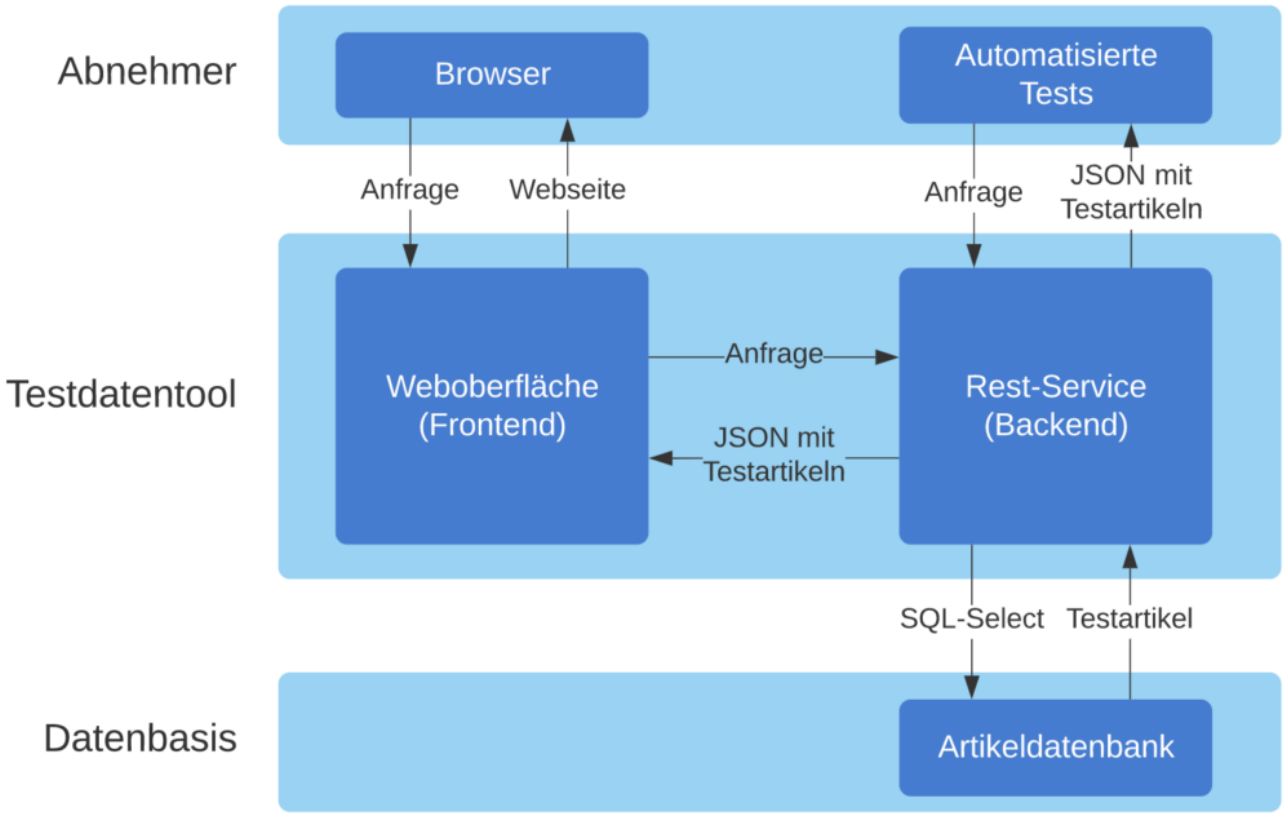
Über viele Jahre hinweg haben wir im Team AiM für andere Teams Testartikel über individuelle SQL-Abfragen herausgesucht und über die Herausgabe der Artikel-ID oder EAN bereitgestellt. Dieser Prozess hat sich mit der Zeit als sehr ineffizient herausgestellt. Um diesen Prozess zu optimieren haben wir das Testdatentool entwickelt, über welches alle Teams eigenständig Testartikel beziehen können.

Das Testdatentool stellt zwei Dienste zur Verfügung. Zum einen bietet es eine **Weboberfläche**, welche die Suche nach Testartikeln über alle gängigen Browser ermöglicht. Zum anderen stellt das Testdatentool einen **REST-Service** bereit, den Entwicklerteams im Rahmen ihrer automatisierten Tests einbinden können, um dynamisch Testartikel zu beziehen.

