



# PRAXISSEMESTER

## BERICHT

*Zeitraum: 05.05.2025 – 29.09.2025*

*Einsatzort: B-ite GmbH*

*Tätigkeit: Full Stack Entwickler*

Bekolli, Alton  
Bekoal01@thu.de  
Matrikel-Nr: 6003337  
Studiengang: Informatik

# Inhaltsverzeichnis

<b>1. Einleitung .....</b>	<b>1</b>
1.1 Das Unternehmen B-ite GmbH .....	1
1.2 Motivation, Erwartung und Rahmenbedingung des Praxissemesters .....	2
<b>2. Fachliche Informationen .....</b>	<b>3</b>
2.1 Tech Stack .....	3
<b>3. Praktische Tätigkeiten im Praxissemester Projekt 1 .....</b>	<b>4</b>
3.1 Erste Schritte mit der B-ite Jobs API .....	4
3.2 Build Prozess .....	6
3.3 Vertiefung einer Standard API .....	7
3.4 Ergebnis .....	9
<b>4. Praktische Tätigkeiten im Praxissemester Projekt 2 .....</b>	<b>10</b>
4.1 Erste Schritte in den Templates .....	11
4.2 Vertiefung der Templates .....	12
4.3 Das Hochladen eines Templates .....	13
4.4 Ergebnis .....	15
<b>5. Praktische Tätigkeit im Praxissemester Projekt 3 .....</b>	<b>16</b>
5.1 Fachliche Analyse der Datenbank .....	16
5.2 Das Backend des Cookiebanners .....	17
5.3 Web User Interface .....	18
5.3.1 Beschreibung der Struktur .....	19
5.3.2 Admin Tool .....	20
5.3.3 Einbinden eines Cookie Banners .....	23
5.3.4 Ergebnisse .....	23
<b>6. Kleine Projekte .....</b>	<b>25</b>
<b>7. Fazit .....</b>	<b>26</b>
<b>8. Abbildungsverzeichnis .....</b>	<b>27</b>
<b>9. Quellenverzeichnis .....</b>	<b>28</b>

## 1.Einleitung

### 1.1 Das Unternehmen B-ite GmbH

Die B-ite GmbH ist ein mittelständisches Softwareunternehmen mit Sitz in Ulm. Seit ihrer Gründung hat sich das Unternehmen auf digitalen Lösungen im Bereich Recruiting und Bewerbermanagement spezialisiert und zählt heute zu den etablierten Anbietern im HR-Tech-Sektor. Das Hauptprodukt von B-ite ist der B-ite Bewerbemanager, eine moderne cloudbasierte Softwarelösung. Mit diesem Tool können Unternehmen Stellenausschreibungen veröffentlichen, Bewerbungen zentral verwalten, Bewerberkommunikation steuern und Auswertungen zum gesamten Recruiting Prozess vornehmen. Ergänzt wird das Angebot durch individuell gestaltete Job Posting Vorlagen, die an das Corporate Design der Kunden angepasst sind, sowie durch die Job-API, über die Stellenanzeigen direkt und nahtlos in bestehende Unternehmenswebseiten eingebunden werden können. Die Vorlagen die von dem B-ite Team erstellt werden, werden nicht nur für Unternehmenswebseiten eingesetzt, sondern auch für externe Plattformen wie Indeed optimiert, sodass eine maximale Reichweite und Sichtbarkeit erzielt wird.

Ein besonderes Merkmal der B-ite Lösung ist die enge Verbindung zu aktuellen Recruiting Trends. So können sich Bewerbende auch direkt über WhatsApp bewerben, was den Bewerbungsprozess vereinfacht und die Hürden für Interessierte deutlich senkt. Darüber hinaus legt B-ite großen Wert auf Suchmaschinenoptimierung. Die Stellenanzeigen werden so gestaltet und technisch aufbereitet, dass sie von Google und andere Suchmaschinen bestmöglich gefunden werden und weit oben in den Suchergebnissen platziert werden.

Viele zusätzliche Entwicklungen erweitern das Portfolio von B-ite. Ein Beispiel hierfür ist das Pitchman Team, das sich mit Innovationen und Ansätzen rund um Personalmarketing und Bewerberansprache beschäftigt. Damit reagiert das Unternehmen flexibel auf neue Anforderungen des Marktes und unterstützt seine Kunden auch über die reine Softwarelösung hinaus.

Ein Zentrales Ziel von B-ite ist es, Unternehmen aller Größenordnungen bei der Personalgewinnung, um im gesamten Recruiting Prozess zu unterstützen. Das zeigt sich auch in der Vielfalt der Kundschaft: Zu den Kunden zählen Städte und Kommunen, Hochschulen, Pflege und Sozialeinrichtungen sowie zahlreiche Industrie und Privatunternehmen. Diese breite Ausrichtung macht die Softwarelösung besonders flexibel und praxistauglich.

Das Unternehmen umfasst rund 113 Mitarbeitende, die in verschiedenen Bereichen gegliedert sind: Buchhaltung, Entwicklung, ein eigenes Indeed Team, ein Marketing,

Operation, Personal, Vertrieb, der Verwaltung, dem Personal und dem spanischen Team. Diese Struktur ermöglicht es, Kundenprojekte von der ersten Beratung über die technische Umsetzung bis hin zu den langfristigen Betreuungen effizient abzuwickeln. Neben den Produkten und der Organisation legt B-ite besonderen Wert auf Datenschutz und Sicherheit. Alle Lösungen werden in Deutschland entwickelt und betrieben. Das Hosting erfolgt in einem nach ISO-27001 zertifizierten Tier3 Rechenzentrum, wodurch die Einhaltung der Datenschutzgrundversorgung gewährleistet wird. Damit bietet B-ite den Kunden nicht nur funktionale, sondern auch rechtssichere Zusammenarbeit. Insgesamt steht die B-ite GmbH damit für ein Unternehmen, das moderne Softwareentwicklung mit praxisnahen Recruiting Lösungen verbindet und seine Kunden bei allen Herausforderungen im Personalmanagement zuverlässig unterstützt.

## **1.2 Motivation, Erwartung und Rahmenbedingung des Praxissemester**

Für mein Praxissemester war es mir wichtig, ein Unternehmen zu finden, das mir nicht nur theoretisches Wissen vermittelt, sondern auch vor allem Praktische Einblicke in die moderne Softwareentwicklung gibt. Die Wahl fiel auf die B-ite GmbH, wo ich zuvor schon als Werkstudent tätig war. Während dieser Zeit konnte ich als erste Erfahrung in der Webentwicklung sammeln und mir einen Eindruck von den internen Abläufen verschaffen. Gemeinsam mit meinem Vorgesetzten wurde im Vorfeld abgestimmt, dass ich im Rahmen des Praxissemester umfangreiche Aufgaben übernehmen darf, als es in meiner Rolle als Werkstudent möglich war.

Genau diese Aussicht auf mehr Verantwortung und komplexere Tätigkeiten hat meine Entscheidung bestärkt, das Praktikum bei B-ite zu absolvieren.

Meine Erwartungen an das Praxissemester waren dementsprechend vielfältig. Zum einen wollte ich ein tieferes Verständnis für die Entwicklung von Webanwendungen und APIs erlangen, zum anderen Einblicke in die Arbeitsprozesse eines Softwareunternehmens gewinnen. Dazu zählt für mich nicht nur die rein technischen Aufgaben, sondern auch die Zusammenarbeit im Team, der Umgang mit Tickets Systeme und die Abläufe bei Code Reviews. Ein weiteres Ziel war es, mein Wissen in Frameworks wie Preact / React und Programmiersprachen wie Typescript und Go praktisch anwenden zu können.

