fortiss

Digitale Rohstoffbörse
für nachhaltige Rohstoffe

Dokumentation SW-Architektur

fortiss GmbH – München, Dezember 2020

Zuwendungsempfänger

fortiss GmbH
An-Institut an der Technischen Universität München
Guerickestraße 25
80805 München

Kontakt: Rainer Diesch
Mail: diesch@fortiss.org
Tel.: +49 (0) 89 360 35 22 523
Fax: +49 (0) 89 360 35 22 50

Anlagen zum Abschlussbericht 2021

Verbundprojekt

Digitale Rohstoffbörse für nachhaltige Rohstoffe

Teilvorhaben

Entwicklung einer Internetplattform für die Förderung von Kommunikation, Handel und Verwertung von Rohstoffen durch das Zusammenbringen relevanter Zielgruppen.

Laufzeit

01.03.2018 - 28.02.2021

Inhalt

1. Gesamtarchitektur

Technischer Aufbau

Um eine Überführung in eine dauerhaft nutzbare Lösung zu ermöglichen wurden für den Prototyp ausschließlich zukunftssichere State-of-the-Art Technologie verwendet. Diese stellt einen langfristigen zukünftigen Betrieb sowie die eine uneingeschränkte Wartbarkeit und Weiterentwickelbarkeit der Plattform sicher. Nachfolgend werden die verwendeten Technologien kurz zusammengefasst:

Vue.js ist ein JavaScript-Webframework zur Erstellung von Client-Seitigen Applikationen. Als State-Of-The-Art Framework besitzt Vue.js eine große Entwicklerbasis, ist zukunftssicher und auf modulare Wiederverwendbarkeit ausgerichtet.

Die Serverseitige JavaScript basierte Plattform (node.js) sowie darauf aufbauendes Web-Framework (express) zur Entwicklung von State-Of-The-Art Web-Anwendungen. Skalierbarkeit, Performanz, Verbreitungsgrad und eine große Entwicklercommunity sind entscheidend für diese Technologieauswahl.

MongoDB ist eine NoSQL-Datenbank, die ideal in Kombination mit Node.js und Express einsetzbar ist. Die Datenbank ermöglicht insbesondere Speicherung und Einsatz projektspezifischer Operationen wie z.B. eine Umkreissuche auf geobasierten Daten.

Werden diese Technologien zusammen in einer Applikation als Basis verwendet, wird von einem **MEVN**-Stack gesprochen. Dieses Akronym steht für MongoDB, Express.js, VueJS und Node.js. Es handelt sich bei diesem Prototyp um eine Full-Stack-Application mit der gegebenen MEVN Architektur. Folgende Grundlagenkenntnisse/Softwarepakete sind notwendig für das Aufsetzen, Weiterentwickeln und Warten des Prototyps:

- Wissen in JavaScript (sowohl client- als auch server-seitig)
- Konzepte von REST und CRUD
- Installation folgender Software
  - Node.js
  - NVM
  - MongoDB
- Zusätzlich sind folgende Techniken von Vorteil, die im Projekt verwendet werden
  - git
  - .env-Umgebungen
  - eslint
  - prettier
  - babel
  - nodemon
  - vuex

Applikationsarchitektur

Abbildung 2 zeigt schematisch die Architektur der Applikation. Zudem ist hier das Zusammenspiel der unterschiedlichen Technologien mit dem schlussendlichen Anwender visualisiert. Der erste Aufruf der Webseite führt zur „Auslieferung" der Client-Applikation, die erst im Browser des jeweiligen Nutzers zur Ausführung kommt (Vue.js). Diese Applikation übernimmt den Aufbau sowie die Darstellung der Webseite im Browser des Plattform-Nutzers. Werden durch die Anwendung Daten benötigt, schickt die Applikation eine Anfrage an das sog. Backend, das zentral auf einem Applikationsserver bereitgestellt wird (node.js, express). Hierbei werden jedes Mal unterschiedliche Plausibilitäts- und Sicherheitsrelevante Prüfungen (z.B. insbesondere eine Nutzer-Authentifizierung) durchführt und bei bestehender Plausibilität die Daten aus der zentral bereitgestellten Datenbank (MongoDB) extrahiert. Das Backend sendet anschließend die angeforderten Daten an die Client-Applikation, die diese wiederum grafisch aufbereitet dem Benutzer über die Benutzeroberfläche präsentiert.

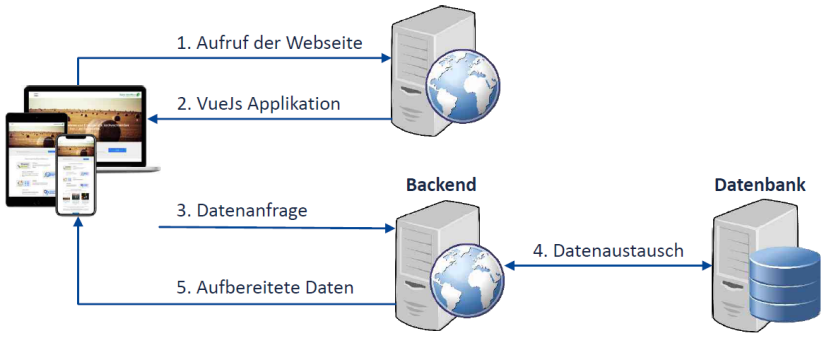


Abbildung 1: Technische Architektur

Server Deployment

Um alle Funktionen der Rohstoffbörse nutzen zu können, müssen für eine Produktivumgebung zwei Server und ein E-Mail-Postfach eingerichtet werden. Der Webserver ist für die Auslieferung der Client-Applikation sowie der statischen Bilder zuständig. Der node-Server ist für die Anfragen an das Backend zuständig. Das E-Mail-Postfach ist für die Versendung von Registrierungs-E-Mails sowie das Zurücksetzen von Passwörtern notwendig. Dieses kann in den .env-Variablen angegeben werden. Für die Entwicklung, kann das private Postfach verwendet werden.