Aufbau

Die Weboberfläche und der REST-Service bilden zusammen das **Frontend** und das **Backend** des Testdatentools (siehe Abbildung). Während das Frontend die Suchanfrage durch den Anwender über eine Weboberfläche entgegennimmt, führt das Backend die Suche über einen REST-Service auf der Datenbank aus, um die gefundenen Treffer im JSON-Format zurückzugeben. Die Begriffspaarungen

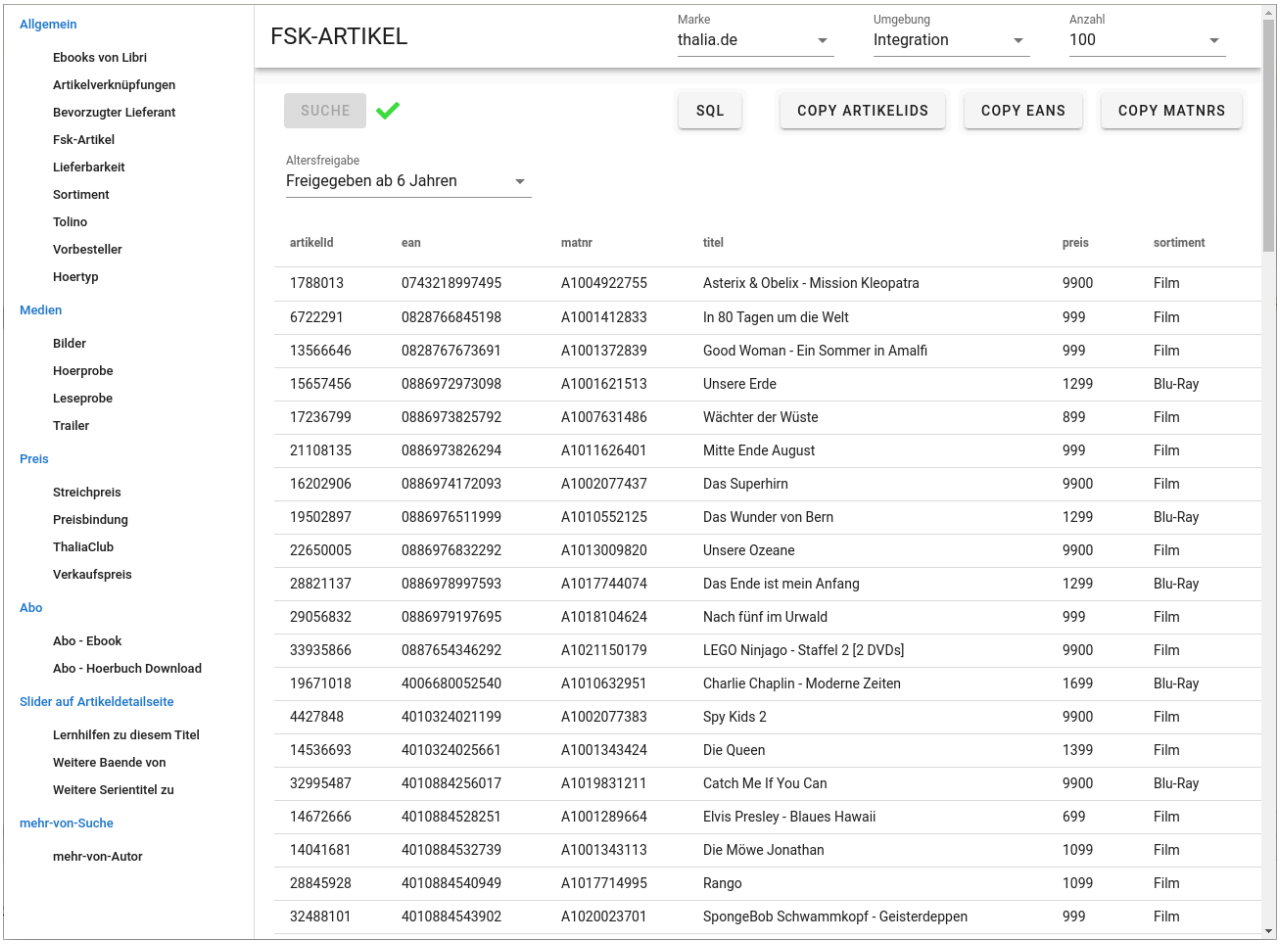
Weboberfläche/REST-Service und Frontend/Backend werden in diesem Blogartikel synonym verwendet.