Rückblickend lässt sich festhalten, dass sich diese Erwartung im Verlauf des Praxissemester erfüllt und teilweise sogar übertroffen wurde.

Die Rahmenbedingungen meines Praxissemester waren klar definiert. Der Zeitraum erstreckte sich zunächst vom 05.05.2025 bis zum 26.09.2025, der wurde jedoch um ein Tag verlängert, da ich wegen den Feiertagen und meine zwei Urlaubstage nicht auf die 100 Tage gekommen wäre. Dadurch ging mein Praxissemester vom 05.05.2025 bis zum 29.09.2025. Die Reguläre Arbeitszeit betrug dabei mindestens sechs Stunden täglich, wobei ich jede Woche zwischen 40 bis 43 Stunden die Woche gearbeitet habe. Die Tätigkeit wurde mit einem Stundenlohn von 14 Euro vergütet. Inhaltlich war ich in

verschiedenen Aufgabenbereichen eingebunden, die sowohl die Bearbeitung von Tickets im Bereich API Integration und Templates als auch mein eigenes Projekt, nämlich das Entwickeln des Backend für den Cookie Banner inklusive ein Admin Tool umfasste. Die detaillierte Beschreibung dieser Tätigkeiten erfolgt in den folgenden Kapiteln.

## 2.Fachliche Informationen

### 2.1 Tech Stack

Für die Tägliche Arbeit bei B-ite kommt eine Vielzahl an modernen Tools und Technologien zum Einsatz. Diese decken sowohl den Bereich der Infrastruktur als auch die Frontend und backend Entwicklung sowie das Build und Paketmanagement. Im Folgenden möchte ich erläutern, welche Werkzeuge wir verwenden und wieso dies für die Entwicklung sinnvoll sind.

Ein Zentrales Element ist die Versionsverwaltung mit Git. Intern wird dafür eine eigene GitLab Instanz betrieben, die unter dem Namen BITEhub bekannt ist. Dafür lassen sich nicht nur Änderungen am Code nachverfolgen, sondern auch Tickets verwalten und Merge Request anlegen. So wird sichergestellt, dass alle Änderungen überprüft werden. Bevor sie in die produktive Version übernommen werden. Zusätzlich nutzen wir Gitea als weiteres Tool für die Vorlageverwaltung und Tickets. Beide Systeme geben einen guten Überblick über aktuelle Aufgaben und erleichtern die Zusammenarbeit im Team. Auch das Betriebssystem spielt eine Rolle. Ein Teil des Teams arbeitet mit Arch Linux, einer sehr flexiblen Linux Distribution. Arch bietet den Vorteil, dass es stets auf der neuesten Stand gehalten wird ("Rolling Release") und sich individuell an die Bedürfnisse der Entwickler anpassen lässt. Dadurch können wir ein leichtgewichtiges und schnelles System nutzen. Das optimal auf die Arbeit mit modernen Frameworks abgestimmt ist.

Für die Frontend Entwicklung setzen wir auf moderne Webstandards wie HTML, CSS /SCSS und JavaScript. Insbesondere SCSS erlaubt es, Styles strukturiert und wiedererwendbar zu schreiben, was bei der Vielzahl an Kundenvorlagen unverzichtbar ist.

Als Framework kommt bei mir häufig Preact in Verbindungen mit TypeScript zum Einsatz. Preact ist eine besondere leichtgewichtige Alternative zu React, was vor allem dann sinnvoll ist, wenn Performance und Ladezeiten eine große Rolle spielen, zum Beispiel auf Kundenseiten mit vielen Stellenausschreibungen. Durch TypeScript haben wir zusätzlich den Vorteil einer statischen Typisierung, die Fehler frühzeitig erkennt und die Codequalität verbessert. Daneben habe ich bei komplexeren Projekten wie dem Cookiebanner auch React mit Elastic UI eingesetzt, was ein modernes und professionelles User Interface ermöglicht. Auf der Backend-Seite arbeiten wir oftmals mit Golang in Kombination mit

PostgreSQL als Datenbank. Go ist besonders gut geeignet für skalierbare Webanwendungen und zeichnet sich durch seine hohe Performance und Einfachheit aus. Gleichzeitig bietet es eine hervorragende Integration von Concurrency, was parallele Prozesse erleichtert, was ein wichtiger Vorteil ist, wenn viele Anfragen gleichzeitig verarbeitet werden. Für den Aufbau unserem Webserver setzen wir auf das Echo Framework, das eine schlanke Struktur für Rest APIs bereitstellt.

Zusammen mit PostgreSQL, einer stabilen und leistungsfähigen relationalen Datenbank, entsteht so eine robuste backend Architektur, die zuverlässig und schnell ist. Ein weiterer Bestandteil des Tech Stacks sind die Build und Bundeling Tools. Hierbei nutzen wir vor allem Yarn als Paketmanager, mit yarn build können wir fertige Versionen erstellen, die beim Kunden eingesetzt wird. Über yarn watch lassen sich Änderungen am Code in Echtzeit, beobachten, sodass Anpassungen sofort sichtbar sind, ein enormer Vorteil für eine schnelle Entwicklungsarbeit. Zusätzlich kommt auch Vite zum Einsatz, ein modernes Build Tool, das sich durch sehr schnelle Entwicklung Server und kurze Build Zeiten auszeichnet. In meinem Team haben Leute statt Vite auch Webpack genutzt, dies ist ebenfalls sehr gut, da es sich ebenfalls durch seine hohe Flexibilität und umfangreiche Konfigurationsmöglichkeiten auszeichnet. Ich habe wie gesagt Vite genutzt und werden später genau drauf eingehen.

Zusammenfassend lässt sich sagen, dass der Tech bei B-ite aus bewusst gewählten modernen Technologien besteht. Die Kombination aus flexiblen Tools für die Infrastruktur, performanten Frameworks im Frontend und einer stabilen Architektur im Backend ermöglicht es, Projekt effizient umzusetzen und gleichzeitig den hohen Anforderungen der Kunden gerecht zu werden.

### **3.Praktische Tätigkeiten im Praxissemester Projekt 1**

Ein wesentlicher Bestandteil meines Praxissemester bestand in der täglichen Arbeit mit der B-ite Jobs API. Dabei handelt es sich um eine Schnittstelle über die Stellenanzeige von Kunden direkt auf deren Unternehmenswebseiten eingebunden werden können. Diese Aufgaben war fester Bestandteil meines Arbeitsalltags: Nahezu jeden Tag gab es neue Tickets, die Anpassung an bestehenden Integrationen, Erweiterungen für neue Kunden oder Fehlerbehebungen umfassten. Durch die Vielzahl an Projekten konnte ich einen tiefen Einblick in die praktische Arbeit mit Apis und die Herausforderungen der Integration gewinnen

#### **3.1 Erste Schritte mit der B-ite Jobs API**

Die Einbindungen der B-ite Jobs API folgt bei allen Projekten einem ähnlichen Aufbau. Zunächst wird für jeden Kunden ein eigenes Verzeichnis angelegt, das den jeweiligen Kundennamen trägt. Innerhalb dieses Ordners befinden sich dann die relevanten Dateien

für die Integration. Typischerweise besteht die Grundstruktur aus einer TypeScript(Tsx) für die Logik und einer SCSS-Datei für die Gestaltung.