Es können für den live-Betrieb beide Bestandteile auf einem physischen Server zusammen konfiguriert werden. Nachfolgend ein Beispiel für die Konfiguration eines nginx-Servers zum Betrieb beider Server auf einer physischen Maschine:

```
root /var/www/html/dist;
index index.html index.htm

location /api {
        proxy_pass http://127.0.0.1:3000;
}

location /public/ {
        root /var/www/api;
```

```
        try_files $uri @home;
}

location / {
        root /var/www/html/dist;
        try_files $uri /index.html;
}

location @home {
        root /var/www/html/dist;
        try_files $uri /index.html;
}
```

Dies zeigt die unterschiedlichen Anfragepfade der Applikation auf einen Server. Alle Anfragen zur API werden an den node-Server weitergeleitet sowie der Public-Pfad für dynamische Bilder, die beim Backend gespeichert werden. Werden die Server auf unterschiedlichen physischen Maschinen gespeichert, müssen die IP-Adressen ausgetauscht werden.

2. Client Applikation

Die Client-Applikation basiert auf der Standard-CLI (https://cli.vuejs.org/) in der Version 2.6.10. Während der Entwicklung der Applikation wurde die Vue-Version von 1 auf die neue Hauptversion 2 aktualisiert, um die Zukunftsfähigkeit, Wartbarkeit und Sicherheit der Applikation gewährleisten zu können. Die Applikation bedient sich unterschiedlicher Konzepte für das State-Management, die Ordnerstruktur (Code-Modularisierung), die Serververbindung, Routing sowie die Vorbereitung für zusätzlich einzubindende Bestandteile wie z.B. die Einbindung von Mehrsprachigkeit. Diese Konzepte werden im Folgenden erläutert.

Applikation aufsetzen

Wird die Applikation zum ersten Mal gestartet, muss ein Benutzer registriert werden. Nach Registrierung des Benutzers, muss die Rolle dieses in der MongoDB-Datenbank auf admin gesetzt werden. So erhält dieser vollen Zugriff auf alle Funktionen der Applikation.

State-Management

Jede Applikation hat einen Status, der durch die Ausprägung aller Variablen der Applikation bestimmt ist. Der Entwickler ist dafür verantwortlich, dass der Status einer Applikation nur in einer vorhersehbaren Weise verändert (mutiert) werden kann, um Fehler in der Applikation (nicht gewollter Status) zu vermeiden. Zudem sollte die Applikation erweiterbar bleiben, was einen großen Einfluss auf die Architektur und das Management des Status nach sich zieht. Vue.js-Anwendungen können auf verschiedene Weise mit einem solchen Status verwaltet werden. Für diesen Prototyp wurde Vuex (https://vuex.vuejs.org/) verwendet. Dies ist ein Zustandsverwaltungsmuster mit Regelungen, die eine Mutation des Status nur auf vorhersehbare Weise zulässt. Der Status wird global für alle Komponenten der Applikation gehalten. Zudem ist die Möglichkeit zur Integration in die offiziellen Devtools von Browsern gegeben, was ein Debugging sowie Export/Import von Zustands-Snapshots und weitere nützliche Funktionen bietet.

Abbildung 2 zeigt einen Überblick über die grobe Funktionsweise des Konzeptes. Eine Vue-Komponente sendet (dispatch) eine Nachricht an Vuex, welche diese eine Aktion (Action) auslöst. Diese Aktion führt unterschiedliche Funktionen, Berechnungen oder auch die Abfrage eines Backends (Serverbestandteil) durch und berechnet den neuen Status einer oder mehrerer Variablen im Status (State) der Applikation. Der neue Wert wird durch die Aktion festgelegt (commit) und somit eine Mutation (Mutation) ausgelöst, welche einerseits an die Devtools des Browsers gebunden ist und diese Anzeigt als auch andererseits den Status der Applikation schlussendlich ändert (mutate). Ändert sich der Status von Vuex werden automatisch alle Komponenten der Applikation neu erstellt (Render), die von der Änderung einer Variable im Status von Vuex betroffen ist. Der Vorteil besteht darin, dass eine Variable im Status von mehreren Komponenten verwendet werden kann. Ebenso wird bei einer Änderung des Status nicht die gesamte Seite der Applikation neu erstellt, sondern nur diese, welche von einer Änderung des Status betroffen sind.

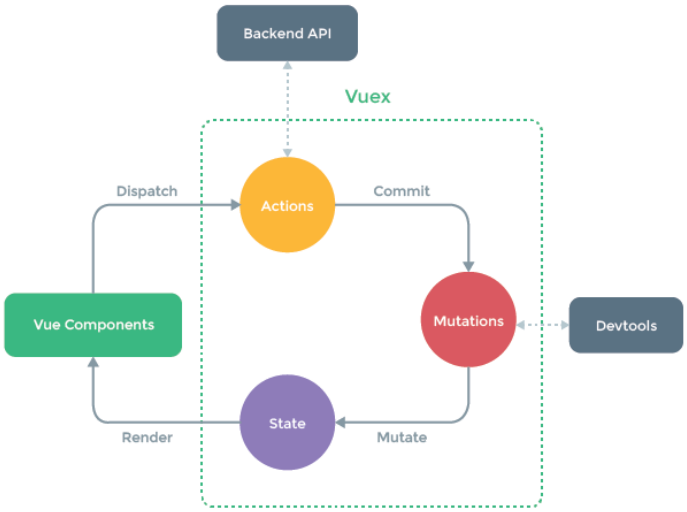


Abbildung 2: Vuex Überblick (https://vuex.vuejs.org/)

