CultureMix Backend Dokumentation

1. Über das Projekt

Das CultureMix Backend wurde entwickelt um Metadaten von Kulturschaffenden in Berlin durchsuchbar und editierbar zu machen. Ein Ziel des Projekts war es, die Daten zentral und effizienter zu erfassen und über eine API bereitzustellen. Diese API soll ermöglichen, dass verschiedene Akteure die Daten nutzen und darstellen können.

2. Komponenten

Das CultureMix Backend besteht aus zwei verschiedenen Komponenten: API und CMS. Die API dient als Schnittstelle zur Datenbank. Das CMS ist eine Oberfläche zum Editieren der Datenbank.

Repositories:
API https://github.com/wbkd/tsb-generic-api
CMS https://github.com/wbkd/tsb-generic-cms

3. API Einführung

Der API-Server wurde mit Node.js bzw. Hapi.js implementiert. Als Datenbank wurde MongoDB verwendet. Bei der Umsetzung wurde besonders darauf geachtet, dass die API möglichst generisch aufgebaut ist. Das bedeutet, dass Entitäten sowie Felder in der Datenbank durch einfache Konfiguration hinzugefügt und geändert werden können. Das generische System basiert auf der Hapi-Bibliothek “Joi”, mithilfe derer Schemas für die einzelnen Tabellen der Datenbank konfiguriert werden können. Mit Joi wird außerdem die Datenvalidierung gewährleistet. Die definierte Schemas werden in mongoose models übersetzt um eine schnittstelle zu der Datenbank zu ermöglichen. Gewisse *Joi* Funktionen können nicht automatisch in *mongoose* schemas übersetzt werden und werden daher nicht unterstützt, diese kann man hier finden.

4. API Installation

Um den API-Server zu starten, sind folgende Anforderungen erforderlich:

1. Node.js (empfohlene Version: ab v10)
2. Eine MongoDB-Instanz als Datenbank (empfohlene Version: ab v4)
3. Ein AWS S3 Bucket für den File-Upload
4. Ein HERE Maps API Key für das Geocoding

Die jeweiligen Credentials der externen Services werden mittels einer *.env* Datei bereitgestellt. Diese sollte beispielhaft folgendermaßen aussehen:

```
REDIS_URL=localhost:6379
MONGODB_URI=mongodb://localhost:27017/data
PORT=8000

AWS_ACCESS_KEY_ID=AFIOIWTMSLDG4593ASK
AWS_SECRET_ACCESS_KEY=iqlaKF3902FFewerio59JEWe94Kwe352Koep93

ADMIN_EMAIL=test@webkid.io
ADMIN_PASSWORD=ASecurePasswordForThe1AdminUser!

HERE_APP_ID=oW4i5omfEWO3453SDf
HERE_APP_CODE=ok3259F8Fk923K53ko3Ko57
```

Um eine*n Admin-Nutzer*in anzulegen, können die Environment-Variablen *ADMIN_EMAIL* und *ADMIN_PASSWORD* genutzt werden. Beim Start des Servers wird mittels dieser Variablen ein*e Admin-Nutzer*in erstellt. Falls ein*e Admin-Nutzer*in mit der angegebenen E-Mail bereits existiert, kann man über die *ADMIN_PASSWORD* Variable das zugehörige Kennwort ändern.

Zusätzlich müssen zur Vorbereitung noch die Dependencies der Node.js-Anwendung installiert werden. Dies ist üblicherweise über den Befehl *npm install* möglich.

Nach der Vorbereitung der oben stehenden Schritte kann die Anwendung mit dem Befehl *npm start* gestartet werden. Die Anwendung läuft dann unter dem Port, der in der Environment-Variable PORT spezifiziert wurde.

5. API Struktur

Als Einstiegs- und Konfigurationsdatei der API dient die manifest.json. In dieser JSON-Datei werden die einzelnen Routen (wie zum Beispiel /users, /files) konfiguriert. Zudem werden hier verschieden Hapi-Plugins geladen (zum Beispiel hapi-swagger zur automatischen Dokumentation von Endpoints). Weitere Endpoints, auch der “generische” Endpoint für die Entitäten werden mittels eigener Hapi-Plugins implementiert. Diese Struktur ermöglicht zum einen eine einfache Konfiguration, zum anderen aber auch eine gewisse flexibilität, da spezielle

Endpoints einfach per Plugin hinzugefügt werden können und somit auch ihre eigene Funktionalität mitbringen können.

Der Code zur Implementierung der generischen Routen ist unter */src/api/crud* abgelegt. Weitere Custom-Routen wie zum Beispiel /users oder /changes mit speziellen Funktionalitäten sind als eigene Plugins in /src/api/… definiert.