Weboberfläche

Die Weboberfläche des Testdatentools ermöglicht es dem Anwender über wenige Klicks einen passenden Testartikel zu finden (siehe Abbildung). Insgesamt bietet die Weboberfläche 23 Kategorien, die für die Suche nach einem Testartikel zur Verfügung stehen. Für jede Suche können die Marke (thalia.de, thalia.at, bol.de oder orellfuessli.ch), die Umgebung (Integrations- oder Produktivumgebung) und die Anzahl der benötigten Testartikel über entsprechende Comboboxen eingestellt werden. Einige Kategorien können zusätzlich über weitere, individuelle Filter konkretisiert werden. Die Abbildung veranschaulicht einen solchen zusätzlichen Filter für die Kategorie „FSK-Artikel“. Nachdem die Suche ausgeführt und die gefundenen Artikel in der Trefferliste angezeigt werden, kann der Anwender sich das zu Grunde liegende SQL über den Button „SQL“ anzeigen lassen. Weiter kann der Anwender über den Button „COPY EANS“ alle EANS aus der Trefferliste in den Zwischenspeicher des Betriebssystems kopieren. Über

diesen können die EANS in jedes andere Programm übertragen werden. Die Buttons „COPY ARTIKEL-IDS“ und „COPY MATNRS“ verhalten sich entsprechend analog zu „COPY EANS“.



REST-Service

Der REST-Service besitzt für jede Kategorie, welche die Weboberfläche anbietet, eine REST-Ressource. Hinter jeder REST-Ressource verbirgt sich wiederum ein SQL, über das die Testartikel auf der Artikeldatenbank gesucht werden. In das SQL werden die Werte eingesetzt, die der Aufrufer über URL-Parameter mitliefert. Die gefundenen Testartikel werden in einem definierten JSON-Format zurückgeliefert.

Wir haben uns dazu entschlossen, den Funktionsumfang des REST-Service stets mit dem der Weboberfläche synchron zu halten. Denn während die Implementierung einer neuen Kategorie in der Weboberfläche ohne eine Anpassung des REST-Service nicht möglich ist, wäre es sehr wohl möglich, den

REST-Service für einen automatisierten Test um eine neue Kategorie zu erweitern, ohne diese in die Weboberfläche aufzunehmen. Indem wir die Weboberfläche und den REST-Service synchron halten, stellen wir sicher, dass die Entwicklerteams die Weboberfläche als bildhafte Dokumentation des REST-Service nutzen können.

Designentscheidungen

In diesem Kapitel beschreibe ich Designentscheidungen, die wir für das Testdatentool getroffen haben.

Bei der Implementierung der Weboberfläche haben wir besonders viel Wert auf eine einfache Erweiterbarkeit gelegt, damit die Weboberfläche auch ohne tiefgreifende Vue.js-Kenntnisse von allen Entwicklern des Teams erweitert werden kann. Das nachfolgende Codebeispiel verdeutlicht, dass lediglich die Erstellung einer neuen Klasse notwendig ist, um die Weboberfläche um eine neue Kategorie zu erweitern. Nachdem diese Klasse erstellt ist, muss diese nur noch in der main.js eingetragen werden, um die Erweiterung im Frontend abzuschließen.