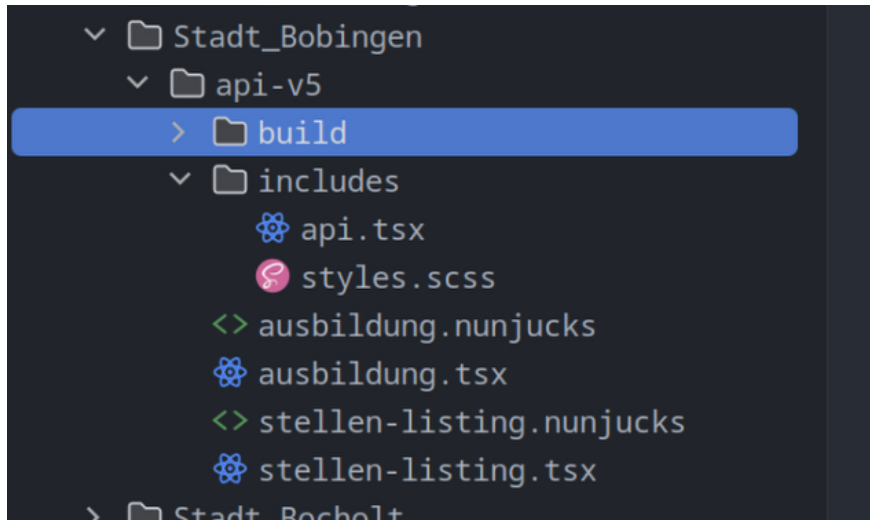


Abb. 1 - Ordner Struktur API

Abhängig von den Anforderungen des Kunden können mehrere Listings erstellt werden. Ein Listing ist dabei eine Art Ansicht der Stellenanzeige. Manche Kunden wünschen beispielsweise eine separate Übersicht für Ausbildungsstellen und eine weitere für reguläre Stellenangebote. In diesem Fall werden zusätzliche Dateien wie ausbildung.tsx oder stellen-listing.tsx angelegt. Diese Dateien sind bewusst schlank gehalten und erhalten nur den Aufruf der zentrale API logik

Die api.tsx bildet dabei das Herzstück der Integration. Hier wird festgelegt, wie die API mit dem Bewerbermanagement verbunden wird. Alle Funktionen und Komponenten werden hier geschrieben, es sei denn die listings unterscheiden sich stark voneinander, so tritt auch manchmal der Fall ein, dass eine zweite, beispielsweise Api2.tsx erstellt wird. Um zu verstehen was im Listing passiert, muss man erstmal in der Api.tsx reinschauen. Nach dem wir unsere Sachen importieren fängt unsere api.tsx immer so an.

```
export function api(listingName: string, channel: number, ref: string, zuordnung: string) {  
  v5InitListing(listingName).then(function ({Api, listing}: V5Loaded) {  
  
    /*Hier kommt die Logik rein*/  
  })  
}
```

Abb. 2 - Export function Api

Dort wird die Funktion api() exportiert, über die später die Listings angesteuert werden. Dabei haben die Parameter folgende Bedeutungen. Der listingname definiert, welches Listing geladen werden soll, z.B stellen-listing, Ausbildung oder je nach dem, wie der Kunde die listings benennen möchte. Der Channel gibt an, aus welchem Kanal die Daten geladen werden. Im Bewerbermanagement System gibt es 5 Channels, wo in der Regel

Channel 0 die Homepage vom Kunden ist, Channel eins das Intranet. Die restlichen Channels werden von den Kunden immer unterschiedlich verwendet, so ist Chanel 2 für die Ausschreibungen auf LinkedIn und Channel 5 für Test zwecke. Das ref bedeutet, das es eine Referenz ist, mit der das Listing eindeutig angesprochen wird. Der letzte Parameter wäre dann die Zuordnung, der wird nur erstellt, wenn der Kunde auch mehrere listings haben möchte. Es gibt auch Kunden die noch anderen Sachen wollen, das, was ich jetzt erklärt habe, ist aber nur der "Standard" Fall.

Die Funktion v5InitListing initialisiert anschließend das Listing, indem sie die notwendigen API-Komponente bereitstellt. In der Rückgabe erhalten wir unter anderen das Objekt Api, über das die Kommunikation mit dem Bewerbermanagement erfolgt, sowie das Objekt listing, in dem die eigentliche Stellendaten erhalten sind.

Nachdem wir nun verstanden haben, was die jeweiligen Parameter machen, schauen wir uns noch das listing genauer an.

```
1 import { api } from './includes/api';
2
3 api( listingName: 'spresso-listing', channel: 0, ref: 'homepage', zuordnung: 'spresso');
```

Abb. 3 - Listing

Auf diese Weise lässt sich für ein Kunden mit minimalem Aufwand mehr als ein Listing bereitstellen. Unterschiedliche Aufrufe nutzen immer dieselbe Logik aus der Api.tsx, ändern sich jedoch in den Parametern. Das spart nicht nur Entwicklungszeit, sondern sorgt auch dafür, dass Anpassungen an der Logik zentral an einer Stelle vorgenommen werden können.

Die SCSS-Datei sorgt dafür, dass das Layout an das Corporate Design des Kunden angepasst wird. Farben, Schriftarten, Abstände und Symbole werden so gestaltet, dass die Stellenseite optisch zum restlichen Internet auftritt des Unternehmens passen.

Auf diese Weise entsteht für Bewerbende ein einheitliches und professionelles Erscheinungsbild.

### 3.2 Build Prozess

Bevor eine Integration beim Kunden eingesetzt werden kann, müssen die entwickelten Dateien zunächst gebaut werden. Die Entwicklung findet in TypeScript / Preact / SCSS statt. Die Formate können Browser jedoch nicht direkt verarbeiten. Deshalb werden sie durch ein Build Prozess in optimierte JavaScript und CSS-Datei übersetzt.

Während der Entwicklung nutze ich den Befehl yarn watch, um Änderungen live zu verfolgen. Das bedeutet, wenn ich im Code oder im Layout etwas ändere, sehe ich die Anpassung sofort im lokalen Testlauf. Dadurch lassen sich Fehler schnell finden und beheben.



```
hostuser@ws-43:/project/projekts/customer-templates$ yarn watch -p st/Stadt_Bobingen/api-v5
yarn run v1.22.21
warning ../../package.json: No license field
$ gulp watch -p st/Stadt_Bobingen/api-v5
[19:44:44] Using gulpfile /project/projekts/customer-templates/Gulpfile.js
[19:44:44] Starting 'watch'...
Project config
{
  mode: 'build-dev',
  paths: {
    root: 'st/Stadt_Bobingen/api-v5',
    out: 'st/Stadt_Bobingen/api-v5/build-dev',
    styles: [ 'st/Stadt_Bobingen/api-v5/*.scss' ],
    scripts: [
      'st/Stadt_Bobingen/api-v5/*.ts',
      'st/Stadt_Bobingen/api-v5/*.tsx'
    ],
    html: [ 'st/Stadt_Bobingen/api-v5/*.nunjucks' ]
  }
}
```

Abb. 4 - Build Api

Sobald ein Projekt fertiggestellt wird, wird es mit yarn build kompiliert. Dieser Befehl erstellt aus den .tsx und .scss Dateien die finalen Ressourcen, die später beim Kunden ausgeliefert werden. Je nachdem, wie viele Listings ein Kunde hat, entsteht dabei entsprechend viele JavaScript Dateien ( .js ) und zusätzlich die minimierte Version .min.js. Ein Beispiel, hat ein Kunde zwei verschiedene Listings, eines für reguläre und eines für Ausbildungsplätze, so werden beim Build Prozess zwei Paare an Dateien erstellt. In unserem Beispiel wären das einmal stellen-listing.js und stellen-listing, des Weiteren die ausbildung.js und ausbildung.min.js.

Die JS-Datei sind unkomprimierten Version, die im Zweifel noch gelesen oder debuggt werden können. Die .min.js Datei hingegen sind komprimiert und damit deutlich kleiner, was Ladezeit spart und die Performance auf der Kundenwebseite verbessert. Durch diesen Build Prozess wird sichergestellt, dass alle Änderungen aus der Entwicklung korrekt in Produktionsfähige Dateien übersetzt werden, die dann über das System an den Kunden ausgeliefert werden können.