Abbildung 3 zeigt nun die Umsetzung dieses Konzeptes und einige Feinheiten im Programmcode einer Applikation. Der Store ist als globale Variable in einer Vue.js-Applikation als $store verfügbar. Zum Handling der einzelnen Bestandteile gibt es globale ACTIONS, MUTATIONS aber auch GETTERS. Zur Meldung an den Store wird die Funktion $store.dispatch() ausgeführt, die eine ACTION auslöst, welche asynchron ist und somit ein „Einfrieren" der Applikation auf Client-Seite verhindert. ACTIONS berechnen einen neuen Status und senden diesen mit Hilfe der commit()-Funktion an eine MUTATION, welche Synchron ausgeführt wird, da der STATE nur synchron geändert werden kann. Die MUTATION ändert nun ein oder mehrere Variablen des STATE. Daraufhin werden automatisch alle Komponenten neu erstellt, welche von der Änderung betroffen sind. Um den aktuellen Wert von Variablen des Status in einer Komponente nutzen zu können, werden GETTER verwendet.

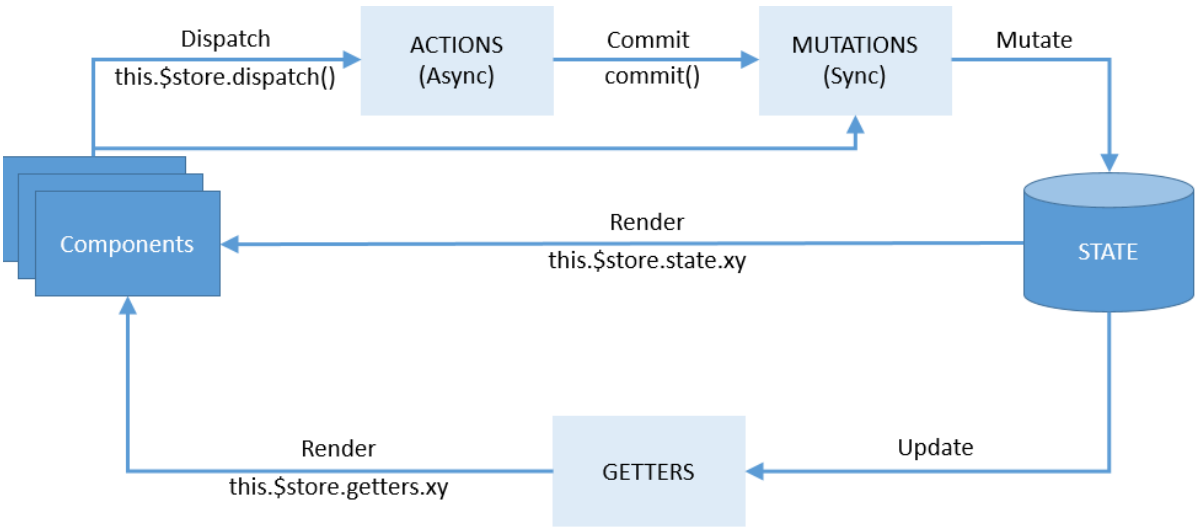


Abbildung 3: Vuex-Funktionen

Um die Funktionalität des Vuex-Stores zu gewährleisten müssen einige Regeln für die MUTATIONS beachtet werden. Im Folgenden werden die Grundlegenden beschrieben. Weitere Informationen sind unter https://vuex.vuejs.org/guide/mutations.html gegeben.

1. Folge den reaktiven Regeln (Vuex-Store ist reaktiv aufgebaut)
    a. Initialisiere alle gewünschten Variablen im Store von Beginn an.
    b. Verwende für eine Änderung immer ein neues Objekt. (nutze Funktionen wie *Vue.set(obj, ‚newProp', 123)* oder ersetze das Objekt mit *state.obj = { ...state.obj, newProp: 123 }*)
    c. MUTATION müssen Synchron sein, da ansonsten ein Debugging nicht mehr möglich ist.
    d. Verwende möglichst Konstanten für MUTATIONS. Dies erhöht die Wartbarkeit, Übersicht und ermöglicht es, die Vorteile von Entwicklertools wie linter für MUTATIONS zu vrwenden. Abbildung 4 zeigt ein Beispiel der Verwendung von MUTATIONS mit Konstanten.

```js
// mutation-types.js
export const SOME_MUTATION = 'SOME_MUTATION'
```

```js
// store.js
import Vuex from 'vuex'
import { SOME_MUTATION } from './mutation-types'

const store = new Vuex.Store({
  state: { ... },
  mutations: {
    // we can use the ES2015 computed property name feature
    // to use a constant as the function name
    [SOME_MUTATION] (state) {
      // mutate state
    }
  }
})
```

Abbildung 4: MUTATIONS unter Verwendung von Konstanten (https://vuex.vuejs.org/guide/mutations.html)

Der Store selbst ist ein großer Datenspeicher für die gesamte Applikation. Um diesen Datenspeicher leichter warten und erweitern zu können, ist dieser modular aufgebaut. Abbil-

dung 5 zeigt dies am Beispiel des Prototyps. Links oben im Bild befindet sich die Ordnerstruktur des Stores. Die index.js des Stores wird im ersten Schritt von der Applikation aufgerufen (siehe links unten). In dieser werden alle Module, die sich innerhalb des Store befinden geladen. Dieser Schritt 2 enthält nun den export aller enthaltenen Module des Stores (siehe store/modules/index.js rechts unten im Bild). Im letzten Schritt (siehe rechts oben) werden die einzelnen Module mit ihren dazugehörigen Funktionen sowie des Ausgangsstatus des Srote-Bereichs geladen. Um nun den Store zu erweitern benötigt es lediglich der Entwicklung eines neuen Moduls im Ordner ./store/modules, sowie eines entsprechenden Eintrags in der Datei ./store/modules/index.js.