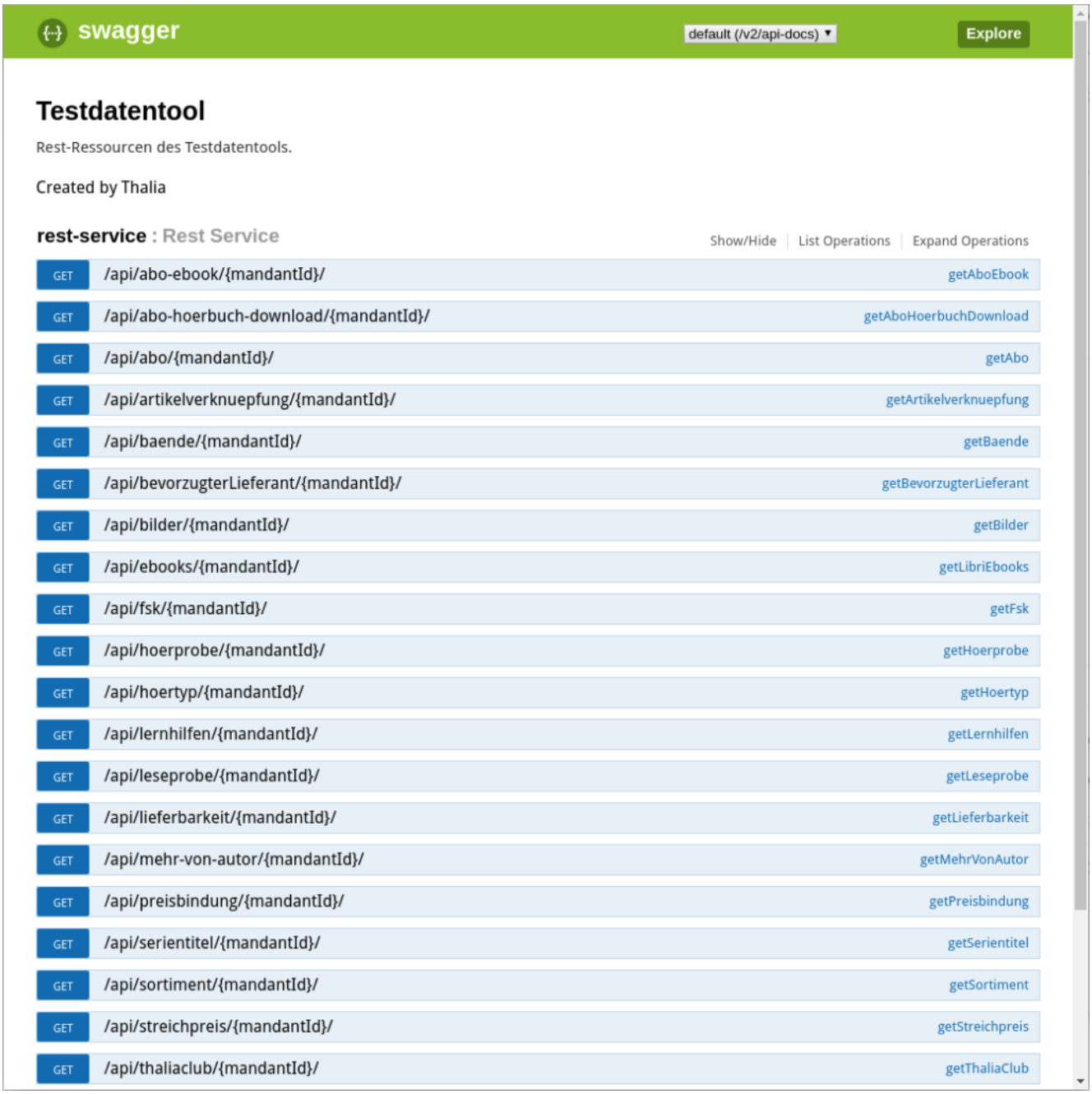
Fsk.vue | main.js

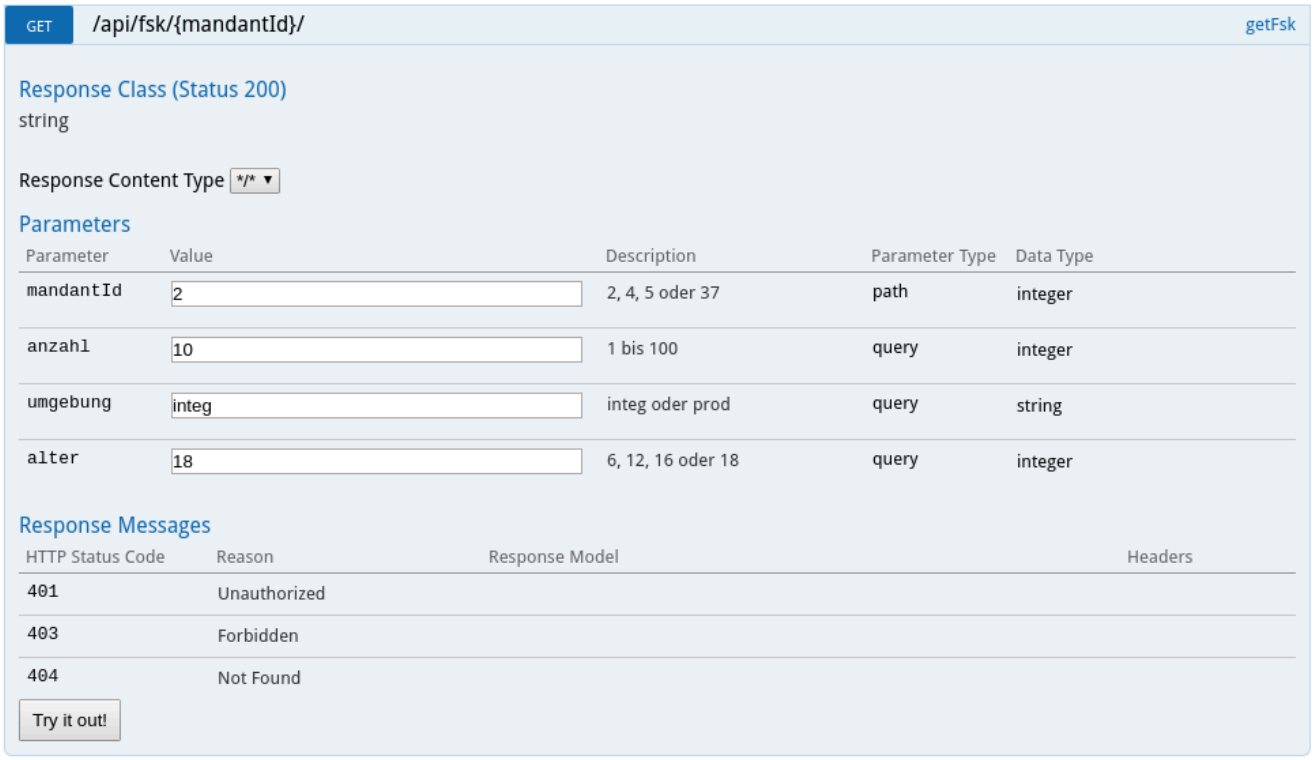
```vue
<template>
    <div>
        <theme :headline="headline"
               :resource="resource"
               :comboboxes="comboboxes"/>
    </div>
</template>

<script>

    export default {
        created() {
            // Legt fest, welcher Eintrag in der Combobox zu Beginn angezeigt wird.
            this.comboboxes[0].selectedValue = this.comboboxes[0].defaultValue
        },
        data() {
            return {
                // Bestimmt, welche Ueberschrift fuer das Suchthema im Frontend angezeigt wird.
                headline: 'Fsk-Artikel',
                // Bestimmt, wie die Rest-Ressource des Rest-Service aufgerufen wird (hier: /testdatentool-backend/api/fsk/)
                resource: 'fsk',
                comboboxes: [
                    {
                        // Eintraege, die in der Combobox angezeigt werden. Die Variable "name" legt fest, wie die Eintraege
                        // in der Combobox heissen. Die Variable "value" definiert, wie genau das Backend fuer den Eintrag
                        // aufgerufen wird. Waehlt der Nutzer in der Combobox den Eintrag "Freigegeben ab 18 Jahren", dann wird
                        // das Backend mit dem Query-Parameter ?alter=18 aufgerufen. Der Name des Query-Parameters (hier: alter)
                        // wird ueber die Variable "queryParamName" festgelegt.
                        entries: [
                            {name: 'Freigegeben ab 6 Jahren', value: '6'},