### 3.3 Vertiefung einer Standard API

Nun werde ich eine einfache und Standard API zeigen und erklären.

```
Api.utils.appendStyles(styles);

const apiClient: Client = Api.createClient({
  key: ''
});

const searchConfig: ApiSearchConfig = Api.createSearchConfig({
  channel: channel,
  locale: 'de',
  sort: {
    by: 'title',
    order: 'asc'
  },
  page: {
    num: 1000
  }
});
```

Abb. 5 - Beispiel für eine API-Client und Suchkonfiguration

Mit der ersten Zeile binden wir die zugehörige SCSS-Datei ins Listing ein. Dadurch trennen wir Logik und style voneinander. Der nächste Schritt ist das Erstellen eines API-Client mit Authentifizierung. Der Client kümmert sich um alle Request zu Bewerbermanagement. Der searchConfig Block ist für die API auch sehr wichtig, dort geben wir die Sprach und Regionseinstellungen für Texte und Datums / Zahlformatierungen. Ebenfalls geben wir an nach was sortiert werden soll, also wie die Stellenanzeigen angezeigt werden sollen. In dem Fall wird nach dem Titel aufsteigend sortiert. Der Channel Block ist der "Kanal" für die Stellen, wie zum Beispiel eine öffentliche oder interne Sicht. Page brauchen wir, um eine Maximale Anzahl an Datensätze zu haben. Bevor ich die erste Komponente erkläre, sollte ich zunächst hierauf eingehen.

```

    }
  });

  const controlConfig : ApiControlConfig = Api.createControlConfig({
    ref
  });
  const control : Control = Api.createControl(apiClient, controlConfig, searchConfig);
  const {h, Template, render} = Api.ui;
  const {formatDate2} = Api.utils;

```

Abb. 6 - Beispiel für eine ControlConfig in der Jobs Api

ControlConfig nutzen wir, da es der Komponente hilft zu identifizieren. Die Konstante Control ist unser "Schaltplatt". Darüber holen wir Daten, setzen Filter, Paginieren, reloaden und viel mehr.

Die nächste Zeile ist eine UI-Hilfe, wobei das h für die JSX/VDOM Fabrik steht. Template ist die Basisklasse für wiederverwendbare Listen und Details Komponente.

Wichtig zu erwähnen ist natürlich, dass die Funktionen wie Api.createClient oder Api.createControl nicht aus meinem eigenen Code stammen, sondern ein Teil einer bereitgestellt Bite Jobs API Bibliothek ist. Diese Bibliothek dient als Schnittstelle zwischen dem Frontend Code und dem eigentlichen Backend des Bewerbermanagement.

```

class JobList extends Template {
  show usages  A Alton
  content() {
    const jobPostings : ApiJobPosting[] = control.jobPostings;
    const entries: any[] = [];
    jobPostings.forEach(posting : ApiJobPosting => {
      entries.push(
        <a class="bite-jobs-list--row"
          href={control.fns.getUrl(posting)} target="_blank">
            <div class="bite-jobs-list--row--title">
              {posting.title}
            </div>
            <div class="bite-jobs-list--row--endsOn">
              {formatDate2(posting.endsOn)}
            </div>
          </a>
        );
      });
    return <div class="bite-jobs-list">{...entries}</div>;
  }

  show usages  A Alton
  noContent() {
    return <div class="bite-jobs-list--noresult">
      Derzeit haben wir keine offenen Stellen zu besetzen.
    </div>;
  }
}

```

Abb. 7 - Template Class

Nachdem die API-Logik initialisiert und die Daten über das Control Objekt geladen wurden, müssen die Stellenanzeigen auch im Frontend sichtbar gemacht werden. Dafür

wird eine eigene Komponente geschrieben, die von der bereitgestellten Klasse Template erbt.

Die Klasse JobList erweitert die Basis Klasse Template, welche aus der API-Bibliothek kommt. Dort drinnen werden einmal der Content und die noContent erstellt.

In der Content Methode erstellen wir für jede Stellenanzeige nun ein Design, welches den Titel und das Datum, welches man im System einstellen kann, angibt. Falls es keine Ausschreibungen gibt, so wird die noContent Methode ausgeführt.

```
const app : Element = <div class={"bite-container"}>
  <JobList control={control}/>
</div>
render(app, listing.element);
control.reload();
});
}
```

Abb. 8 - App

Zum Schluss wird die eigentliche "App" gebaut, also das, was im Browser angezeigt wird. Innerhalb des Containers wird die Komponente JobList aufgerufen. Der JobList Klasse wird das zuvor erstellte Control Objekt übergeben, dadurch erhält die Komponente Zugriff auf alle Stellendaten.

Mit render () welches ebenfalls bereitgestellt wird aus der Api.ui, wird die App tatsächlich in das DOM-Element der Seite eingefügt.

Das listing.element ist dabei der Platzhalter im HTML, an dem die Stellenlisten erscheinen soll.

Mit control.Relode() wird ein erster Request an das Backend geschickt, die aktuellen Stellenanzeigen werden geladen und in der JobList angezeigt.

### 3.4 Ergebnis

Jedoch haben wir auch Kunden (und das ziemlich oft) die natürlich nicht nur Standard APIs haben wollen, sondern sehr spezielle, wo es mal schnell bis zu 1000 Zeilen gehen kann für einen Kunden. Als Beispiel füge ich mal einen Kunden ein, welches ich am Anfang meines Praxissemester erstellt habe. Solche Kunden mit sehr speziellen Wünschen gibt es sehr oft, daher habe ich einfach mal eins ausgesucht.

Radius, Location, Dropdown, LotusResult, SelectedFilters, Counter, Filters Komponente

habe ich für diesen Kunden (Vertical Cloud Solution) erstellt.

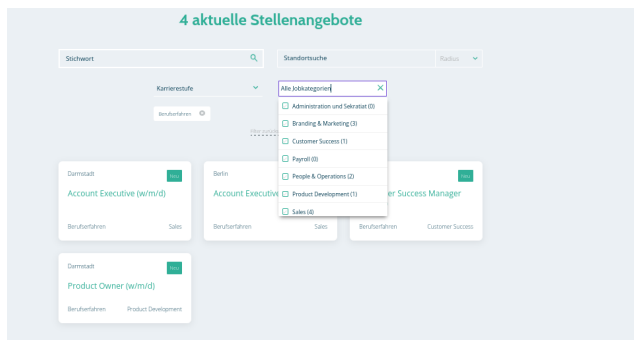


Abb. 9 - Live Api Vertical Cloud Solution

Stand jetzt habe ich fast 200 Apis erstellt und bearbeitet, man muss natürlich bedenken das ich auch als Werkstudent angefangen habe, wobei ich damals noch keinen großen Apis selbst erstellt habe, was sich seit Beginn meines Praxissemester natürlich geändert hat.