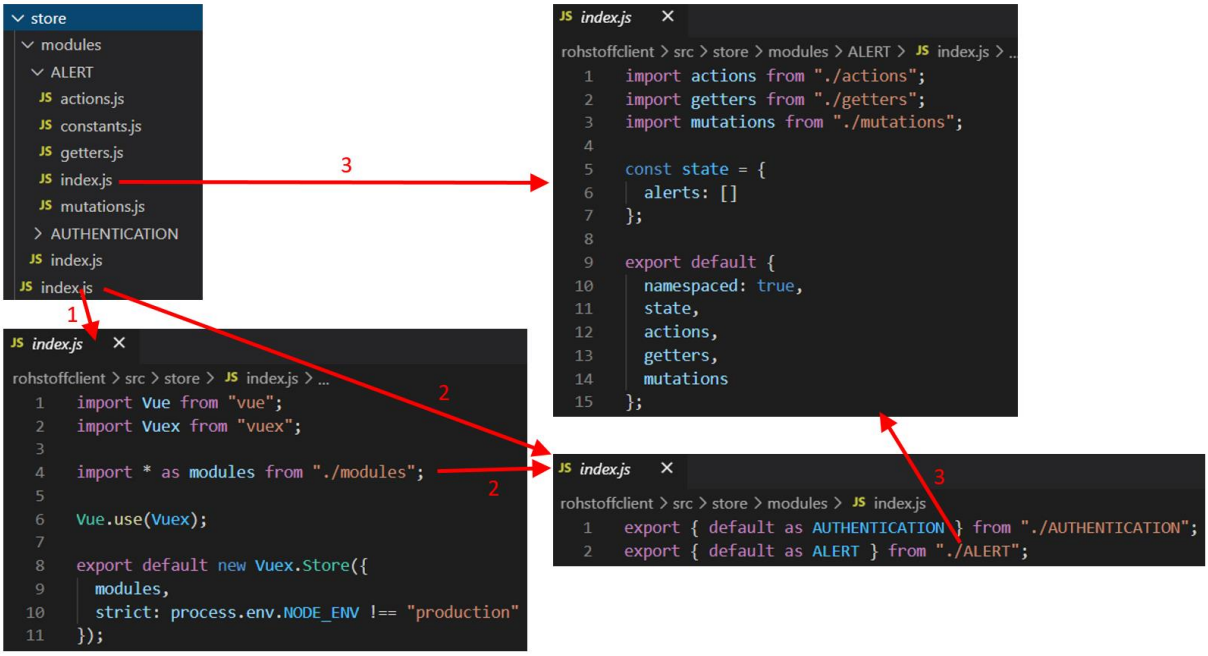


Abbildung 5: Vuex Modularisierung

Ordnerstruktur (Code-Modularisierung)

Ein nicht zu vernachlässigender Teil und Basis der Verständlichkeit, Wartbarkeit, Wiederverwendbarkeit und Erweiterbarkeit des Codes ist die Strukturierung. Der gesamte Code befindet sich im **src**-Ordner, welcher folgendermaßen unterteilt ist.

- **api** beinhaltet alle globalen backend-Calls, die übergreifend für alle Module der Applikation zur Verfügung stehen. Hierzu gehören z.B. login, logout, register, E-Mail-Verification, usw. Der detaillierte Aufbau der Serververbindung wird in Kapitel Serververbindung beschrieben.
- **assets** beinhalten alle statischen Inhalte des Clients wie z.B. Bilder oder Kartenmaterial für die Applikation
- **components** enthalten globale Komponenten, die von unterschiedlichen Modulen genutzt werden können. Komponenten unterscheiden sich insofern von Modulen, dass diese keinerlei Aufrufe zum Backend enthalten und als „Dumme" Komponenten nur zur Darstellung von Daten dienen. Zudem sind diese global für alle Module verfügbar.

- **design** konfiguriert den globalen Vue.js style wie z.B. das Laden der material-icons, das Setzen der Standardfarben, usw.
- **modules** sind alle komplexen Bestandteile der Applikation. Module besitzen „private" Komponenten, eine eigene Backend-api sowie einen eigenen store. Nicht alle Unterkategorien sind zwingend für ein Modul notwendig. Ein Beispiel ist das Produktmodul (Products). Dieses beinhaltet eine Produktliste zur Übersicht (ProductList.vue) mit eigener API (_api), um alle Produkte aus dem Backend auslesen zu können. Wird ein Produkt eingesehen, wird dies mit Hilfe der detaillierten Produktinformationen angezeigt (Product.vue). Innerhalb des Prduktes gibt es eine Preishistorie mit einer relativ komplizierten Chart-Darstellung, welche durch eine Komponente (_components) Chart.vue umgesetzt wurde. In diesem Stil können so komplett eigenständige Module entwickelt werden, welche keinerlei Abhängigkeiten zu anderen Code-Bestandteilen aufweist und in jeder anderen Applikation genauso verwendet werden können. Eine Besonderheit besteht bei der Registrierung eines Moduls für den Store. Ein Store ist global für alle Module verfügbar. Deshalb muss dieser innerhalb des Moduls besonders eingefügt werden. Abbildung 6 zeigt, wie ein Modul-Store zum globalen hinzugefügt wird.

```
created() {
  const STORE_KEY = '$_chat';
  // eslint-disable-next-line no-underscore-dangle
  if (!(STORE_KEY in this.$store._modules.root._children)) {
    this.$store.registerModule(STORE_KEY, store);
  }
},
```