                            {name: 'Freigegeben ab 12 Jahren', value: '12'},
                            {name: 'Freigegeben ab 16 Jahren', value: '16'},
                            {name: 'Freigegeben ab 18 Jahren', value: '18'}
                        ],
                        // Verweist auf den Eintrag, den der Nutzer in der Combobox ausgewaehlt hat. Diese Variable ist eine
                        // Referenz auf die Variable "value" in dem Array "entries". Hat der Nutzer den Eintrag
                        // "Freigegeben ab 18 Jahren" ausgewaehlt, dann traegt die Variable "selectedValue" den Wert "18". Die
                        // Variable "selectedValue" wird immer dann aktualisiert, wenn der Nutzer ueber die Combobox eine neue
                        // Auswahl trifft. Die Variable wird zum ersten Mal in der Methode created() mit einem Wert versehen.
                        selectedValue: '',
                        // Verweist auf den Eintrag, der in der Combobox zu Beginn ausgewaehlt ist. Diese Variable ist eine
                        // Referenz auf die Variable "value" in dem Array "entries".
                        defaultValue: '18',
                        // Beschriftung der Combobox.
                        label: 'Altersfreigabe',
                        // Name des Query-Parameters, ueber den die Auswahl in der Combobox an das Backend uebergeben wird (hier: ?alter).
                        queryParamName: 'alter'
                    }
                ]
            }
        }
    }
</script>
```