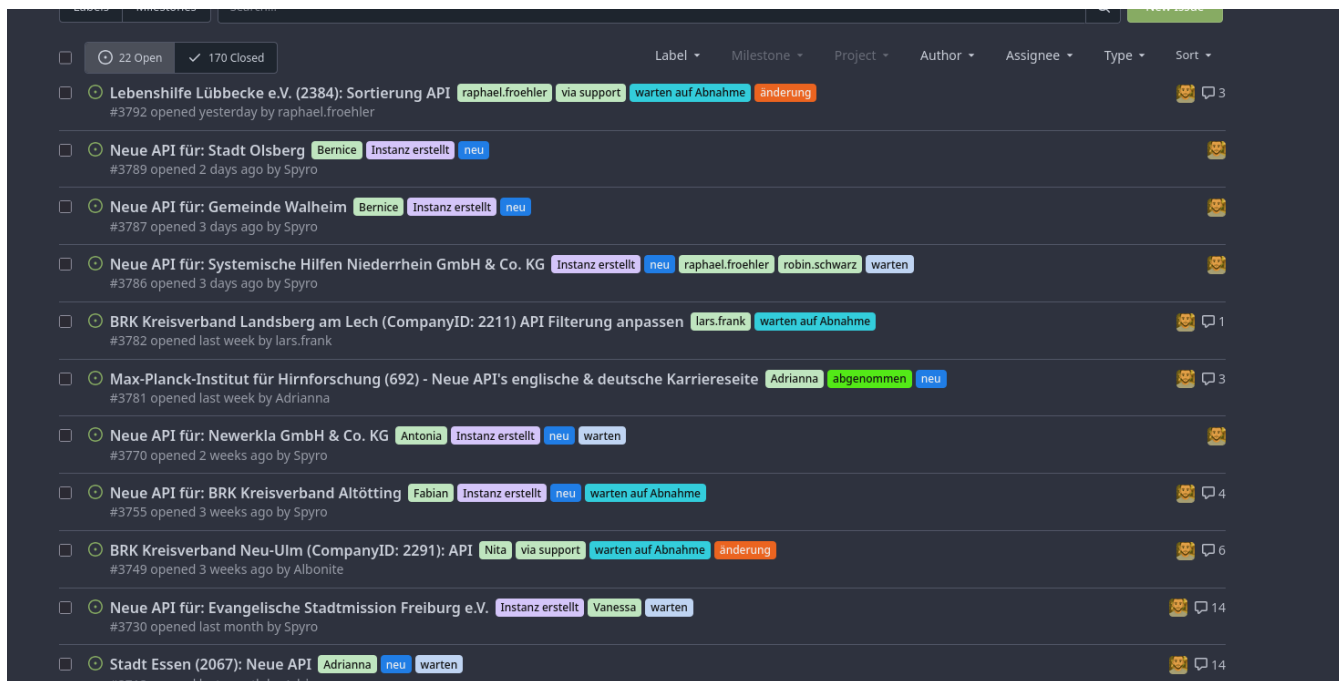


Abb. 10 - Tickets APIs

## 4.Praktische Tätigkeiten im Praxissemester Projekt 2

Ein weiterer zentraler Bestandteil meiner Arbeit waren die sogenannten Templates. Darunter versteht man bei B-ite die individuellen Vorlagen für die Darstellung von Stellenausschreibungen auf den Webseiten der Kunden. Während die API die Daten bereitstellt, sorgen die Templates dafür, dass diese Stellenanzeige optisch passend im Corporate Design des jeweiligen Unternehmens erscheinen. Meine Aufgabe bestand darin, diese Templates mit HTML, CSS und JavaScript umzusetzen, anzupassen und zu erweitern, sodass die Inhalte nicht nur funktional, sondern auch benutzerfreundlich und visuell ansprechend dargestellt wurden.

### 4.1 Erste Schritte in den Templates

Der Aufbau eines Templates ist in der Regel gleich, zur Veranschaulichung habe ich ein Bild von unserem Cs-Templates Ordner genutzt, wo wir alle Templates erstellen.

Templates sind also nicht nur einzelne Dateien, sondern bestehen aus klar definiertem Ordner und Dateistrukturen. Diese Struktur erleichtert die Wartung, ermöglicht die Teamarbeit und sorgt dafür, dass bei jedem Kunden ein einheitlicher Standard eingehalten wird.

Im Beispiel des Kunden Newerkla GmbH sieht man folgende Struktur:

Im Ordner styles/source liegen die SCSS-Dateien, die das Design der Stellenanzeige definieren. Über den Build Prozess werden daraus fertige CSS-Dateien erzeugt, die im Ordner Build gespeichert sind. Inklusive Vorschauen (z.B preview.pdf) und archivierte Versionen. Im Bereich remote\_css befinden sich tatsächlich an den Kunden ausgelieferte Dateien, die später auf der Webseite eingebunden werden. Die Datei template.nunjucks stellt die HTML-Grundstruktur dar, in die die Stellenanzeigen integriert werden. Die Build Prozesse überspringen wir hier, da es sehr identisch, wie bei den Jobs-APIs sind.

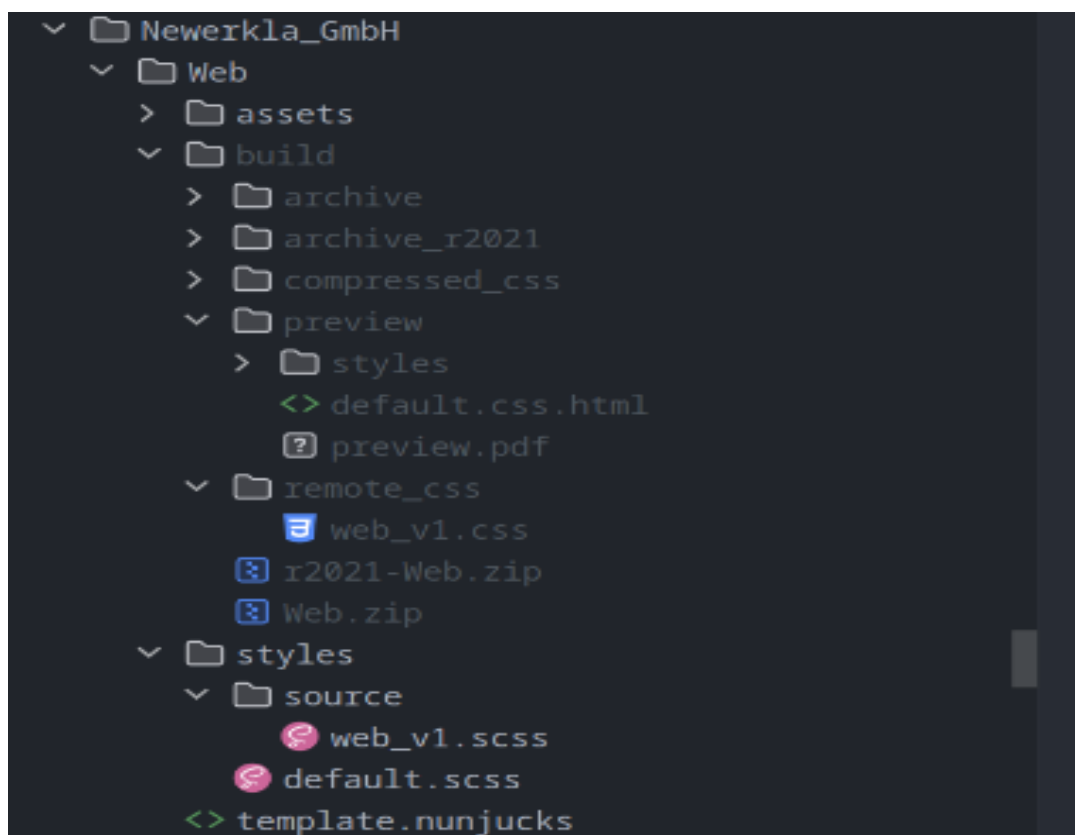
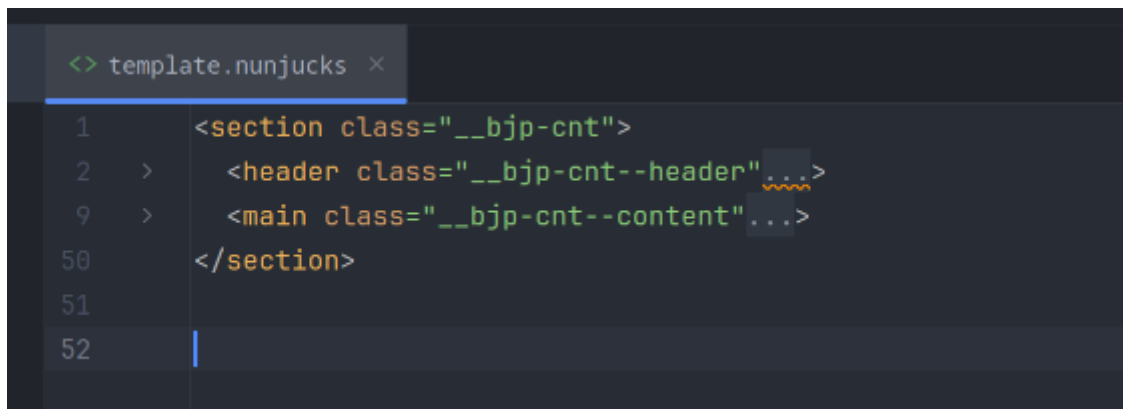