- */src/api/crud* generiert eine CRUD JSON Schnittstelle für die Schemas die in */models* definiert sind
    - *index.js* liest die Konfiguration aus *manifest.json* um das Plugin zu initialisieren. Mögliche Konfigurationen sind:
        - *models* definiert den Pfad wo die Models zu finden sind
        - *cache* konfiguriert das Caching
            - *enabled* ob das caching Aktiviert ist (Boolean). Default: true
            - *expiresIn* maximale Zeit vor dem Ablauf des Cache (in ms). Default: 86400000 (24 Stunden)
    - *cache.js* ermöglicht caching von Anfragen solange keine neue Daten geschrieben/aktualisiert werden
    - *router.js* definiert die HTTP Endpoints die für Schemas zur verfügung gestellt werden
    - *controller.js* ist für das Parsing von der Abfrage zuständig und die Weiterleitung der Abfrage zu *service.js*
    - *service.js* wandelt die Anfrage in eine Datenbankabfrage um
    - *validation.js* konfiguriert die Validierung der Abfragen anhand des Schema
- */src/api/changes* stellt eine Schnittstelle für den Korrekturprozess zur Verfügung
- */src/api/users* stellt eine Schnittstelle für die Benutzerverwaltung zur Verfügung
- */src/api/files* stellt eine Schnittstelle für das Hochladen von Dateien zu AWS S3 zur Verfügung
- */src/api/email* erlaubt das automatisierte Versenden von E-Mails
- */src/api/db* konfiguriert die Kommunikation zur Datenbank mittels *mongoose*
- */src/api/policies* definiert die Policies um den Zugang zu bestimmten Routen zu beschränken oder um bestimmte Funktionen vor und nach eine Abfrage durchzuführen. Diese können dann in *router.js* eingesetzt werden

6. API Models

Die Struktur der Datenbank wird mithilfe der models unter */models* erzeugt. Für jede Tabelle in der Datenbank wird ein eigenes Joi-Schema definiert (zum Beispiel: *events.js*). Es stehen die

herkömmlichen Joi-Funktionen zur Verfügung. Über *.meta({ component: 'textinput' })* kann für ein Feld in der Datenbank definiert werden, welches Component im Frontend ausgespielt werden soll, um dieses Feld zu bearbeiten. Es können weitere properties über .meta() mitgegeben werden, die im Frontend ausgelesen werden können.

Wenn ein neues Schema erstellt wurde, werden dafür automatisch drei Endpoints erstellt:

- */name* - endpoint zum Abruf mehrere Elemente der Collection als Array (per default mit *limit=10*, siehe Parameters unten)
- */name/id* - endpoint zum Abruf eines Elements der Collection als JSON
- */name/schema* - endpoint zum Abruf einer JSON-Darstellung des Joi-Schemas (vor allem für die Erstellung der Forms im CMS)

Der endpoint zum Abruf mehrere Elemente bietet zudem automatisierte Filter-Möglichkeiten. Diese haben folgende Syntax:

- *?fieldname=value* - filtert die Liste nach Elementen bei denen *fieldname* === value entspricht
- *?fieldname={ "nested": value }* - filtert die Liste nach Elementen bei denen fieldname.nested === value entspricht
- *?ids=["id1", "id2"]* - liefert alle Elemente zurück, die als id im ids parameter übergeben werden
- *?fields=["fieldname", "fieldname2.nested"]* - liefert die Liste nur mit den angegebenen Feldern zurück. Dies ist vor allem hilfreich um die übertragene Datenmenge zu verringern
- *?sort=fieldname* - liefert eine sortierte liste zurück
- *?limit=n* - liefert n elemente zurück (*limit=0* für alle Elemente, default 10)
- *?skip=n* - überspringt die ersten n Elemente der Liste
- ?or=[*{ "fieldname1": value }, { "fieldname2": value }]* - filtert die Liste nach elementen bei denen *fieldname1* === value oder *fieldname2* === value
- *?q=Value* - Textsuche nach *Value*

Über die limit und skip Parameter wird per default auch eine Pagination geliefert.

7. E-Mail Templates

Bei dem bestehenden System werden automatisiert E-Mails verschickt. Die entsprechenden Templates für die E-Mails sind in */src/api/email/templates* abgelegt und können dort geändert werden. (from, subject, text)

8. CMS

Das CMS ist eine React-basierte App zur Darstellung und Änderung der Daten der API. Bei der Entwicklung wurde vor allem darauf geachtet, dass die generische Struktur der API weitergeführt wird. Das bedeutet, dass die Seiten, die im CMS angezeigt werden, größtenteils dynamisch je nach Struktur der API generiert werden. Spezielle Anpassungen, die für einen bestimmten use-case erforderlich sein können, sollten trotzdem einfach hinzuzufügen sein.