Abbildung 6: Hinzufügen eines Modul-Stores zum globalen Store

- **router** ist für das Routing der Client-Applikation zuständig und verbindet hier die Links der Seite mit den entsprechenden Anzeigemodulen (views). Ebenso werden die Berechtigungen, welche in Kapitel Routing näher erläutert werden, überprüft.
- **store** ist der Ablageort zur Initialisierung des globalen Vuex-Speichers (siehe Kapitel State-Management) und der Definition globaler Store-Bestandteile, die von jedem Modul verwendet werden können und zu keinem Einzelmodul einzuordnen sind, wie z.B. die Authentifizierung der Benutzer und das Halten der Benutzereigenschaften.
- **util** enthält globale Funktionen für alle Module und Komponenten. filters.js sind Vue-Funktionen zur Transformation von Daten wie z.B. Zeit. form-validation.js ist die Eingabeüberprüfung. get-page-title.js modifiziert die standard-Anzeige von Vue zur Anzeige im Browserfenster. request.js ist die globale Konfiguration und Ausführung von Backend-Calls.
- **views** beinhalten die Seiten (pages) der Applikation sowie deren Layout. Jede View ist in einem Basis-Layout (BaseLayout.vue) eingebettet und beinhaltet deshalb bereits die Navigation (TopNavigation), eine Alert-Anzeige sowie einen Footer. Somit können neue Seiten einfach durch Hinzufügen einer neuen View sowie eines entsprechenden Eintrags im router erweitert werden.

Serververbindung

Die Serververbindung ist mit dem Framework axios umgesetzt worden. Für eine möglichst einfache Verwendung der API, wurde eine globale Konfiguration in ./utils/request.js umgesetzt. Hier wird die Basis-Konstante request konfiguriert indem die Basis-URL sowie ein timeout für die Verbindung festgelegt wird. Die Basis-URL hingegen wird aus den .env-Variablen geladen um bei einem Serverumzug den Code nicht verändern zu müssen. Um nicht für jeden Backend-Aufruf der Applikation alle Eventualitäten prüfen zu müssen, werden interceptors verwendet. Ein interceptor für jeden request (request.interceptors.request.use()) wurde definiert, um für jeden ausgehenden Request den Authentication-Token (sollte dieser vorhanden sein) dem Server mitzusenden. Jede Serverantwort wird daraufhin durch einen response-interceptor auf Fehler überprüft. Speziell, ob der Benutzer Authorisiert ist, den entsprechenden Request zu machen oder ob die aktuelle Sitzung beim Server abgelaufen ist. Zudem werden alle Server-Fehlermeldungen dem Benutzer über die Global vorhandene Alert-Message angezeigt. Diese Überprüfungen müssen nun nicht mehr bei jedem request durchgeführt werden.

Abbildung 7 zeigt nun, wie ein Request verwendet werden kann. Auf der linken Seite sind Beispiele für die Verwendung in einer Komponente mit einem POST, DELETE und GET request. Hierzu muss lediglich die vorhin beschriebene request.js importiert werden. Innerhalb eines Moduls können die Funktionen nun direkt importiert und asynchron verwendet werden (siehe rechte Seite des Bildes).

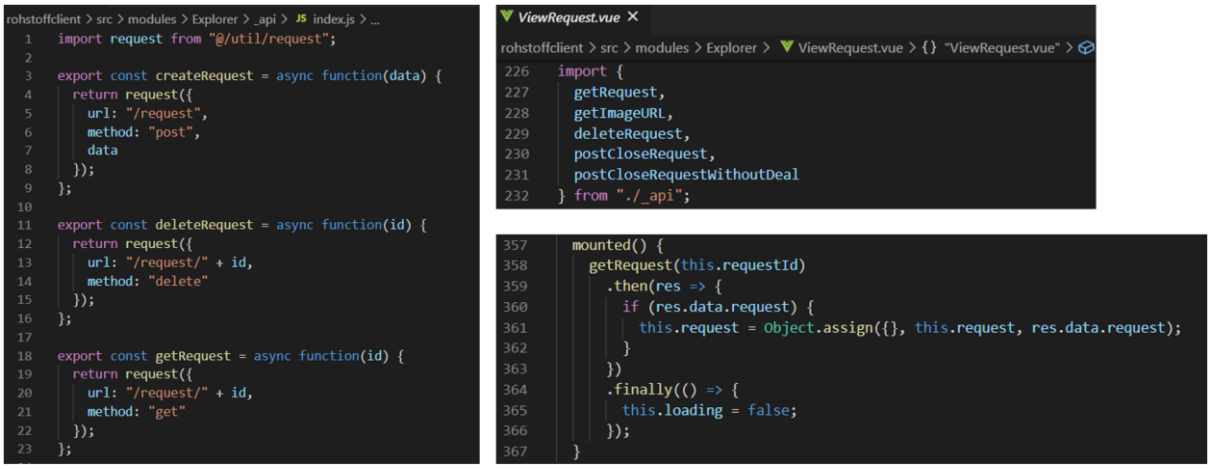


Abbildung 7: Verwendung der Backend-API

Routing

Routing bedient sich dem Modul „vue-router“. Dies ermöglicht das Verwenden unterschiedlicher Routen für die Applikation sowie deren Konfiguration. Für eine leichte Verwendung der Routen und die leichte Einbindung in die Bestehende Seite sind unterschiedlichste Konfigurationen möglich und bereits in die Applikation und die Module integriert. Routen können in Haupt- und Kind-Routen unterteilt werden. Dies äußert sich in der Applikation dadurch, dass diese in der Navigation als Haupt- oder Unterpunkt der Auflistung gezeigt

werden. Zudem wird der Pfad der Route immer relativ in der Konfiguration angegeben. Abbildung 8 zeigt ein Beispiel für die Konfiguration des adminRouters, welches alle Routen für den Admin-Bereich beinhaltet. Die Route /admin/products verweist auf den Kindknoten (children) mit dem Pfad products. Dabei kann bereits gesehen werden, dass auch Routen modular aufgebaut sind. Alle Routen für den admin befinden sich in /router/modules/admin.js. Diese werden dann in der Hauptkonfiguration geladen und nur noch eingebettet. In der Hauptkonfiguration (/router/index.js) befinden sich globale Seiten wie die Startseite, Fehler, Impressum, Datenschutz...).