Abb. 11- Template Struktur

## 4.2 Vertiefung der Templates



```
<> template.nunjucks x
1  <section class="__bjp-cnt">
2  >   <header class="__bjp-cnt--header"...>
9  >   <main class="__bjp-cnt--content"...>
50 </section>
51
52
```

Abb. 12 - HTML Aufbau

Die Abbildung zeigt den Kern einer Template Struktur in der Datei templates.nunjucks. Dabei umschließt die Section Klasse das gesamte Template. Sie ist wie ein Container, der alle Bereiche der Stellenanzeige bündelt. Der Header Teil umfasst meist Banner, Logo und Titel. In der Main Klasse befindet sich der Hauptinhalt. Das Template basiert auf einem von uns entwickelten Framework, bei dem bestimmte Klassennamen direkt mit festgelegten CSS-Attribute verknüpft sind. Alle Klassen beginnen dabei mit dem Präfix \_\_bjp-cnt, wodurch eine klare Struktur entsteht und wir die Elemente in SCSS sauber verschachteln können. So stellen wir sicher, dass sich das Design Konsistenz aufbauen lässt und Änderungen schnell und gezielt umgesetzt werden können.

Wie man sieht, verwenden wir kein Footer, da der Footer unsere globalen Buttons sind, die in den Ausschreibungen automatisch eingebunden werden, sobald wir es bauen. Die führen dann zurück zu der Homepage, E-Mail-Bewerbungen, zu unserem Bewerbermanagement System, WhatsApp Bewerbung und vieles mehr. Dort sind die Kunden sehr verschieden, was das angeht, welche Buttons angezeigt werden wollen. Auf den HTML-Teil gehe ich in dieser Dokumentation nicht weiter ein, da dieser lediglich das Grundgerüst bildet und im Vergleich zu den anderen Sachen weniger relevant sind. Dennoch habe ich täglich dran gearbeitet, weswegen es wichtig ist, dies zu erwähnen.

```
@import "https://static.b-ite.com/css/templates/columns/2.0.0.css";
$blau:#2c7ec1;
$gelb:#ffec00;
$background:#13233c;
$white:#fff;
$black:#000;

body {
  font-family: Arial, sans-serif;
}

.____b_jp-cnt {
  * {
    box-sizing: border-box;
    word-break: break-word;
  }

  color: $black;
  &--text{
    position:absolute;
    font-size: clamp(1rem, 0.7816rem + 1.0495vw, 1.4375rem);
    z-index:1;
    left:2%;
    padding:10px;
    background-color:$background;
    color:$white;
    width:40%;
    top:30%;
  }

  &--ul-custom{
    ul {
      counter-reset: list-bullet-points-circle;
    }
  }
}
```

Abb. 13 - CSS

In der gezeigten SCSS-Datei sieht man, die Einbindung unser kleines internes Framework. Am Anfang werden zentrale Farben definiert, die sich dann im gesamten Template wiederverwenden lassen. Darunter folgen die grundlegenden Styles für den Kunden wie Schriftart, Abstände Positionen und viel mehr. Darauf aufbauend lassen sich dann verschiedene Varianten anlegen, etwa für die Mobile Darstellung oder die Print Version. Dadurch können wir sicherstellen, dass die Stellenausschreibungen unabhängig vom Endgerät oder Anwendungsfälle einheitlich und professionell aussehen. Das Einbinden der JavaScript-Dateien, welche die Templates dynamischer machen, erfolgt über Script-Tags am Ende des HTML-Dokuments. Zunächst wird das JavaScript lokal in die HTML-Datei geschrieben und getestet. Anschließend wird die fertige Datei in den Static-Bereich hochgeladen. Von dort aus wird das Script am unteren Ende des HTML-Codes eingebunden, sodass es beim Laden der Seite ausgeführt wird und zusätzliche Funktionalitäten bereitstellt.

### 4.3 Das Hochladen eines Templates

Nachdem ein Template fertiggestellt ist, wird der Code zunächst in BiteHub hochgeladen. Dies hat den Vorteil, dass auch meine Kolleginnen und Kollegen jederzeit Zugriff auf den aktuellen Stand haben und bei Bedarf Änderungen vornehmen können. Gerade wenn ein Ticket nach einiger Zeit erneut geöffnet wird, ist es wichtig, dass die Arbeitsschritte und Anpassungen für alle nachvollziehbar sind und eine gemeinsame Bearbeitung möglich bleibt.

Parallel dazu wird das Template auch in unser internes System hochgeladen. Beim sogenannten Build-Prozess entsteht automatisch eine ZIP-Datei, die nach einem

bestimmten Schema benannt wird (z. B. r-2021.zip). Diese Datei enthält alle notwendigen Ressourcen und wird in das System importiert, wodurch das eigentliche Template entsteht.

Das HTML kann anschließend über den integrierten Editor bearbeitet werden. Die zugehörigen CSS-Dateien hingegen sind nicht direkt editierbar, da wir diese aus Sicherheitsgründen nur als externen Link einbinden. Ein Beispiel dafür ist folgender Import:



Abb. 14 - Import CSS Datei in Bewerbermanagement

Der Grund für dieses Vorgehen liegt darin, dass Änderungen am Styling ausschließlich von uns als Entwickler vorgenommen werden sollen. Würde der Kunde direkten Zugriff auf die CSS-Dateien erhalten, bestünde die Gefahr, dass unsauberer oder fehlerhafter Code eingebaut wird, was das gesamte Template beeinträchtigen könnte.

Nach dem erfolgreichen Hochladen eröffnet das System zahlreiche weitere Möglichkeiten. So können beispielsweise Tochtergesellschaften angelegt werden, deren Stellenanzeigen über verschiedene Portale exportiert werden. Außerdem besteht die Möglichkeit, Datenfelder einzurichten. Diese erleichtern die Arbeit erheblich, da der Kunde Inhalte flexibel anpassen kann, ohne jedes Mal den HTML-Code bearbeiten zu müssen.

Ein praktisches Beispiel: Hat ein Kunde zehn Stellenausschreibungen mit identischem Aufbau, so müssen die Inhalte nicht mühsam in allen HTML-Dateien einzeln verändert werden. Stattdessen können die Texte zentral über die hinterlegten Datenfelder gepflegt werden. Änderungen werden dann automatisch in allen entsprechenden Templates übernommen.

Neben diesen Funktionen existiert noch eine Vielzahl weiterer Möglichkeiten, die hier jedoch nicht im Detail beschrieben werden. Für den Rahmen dieser Dokumentation, die zwischen 20 und 25 Seiten umfassen soll, konzentriere ich mich auf die zentralen Aspekte meiner Arbeit.