Die zweite Designentscheidung betrifft die Optimierung des Antwortzeitverhaltens des Backends. Von Beginn an war klar, dass einige Suchen bis zu 10 Sekunden dauern würden, da die jeweiligen SQLs sehr komplex sind. Um die Antwortzeiten zu optimieren, wurde das Backend um einen Cache inklusive Cacheaufwärmmechanismus erweitert. Hierdurch wird sichergestellt, dass das Testdatentool auch bei Langläufer-SQLs von über 10 Sekunden, in unter 500 Millisekunden antwortet. Anders ausgedrückt garantiert der Cacheaufwärmmechanismus, dass für jede Langläufer-Suche zu jedem Zeitpunkt ein gültiger Cache-Eintrag existiert und die Suche aus dem Cache bedient werden kann.

Die letzte Designentscheidung betrifft ebenfalls das Backend und besteht in der Nutzung von Swagger zur Dokumentation der REST-Ressourcen. Swagger bietet unserem Team vor allem zwei Vorteile. Zum einen spart unser Team durch Swagger Zeit, da die Dokumentation mit wenig Aufwand erstellt werden kann.

Zum anderen ermöglicht Swagger eine inhaltlich ausdrucksstarke Dokumentation jeder einzelnen REST-Ressource. Dies ist besonders wichtig, da der REST-Service durch andere Teams genutzt wird und dessen Verwendung weitestgehend selbsterklärend sein soll. Die Einbindung von Swagger gestaltet sich dabei denkbar einfach. Nach der Aufnahme der Swagger-Dependency in die pom.xml und der Anreicherung der REST-Ressourcen um Metadaten zu jedem URL-Parameter, genügt eine Annotation über der REST-Service-Klasse, um Swagger einzubinden. Wie die Swagger-Dokumentation für den REST-Service aussieht ist der linken Abbildung zu entnehmen. Die rechte Abbildung zeigt die Detailansicht einer einzelnen Ressource.





Technologien

In diesem Kapitel möchte ich kurz die Technologien nennen, die für die Entwicklung des Testdatentools verwendet wurden. Weiter erläutere ich stichpunktartig die Gründe, welche für die Auswahl der Technologien ausschlaggebend waren.

Vue.js als Framework für die Weboberfläche – Vorteile:

- Vue.js setzt auf bekannte Technologien wie JavaScript, HTML und CSS
- Eine sehr aktive Community und eine umfassende Dokumentation mit vielen Best-Practice-Beispielen
- Steile Lernkurve, da das Konzept hinter Vue.js leicht zu verstehen ist
- Sehr gute Performance aufgrund einer effizienten Implementierung des virtual DOM

Spring-Boot als Framework für den REST-Service – Vorteile:

- Unser Team hatte bereits Erfahrungen im Umgang mit Spring-Boot
- Aufsetzen eines REST-Service ohne langwierige Konfigurationsarbeiten
- Einfacher Datenbankzugriff durch Spring-Boot-Funktionalitäten

Erfahrungsbericht

Dieser Erfahrungsbericht soll denen helfen, die eine Anwendung im Stile des Testdatentools für ihr eigenes Team in Erwägung ziehen.

Als wir das Testdatentool eingeführt haben, war die Resonanz durch die anderen Teams durchweg positiv. Insbesondere die Weboberfläche wurde von Beginn an intensiv genutzt und überzeugt seitdem durch ihre übersichtliche und einfache Bedienung. Auf der anderen Seite hat es einige Zeit gedauert, bis der REST-Service als Testartikel-Lieferant für automatisierte Tests bei den Teams Einzug erhalten hat. Tatsächlich gibt es immer noch Teams, die auf die Nutzung des REST-Service im Rahmen ihrer automatisierten Tests verzichten und stattdessen darauf zurückgreifen, Testartikel statisch in ihre Tests einzubinden. Dies ist für viele Teams völlig ausreichend, wenn Testartikel in den Tests nur eine untergeordnete Rolle spielen.

Das Testdatentool hat unserem Team dabei geholfen, die Anzahl der Anfragen, die sich um die Bereitstellung von Testartikeln drehen, um den Faktor vier zu reduzieren. Hierdurch hat sich die Entwicklungszeit des Testdatentools nach kurzer Zeit bezahlt gemacht. Nicht zu vergessen, dass es auch auf Seiten der Teams, die nach Testartikeln suchen, zu Zeiteinsparungen kommt. Abschließend kann ich mit Überzeugung sagen, dass sich dessen Einführung für uns und alle Beteiligten gelohnt hat.