Abbildung 8: Router Example

Jede route ist durch ein json-Objekt spezifiziert. Folgende Konfigurationsmöglichkeiten sowie deren Auswertungen sind möglich:

- **name**: ‚router-name'; Name der Route
- **path**: „/"; Pfad der Applikation zur Route
- **component**: BaseLayout; Komponente, die für den Pfad geladen wird (view)
- **redirect**: „/path"; Wenn gesetzt, wird sofort zu dieser Route weitergeleitet
- **hidden**: true; Wird dies gesetzt, ist das Element in der Navigation nicht sichtbar
- **props**: true; Wenn gesetzt, sind Übergabe-properties im Pfad erlaubt
- **meta**: {
    - **title**: „NAME"; Setzt den Namen der Seite im Browser-Tab

- **icon**: „svg-name“; Setzt das Item in der Navigation (Material-icons erlaubt)
- **notificationtype**: „TYPE“; Fragt den Server nach Notifications und zeigt die Anzahl dieser in der Navigation
- **roles**: [„admin“, „editor“, „regular“, „guest“]; Setzt die Berechtigung für die jeweilige Route, die geladen wird. Sollte hier nur guest angegeben, wird die Route nur dann in der Navigation angezeigt, wenn der Benutzer nicht eingeloggt ist (z.B. Login)

Zur Umsetzung der Navigation sowie der Berechtigungsprüfung wurde ein permissionRouting.js entwickelt. Dieser Code-Bestandteil überprüft vor jedem Betreten einer Route die Berechtigung des angemeldeten Benutzers und gleicht diese mit der Spezifikation der Routen ab. Zudem wird überprüft, ob sich die Rolle eines Benutzers in der Zwischenzeit verändert hat (z.B. von guest zu regular) und stößt daraufhin das Laden aller Routen für die neue Rolle im Vuex-Store an, um daraufhin die Navigation zu aktualisieren.

Mögliche Erweiterungen

Erweiterungen sind durch diesen Aufbau sehr einfach möglich. Alle Module können von jeder anderen Applikation durch die Trennung vom restlichen Code verwendet werden. Seiten können durch Einbindung einer Route und einer View einfach eingebunden werden. Änderungen von Server oder anderen Backend-Parametern sind ohne Code-Anpassungen möglich. Design und API lassen sich einfach austauschen. Eine mögliche Erweiterung, die noch nicht vorgesehen ist, wäre ein Multilanguage-Support. Dieser würde einfach umgesetzt werden können. Hierzu sollte ein language-Ordner mit den Sprachdateien angelegt werden. Das Modul vue-i18n kann dann global für die Vue-Applikation registriert und verwendet werden. Hierzu müssen alle Zeichenfolgen, die in der Applikation verwendet werden in die Sprachdateien ausgelagert werden.

3. Server Applikation

Das Backend des Prototyps besteht aus einer Node.js-Applikation. Das Framework Express, ein einfaches und flexibles Node.js-Framework für Webanwendungen, wurde für eine schnelle Entwicklung verwendet. Zusätzlich wurden folgende Frameworks genutzt:

- **bcrypt** zur Verschlüsselung von Daten.
- **cors** zur Verwendung von cross-origin resource sharing in der Applikation.
- **helmet** für Standard Sicherheitseinstellungen.
- **jsonwebtoken** zur Erstellung und Prüfung von Session-Tokens.
- **mongoose** zur Verbindung mit der MongoDB Datenbak.
- **morgan** zum logging von Serveranfragen.
- **multer** für das Behandeln von multipart/form-data anfragen (upload images).
- **nodemail** für das Versenden von E-Mails (Verifizierung, Password reset)
- **passport** zur Authentifizierung der Anfragen.

Die grundlegenden Einstellungen für die Applikation wird in der datei ./src/index.js definiert. Hier wird zuerst definiert, ob der Server in einer produktiv- oder in der development-Umgebung gestartet wird. Unterschied hierbei ist, dass bei der Produktivumgebung unterschiedliche logging-optionen nicht zur Verfügung stehen, die dev-Tools im Browser deaktiviert werden sowie keine stack-traces an clients gesendet werden. Zudem wird der öffentliche Ordner für die Auslieferung von Bildern definiert, die Datenbankverbindung etabliert, der Router mit einer default-Schnittstelle (in diesem Fall /api/v1) definiert und Standardbehandlungen für Fehlermeldungen (z.B. route nicht definiert) festgelegt. Schlussendlich wird der Server auf dem definierten Port in der .env-Datei gestartet.