## 4.4 Ergebnis

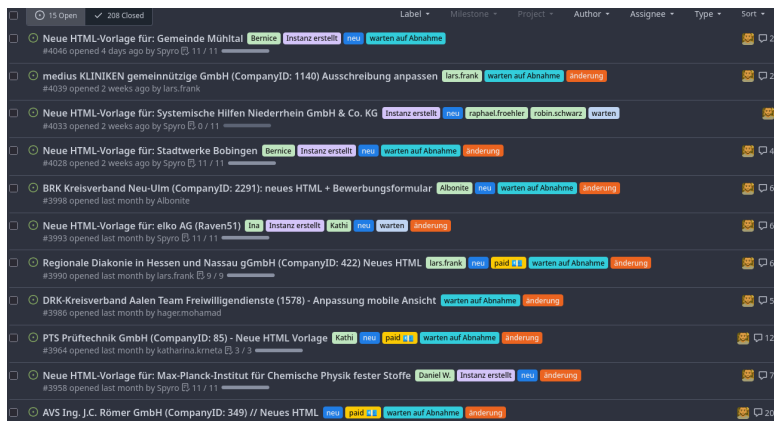


Abb. 15 - Tickets Templates

Als Ergebnis meiner Arbeit mit den Templates lässt sich festhalten, dass ich während meines Praxissemesters über 200 Tickets bearbeitet habe. Darunter waren sowohl kleinere Anpassungen für bestehende Kunden als auch komplette Neuentwicklungen von Templates für Unternehmen. Besonders herausfordernd war dabei, dass viele Kunden sehr individuelle Wünsche und Vorgaben hatten, sei es im Hinblick auf das Corporate Design, die technische Integration oder spezielle Zusatzfunktionen.

Ein konkretes Beispiel hierfür ist das Projekt für gastromatic. Während ich zuvor bereits die Anbindung der Jobs-API für dieses Unternehmen vorgestellt habe, zeigt das hier dargestellte Bild die Umsetzung des zugehörigen Templates. Ziel war es, die Stellenausschreibungen nicht nur optisch im Corporate Design von gastromatic darzustellen, sondern auch dynamische Elemente über die API direkt in das Template einzubinden. So konnten Inhalte wie Stellenbezeichnungen, Anforderungen oder Icon automatisiert geladen und angezeigt werden.

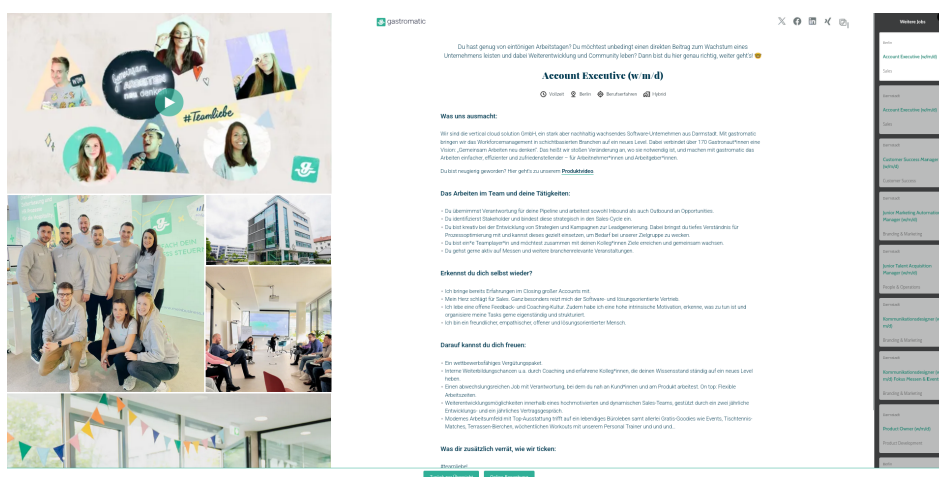


Abb. 16 - HTML Live Kunden Vertical Cloud Solution

Das Ergebnis ist ein modernes, ansprechendes Layout, das zusätzlich mit multimedialen Inhalten wie Bildern und Videos angereichert wurde. Auf der linken Seite sieht man eine Collage aus Teamfotos und Unternehmensimpressionen, die potenziellen Bewerberinnen

und Bewerbern einen authentischen Einblick in die Unternehmenskultur vermittelt. Die Form wie die Bilder erscheinen wollte der Kunde genauso haben, das heißt man muss perfekt mit CSS und JavaScript dies darstellen. Auf der rechten Seite befindet sich der eigentliche Stellenausschreibungstext, ergänzt durch eine übersichtliche Navigation zu weiteren offenen Positionen (eine weitere API-Einbindung).

Anhand dieses Beispiels wird deutlich, wie vielfältig und praxisnah meine Arbeit an den Templates war: von der reinen Frontend-Gestaltung (in dem Fall mit HTML, CSS, JavaScript, Preact und Typescript) über die technische Integration bis hin zur Abstimmung mit den individuellen Wünschen der Kunden. Genau diese Kombination aus technischer Umsetzung und kundenorientierter Anpassung machte die Arbeit spannend und abwechslungsreich. Auch für dieses Template habe ich insgesamt über 1000 Zeilen Code geschrieben.

## 5. Praktische Tätigkeit im Praxissemester Projekt 3

Zu Beginn des Projekts existierte bereits ein Frontend-Prototyp für ein Cookiebanner (von einer Kollegin umgesetzt). Was fehlte, war ein sauberes backend mit Datenbank, API und Admin-Oberfläche, damit Banner für verschiedene Kunden zentral erstellt, versioniert und verwaltet werden können.

Mein Tech-Stack: Go (Backend) + PostgreSQL (DB) + Echo (HTTP-Framework) sowie ein Admin-Tool in React mit Vite als Bundler und Elastic UI (EUI) für die Komponenten.

Detailliert habe ich es in Kapitel 2 erklärt, wieso ich mich für diese Sprachen, Datenbank und Frameworks entschieden habe.

### 5.1 Fachliche Analyse der Datenbank

Ich habe zunächst den bestehenden Frontend-Code analysiert (welche Felder/Strukturen werden angezeigt, welche Beziehungen gibt es?) und daraus ein UML/ER-Modell abgeleitet. Die Kern-Entitäten sind hierarchisch organisiert:

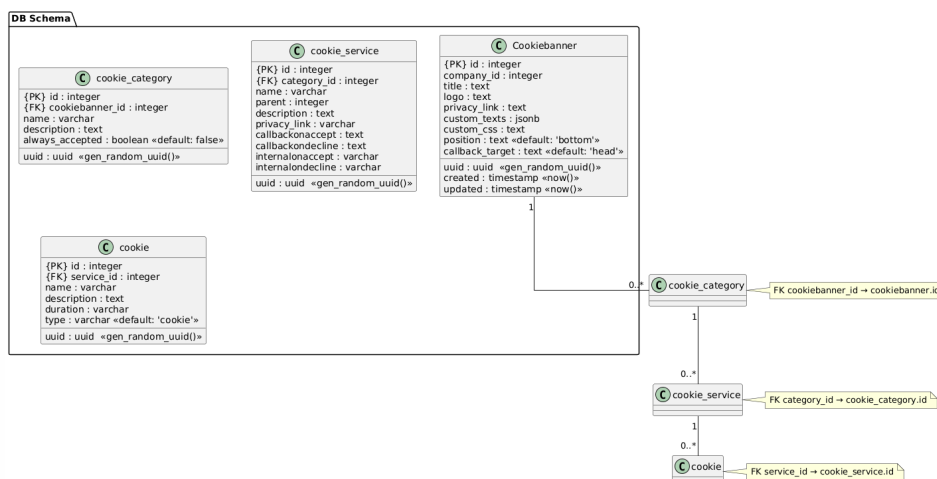


Abb. 17 - UML Diagramm Cookiebanner mit StarUML

Ein Banner kann viele Kategorien haben, eine Kategorie viele Services, ein Service viele Cookies. Damit sind granulare Einwilligungen und feingliedrige Konfiguration möglich: Ein Kunde kann global ein Banner pflegen, einzelne Kategorien/Services aktivieren und die darunterliegenden Cookies transparent dokumentieren.