9. CMS Aufbau und Struktur

Das CMS basiert grundlegend auf dem webkid-react-starterkit (https://github.com/wbkd/react-starter). Dieses ist bereits vorkonfiguriert mit webpack, react und styled-components. Zudem wird für das state management easy-peasy verwendet, welches auf redux und immer.js basiert.

Zur Installation und Start der App kann der Anleitung https://github.com/wbkd/react-starter/blob/master/README.md gefolgt werden.

Die Webpack-Konfiguration befindet sich unter /webpack. Dort wird die grundlegene Konfiguration in /webpack/webpack.common.js definiert. Zusätzlich gibt es eigene Konfigurations-Dateien für Development und Production.

Es gibt zwei Einstiegspunkte:

1. src/index.html -> das HTML-Grundgerüst, in das die React-App gerendert wird
2. src/index.js -> der Einstiegspunkt für den Javascript-Code in dem die React-Anwendung gerendert wird

Da die CSS-Styles ausschließlich in Javascript mittels styled-components geschrieben werden, gibt es keinen Einstiegspunkt für das CSS.

Im Ordner der Anwendung liegt außerdem eine Datei config.json, die global für die Anwendung verfügbar ist (mittels webpack alias: import config from ‘config’). Diese config-Datei wird verwendet um den sonst generischen Code anzupassen. Hier werden beispielsweise die Urls zu der API oder die verschiedenen Entitäten konfiguriert, die mit dem CMS editiert werden sollen.

10. CMS-API-Anbindung

In der config.json gibt es den Punkt “api”. Hier werden die Endpoints der API angegeben, die für die Kommunikation zwischen CMS und API genutzt werden sollen. Das aktuelle Konfigurations-Objekt sieht folgendermaßen aus:

```
"api": {
  "base": "https://tsb-kulturb-api.herokuapp.com/api/v2",
  "files": "https://tsb-kulturb-api.herokuapp.com/api/v2/files",
  "user": {
    "base": "https://tsb-kulturb-api.herokuapp.com/api/v2/users",
    "login": "/login",
    "refreshToken": "/refreshToken",
    "confirmEmail": "/confirm-email",
    "requestPasswordReset": "/reset-password",
    "changePassword": "/change-password",
    "resendConfirmationEmail": "/resend-confirmation-email"
  }
},
```

Wie man sieht, können verschiedene Base-URLS für die Nutzerverwaltung, Fileupload und Daten-Api (base) angegeben werden. Außerdem muss definiert werden, wie die Endpoints für die Nutzerverwaltung benannt sind.

11. CMS Konfiguration

Mit der config.json lässt sich einstellen, welche Menüpunkte in der Seitenleiste des CMS sichtbar sein sollen und welche Funktionen diese Menüpunkte haben sollen.

Dafür ist der Punkt “routes” vorgesehen. In diesem Array sind Route-Konfigurations Objekte aufgelistet, die in der Seitenleiste angezeigt werden und bestimmten Nutzer*innengruppen zugänglich gemacht werden können. Ein Route-Konfigurations-Objekt hat folgende Eigenschaften:

```
{
      // der Name der Entität (wird in der Seitenleiste angezeigt)
    "name": "Veranstaltungen",
```

```
      // Beschreibung der Entität (wird auf der Startseite angezeigt)
      "description": "Veranstaltungen können nur angelegt werden, wenn in der
Datenbank der dazugehörige Veranstaltungsort hinterlegt ist.",
      // Der Endpoint (sowohl in der API als auch im CMS)
    "endpoint": "/events",
      // Ein Icon für die Sidebar (optional) -> siehe
      // https://rsuitejs.com/tools/icons
    "icon": "calendar",
      // Soll die Entität durchsuchbar sein? -> es wird ein Suchfeld über der Liste
angezeigt
    "searchable": true,
      // Sollen Nutzer*innen eine neue Entität erstellen können? -> es wird ein
“Neu”-Button über der Liste angezeigt
    "createable": true,
      // Die Konfiguration der Liste
    "list": {
      // welche Spalten sollen in der Liste angezeigt werden? Label: Spaltenname,
path: Accessor für den Wert im Object, sortKey: Nach welcher Property soll sortiert
werden können?, dataType: Custom-Data type zur Anzeige in der Tabelle
      "columns": [
        {
          "label": "Titel",
          "path": "general.title"
        },
        {
          "label": "Highlight",
          "path": "general.isHighlight"
        },
        {
          "label": "Ort",
          "path": "dates[0].venue.general.name",
          "sortKey": "dates.venue"
        },
        {
          "label": "Datum",
          "path": "dates[0].date.from",
          "sortKey": "dates.date.from",
          "dataType": "date"
        }
```

```
      ]
    },
      // query-parameter die beim request an die API mitgeschickt werden
    "params": {
      // nur bestimmte fields aus dem objekt mit ausgeben (performance)
      "fields": [
        "general.title",
        "general.isHighlight",
        "dates.venue",
        "dates.date.from"
      ],
      // custom-parameter für die culturemix-api um vergangene dates mit auszugeben
      "pastdates": "true"
    }
  }
```