Ordnerstruktur (Code-Modularisierung)

Abbildung 9 zeigt den Überblick über die Code-Struktur des Backends. Die Rechtecke repräsentieren hier nicht nur den Ablauf, sondern auch die Ordnerstruktur. Anfragen an die API werden an die router-Schnittstellen gesendet (siehe Kapitel Routing). Nach der Prüfung der Eingangsvariablen sowie der Authentifizierung, werden die Daten an den jeweiligen controller gesendet. Dieser ist nun dafür zuständig, Daten aus unterschiedlichen services zusammenzustellen und die Antwort für den Aufrufenden vorzubereiten. Auch Fehlermeldungen werden hier abgefangen und benutzerfreundlich aufbereitet. Zwischen dem controller und den services ist eine gestrichelte Linie eingezeichnet, denn alles ab diesem Zeitpunkt kann jederzeit ausgetauscht werden. Da der controller hauptsächlich Aufrufe von services beinhaltet, um die Eingabedaten weiterzureichen, Daten zu ändern oder auch Daten anzufordern und sonst alle weitere Logik für die Zusammenstellung des Ergebnisses beinhaltet, können die services selbst, jederzeit und unabhängig von den controllern ausgetauscht werden. Die services beinhalten nun entweder die Verbindung zur Datenbank unter der Verwendung von models oder auch aufrufe zu externen services. Ein Beispiel für einen externen service innerhalb des Prototyps ist das Senden von E-Mails. Zur Erweiterung der API muss lediglich eine neue route eingebunden werden, sowie der zugehörige controller.

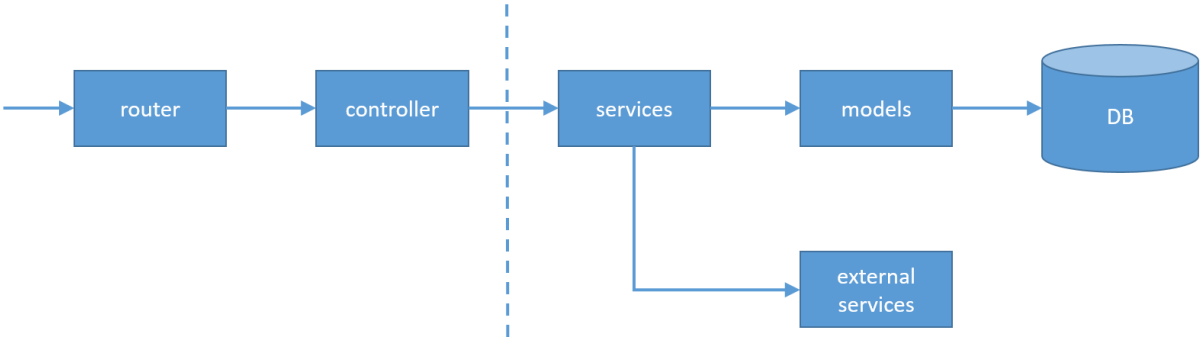
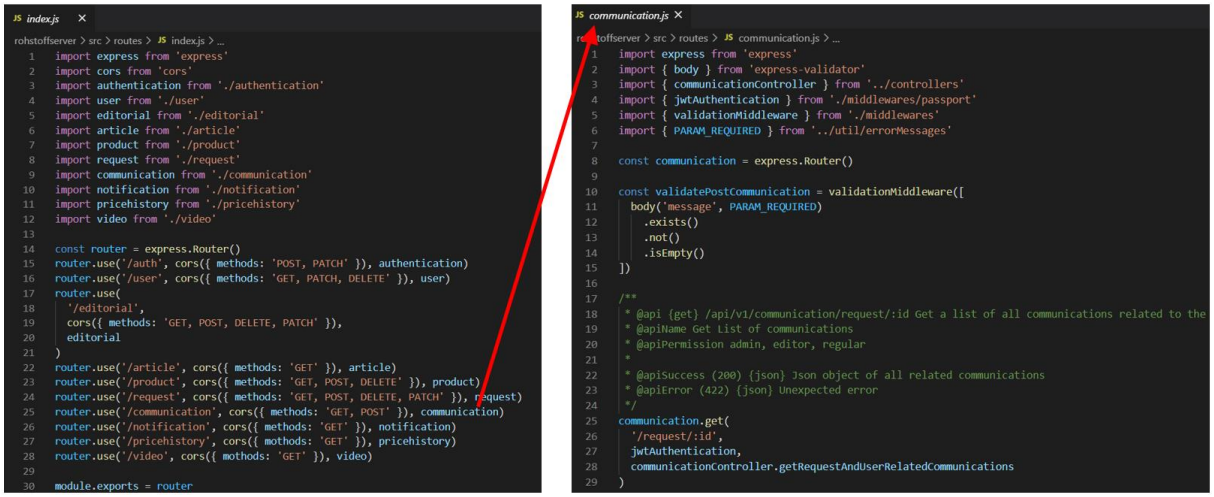


Abbildung 9: Code-Struktur

Routing

Das Routing inkludiert alle Schnittstellen, die zum Aufruf von außerhalb als API zur Verfügung stehen. Es teilt sich auf in direkte Schnittstellen (routes) sowie zugehörige middlewares, die die unterschiedlichen Anfragen an den Server auf unterschiedliche Aspekte hin überprüfen. Hierzu gehört z.B. die Authentifizierung der Benutzer, deren Rollenzuordnung, die Validierung der Parameter (siehe Kapitel Server-Side Validation und Server-Sicherheit) und das prozessieren des Bild-Uploads. Abbildung 10 zeigt, wie das routing abläuft. Auf der linken Seite befindet sich ./routes/index.js in dem jedes Routing-Modul in den router eingebunden wird. Cors() sorgt hierbei für die Spezifikation der erlaubten http-Anfragefunktionen (GET,POST,DELETE,PATCH,HEAD,PUT,TRACE,OPTIONS,CONNECT). Zur Einbindung neuer API-Schnittstellen, müssen diese in der index-Datei (für neue Module) oder innerhalb eines Moduls (für bestehende Module) eingefügt werden. Auf der rechten Seite des Bildes ist ein Beispiel für eine API-Schnittstelle connection aufgeführt. Die get-Schnittstelle für www.rohstoffbörse.com/api/v1/request/:id wird definiert. Die middleware jwtAuthentication überprüft, ob der Benutzer, der die Anfrage gestellt hat auch registriert und angemeldet ist. Ist dies der Fall, wird die Anfrage mit den validierten Input-Parametern (in diesem Fall die ID) an den zuständigen Controller zur Ermittlung aller Kommunikationen eines Benutzers (derjenige, der die Anfrage gestellt hat) weitergegeben.



```
index.js
rohstoffserver > src > routes > index.js > ...
import express from 'express'
import cors from 'cors'
import authentication from './authentication'
import user from './user'
import editorial from './editorial'
import article from './article'
import product from './product'
import request from './request'
import communication from './communication'
import notification from './notification'
import pricehistory from './pricehistory'
import video from './video'

const router = express.Router()
router.use('/auth', cors({ methods: 'POST, PATCH' }), authentication)
router.use('/user', cors({ methods: 'GET, PATCH, DELETE' }), user)
router.use(
  '/editorial',
  cors({ methods: 'GET, POST, DELETE, PATCH' }),
  editorial
)
router.use('/article', cors({ methods: 'GET' }), article)
router.use('/product', cors({ methods: 'GET, POST, DELETE' }), product)
router.use('/request', cors({ methods: 'GET, POST, DELETE, PATCH' }), request)
router.use('/communication', cors({ methods: 'GET, POST' }), communication)
router.use('/notification', cors({ methods: 'GET' }), notification)
router.use('/pricehistory', cors({ methods: 'GET' }), pricehistory)
router.use('/video', cors({ methods: 'GET' }), video)

module.exports = router
```

```
communication.js
rohstoffserver > src > routes > communication.js > ...
import express from 'express'
import { body } from 'express-validator'
import { communicationController } from '../controllers'
import { jwtAuthentication } from './middlewares/passport'
import { validationMiddleware } from './middlewares'
import { PARAM_REQUIRED } from '../util/errorMessages'

const communication = express.Router()

const validatePostCommunication = validationMiddleware([
  body('message', PARAM_REQUIRED)
    .exists()
    .not()
    .isEmpty()
])

/**
 * @api {get} /api/v1/communication/request/:id Get a list of all communications related to the
 * @apiName Get List of communications
 * @apiPermission admin, editor, regular
 *
 * @apiSuccess (200) {json} Json object of all related communications
 * @apiError (422) {json} Unexpected error
 */
communication.get(
  '/request/:id',
  jwtAuthentication,
  communicationController.getRequestAndUserRelatedCommunications
)
```

Abbildung 10: Beispiel Routing

Server-Side Validation und Server-Sicherheit

Für die Serversicherheit ist die Validierung der Eingangsparameter von entscheidender Bedeutung. Hierfür wurde eine validation-middleware entwickelt (./routes/middlewares/paramValidation.js). Diese basiert auf der Standard-Bibliothek express-validator. Alle festgelegten Validierungsregeln werden hier überprüft. Erst bei Erfüllung aller Regeln wird die Anfrage an die nächste Instanz der middleware-Kette einer API-Definition weitergeleitet. Abbildung 11 zeigt die Verwendung der validation-Middleware. Im ersten Schritt werden Validierungsregeln definiert sowie die Felder, welche für eine bestimmte Schnittstelle zu übergeben sind. Alle möglichen Validierungsregeln sind unter https://express-validator.github.io/docs/ einsehbar. Die Validierung ist so definiert, dass alle zusätzliche Parameter (wie z.B. untern ein Parameter „birthday“) herausgefiltert und gelöscht werden. Hierdurch werden wirklich nur diejenigen Parameter an den Programmcode im controller weitergegeben, welche validiert und definiert wurden. Sollte ein Parameter falsch übergeben werden, wird eine Fehlermeldung zu jedem dieser Parameter erstellt und sofort (ohne Ausführung weiterer Code-Bestandteile dieses API-Endpunkts) zurückgesendet. Somit wird eine sichere Verwendung nicht vertrauenswürdiger Eingabeparameter gewährleistet.

```
30  const validateProfileParams = validationMiddleware([
31    body('nickname', PARAM_REQUIRED).exists().not().isEmpty(),
32    body('name', PARAM_REQUIRED).exists().not().isEmpty(),
33    body('surname', PARAM_REQUIRED).exists().not().isEmpty(),
34    body('origin').optional()
35  ])
```

```
120  authentication.patch(
121    '/profile',
122    jwtAuthentication,
123    validateProfileParams,
124    authenticationController.changeProfile
125  )
```

Abbildung 11: Verwendung der Validation-Middleware

Zur Sicherstellung der Authentizität der Inhalte wird jeder Endpunkt durch eine Authentication-Middleware basierend auf dem passport-Framework verwendet. Momentan sind drei Authentifizierungsmechanismen implementiert. localAuthentication wird für den Login zur Webseite (Ausstellung eines Berechtigungs-Tokens) genutzt. jwtAuthentication wird zur Authentifizierung eines Benutzers verwendet, der bereits eingeloggt ist. roleAuthentication überprüft die Rolle des Benutzers, der die Anfrage an die API vornimmt. Ein Beispiel für die Verwendung wird ebenfalls in Abbildung 11 gezeigt. Nach Ausführung und korrekter Authentifizierung des Benutzers, stehen die Informationen dieses Benutzers für die weitere Verwendung im req.user-Objekt zur Verfügung. Das passport-Framework ermöglicht die zukünftige Einbindung unterschiedlicher Authentifizierungsmechanismen wie z.B. Google-Login, Facebook-Login, usw.