Zusätzlich gibt es noch weitere properties zur Konfiguration:

```
// welche Usergruppen sollen auf den Endpoint zugreifen können
"roles": ["ADMIN"],
// Soll bei den Requests das User-Token mitgeschickt werden?
"authenticate": true,
// Custom Bearbeitungs-Component
"editComponent": "UserEdit",
// Custom Create-Route (bei Klick auf den Create Button wird zu /signup
weitergeleitet)
"createRoute": "signup",
```

12. CMS Forms

Um die Daten aus der API im CMS anzuzeigen und editierbar zu machen werden generische forms erzeugt. Dazu müssen in den Joi-Schemas der API als metadaten der Name eines Components mitgeschickt werden. Beispiel:

.meta({ component: ‘textarea’ })

Im CMS werden diese Metadaten aus dem Schema der jeweiligen Entität ausgelesen und das React-Component ‘textarea’ ausgespielt. In der aktuellen Version des CMS sind folgende Komponenten verfügbar:

Name	Beschreibung
textinput	Einfaches Textfeld, kann für einzeilige Texte verwendet werden
url	Textfeld für Webseiten
email	Textfeld für E-Mail Adressen
phone	Textfeld für Telefonnummern
textarea	Mehrzeiliges, einfaches Textfeld
richtext	Mehrzeiliges Textfeld mit Format-Optionen, gibt HTML aus
checkbox	Einfache Checkbox zur Eingabe einzelner Werte
checkboxgroup	Mehrere Checkboxes zum toggle von mehreren Werten
switch	Switch-Component zur Eingabe von Boolean-Werten
select	Dropdown zur Auswahl eines Wertes aus einer Liste
tagchooser	Component für die Auswahl eines Tags (CultureMix-spezifisch)
multitagchooser	Component für die Auswahl mehrerer Tags (CultureMix-spezifisch)
date	Datepicker Component
relation	Ermöglicht das Hinzufügen einer anderen Entität aus der API
multirelation	Ermöglicht das Hinzugügen mehrerer Entitäten aus der API
languageSelect	Ein Select mit Sprachen als Optionene (CultureMix-spezifisch)
map	Zur Darstellung eines geografischen Punktes
upload	File-Upload Component
multiupload	Multi-File-Upload Component
multiform	Generisches Component um die Eingabe von Arrays mit Objekten zu ermöglichen
number	Component zur Eingabe von numerischen Werten
calendar	Kalender-Component für die Eingabe von Dates bei der Event-Entität (CultureMix-spezifisch)

Neue Komponenten können in /src/components/Forms/FormFieldFactory/index.js registriert werden. Zur Erstellung eines neuen Input-Components kann dieser Beispiel-Code als Grundlage verwendet werden:

```
import React from 'react';
import { useField } from 'formik';

// FormInputWrapper stellt das Label und den Erklärtext zu dem Input-Feld dar
import FormInputWrapper from '~/components/Forms/FormInputWrapper';

const ExampleInput = ({ name, label, options, readonly }) => {
 // Input mit Formik verbinden
 const [field, , helpers] = useField({ name });

 // Change-Handler ändert den Wert des Feldes in der Form
 const handleChange = evt => {
   helpers.setValue(evt.target.value);
 };

 return (
   <FormInputWrapper label={label} options={options} readonly={readonly}>
     <input type="text" onChange={handleChange} value={field.value} />
   </FormInputWrapper>
 );
};

export default ExampleInput;
```

Für das Handling der Eingabe-Forms wird die Library Formik verwendet. Daher können mit den Helper-Hooks (useField({ name })) eigene Komponenten für die Form gebaut werden. Das Beispiel oben zeigt ein einfaches Textfeld. Es ist außerdem mit dem FormInputWrapper gewrapped, welcher dafür verwendet wird den Titel und die Beschreibung des Inputs anzuzeigen.