Kernobjekt ist cookiebanner mit administrativen Metadaten sowie konfigurierbaren Inhalt. Freitexte und mehrsprachige variablen werden in `custom_texts` ( jsonb ) gespeichert, das macht das System erwartbar, ohne das Schema zu Ändern. Position und `Callback_Target` steuern, wo das Banner erscheint bzw. wo Skripte injiziert werden. Auf den Rest gehe ich aus Platzmangel nicht ein, da der Rest wie `uuid`, `id`, `custom_css` usw. Selbstverständlich sind. Die `cookie_category`, ordnet Service fachlich und erbt die Zugehörigkeit über `cookiebanner_id`. Neben Namen und description markiert `always_accepted` systemkritische Kategorien (`true` = nicht abwählbar). Jede Kategorie hat eine eigene `uuid` und kann beliebig viele Services tragen.

`Cookie_service` beschreibt konkrete Dritt Dienste (z.B Youtube). Über `Category_id` ist der Dienst einer kategorie zugeordnet. Mit `callbackonaccept` und `callbackdecline` sowie `internalonaccept` und `internalondecline` lassen sich Skripte oder interne Hooks pro Dienst auslösen (z.B Script Injection erst nach einwillige). Das Feld `parent` ermöglicht optional Service Hierarchien (z.B Varianten eines Dienstes). Cookie hält die eigentlichen Cookies eines Dienstens.

## 5.2 Das Backend des Cookiebanners

Nach der fachlichen Analyse des Datenmodells folgt die technische Umsetzung im Backend mit Go und dem Framework Echo.

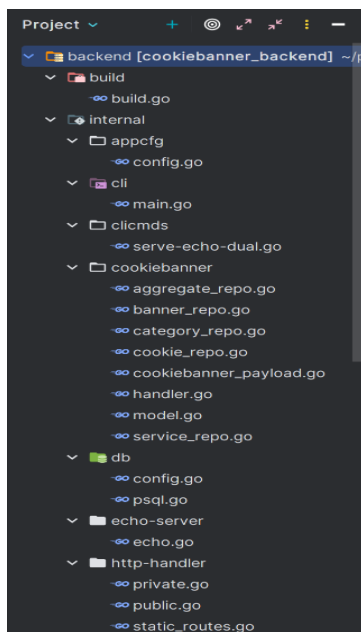


Abb. 18 - Cookiebanner backend Struktur

Nach der Beschreibung der Datenbank erkläre ich das Backend. Ich habe es mit Go und Echo gebaut. Die Teile sind klar getrennt. Es gibt eine Stelle, die die App startet, die

Einstellungen lädt und die Server einschaltet. Es gibt eine Stelle für die Einstellungen der App, dort stehen Dinge wie die Ports und die Daten für die Verbindung zur Datenbank. Für die Datenbank nutze ich eine eigene Datei, die die Verbindung herstellt und offenhält. Es gibt auch einfache Prüfungen, die regelmäßig schauen, ob Datenbank und Server noch erreichbar sind.

Echo ist der Webserver. Ich habe zwei Server gestartet. Einer ist öffentlich und darf nur lesen. Der andere ist privat und nur intern erreichbar, dort kann man Daten anlegen, ändern oder löschen. Diese Trennung macht das System sicherer, denn die sensiblen Funktionen sind nicht von außen sichtbar. Für die öffentlichen Adressen und die privaten Adressen habe ich getrennte Dateien. Außerdem gibt es eine kleine Adresse, die einfach nur „ok“ zurückgibt. Damit kann man schnell prüfen, ob der Server läuft.

Der Bereich „cookiebanner“ enthält die eigentliche Fachlogik. Dort liegen die Datenmodelle für Banner, Kategorien, Services und Cookies. Es gibt Datenstrukturen für das Anlegen und Bearbeiten, die prüfen auch die Eingaben. Die Handler nehmen die Anfragen aus dem Internet an, sie lesen die Daten aus dem Request und geben Antworten zurück. Die Service-Schicht steuert die einzelnen Schritte, sie ruft die Funktionen der Repositories auf und sorgt dafür, dass alles in der richtigen Reihenfolge passiert. Wenn mehrere Schreibvorgänge zusammengehören, startet sie eine Transaktion, das heißt, entweder alles wird gespeichert, oder nichts, damit keine halben Zustände entstehen. Die Repositories sprechen direkt mit der Datenbank und führen die SQL-Befehle aus.

Ein typischer Ablauf beim Anlegen eines neuen Banners läuft so. Die Anfrage kommt beim privaten Server an. Der Handler liest den Inhalt, prüft die Pflichtfelder und gibt die Daten an den Service weiter. Der Service startet eine Transaktion. Zuerst wird das Banner gespeichert, danach die Kategorien, dann die Services und zum Schluss die Cookies. Wenn etwas schiefgeht, wird alles zurückgedreht. Wenn alles funktioniert, wird gespeichert und die neue Kennung des Banners zurückgegeben. Später fragt der öffentliche Server mit einer Adresse alle Daten zu dieser Kennung ab und liefert das komplette Banner an das Skript auf der Kundenseite.

Auf einzelne Codeabschnitte gehe ich nicht ein, da das ebenfalls eine Menge an Code Zeilen sind und das wieder den Rahmen der Dokumentation sprengen würde.

### **5.3 Web User Interface**

In diesem Projekt bestand meine zweite Aufgabe darin, ein Admin-Tool zu entwickeln. Der Hintergrund war, dass mein Chef nicht jedes Mal direkt in der Datenbank einen Cookiebanner anlegen wollte, da dies zu aufwendig und zeitintensiv wäre. Stattdessen sollte ein eigenes Tool diesen Prozess deutlich vereinfachen. Besonders positiv war für mich, dass ich frei entscheiden durfte, welche Technologien ich verwenden möchte. Ich

habe mich dabei für React in Kombination mit dem Framework Elastic UI entschieden und Vite als Bundler genutzt.

### 5.3.1 Beschreibung der Struktur

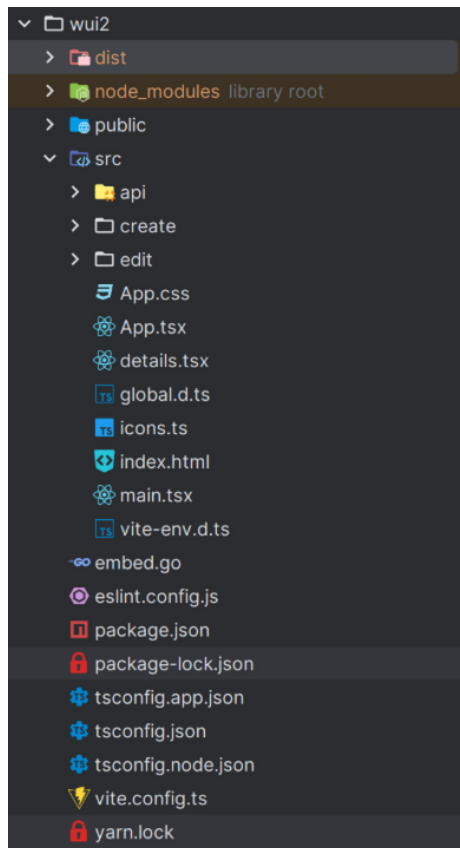


Abb. 19 - Struktur Web User Interface

Das Projekt besteht aus mehreren zentralen Bereichen, die zusammen das Admin-Tool bilden. Im Ordner `dist` liegen die fertig gebauten Dateien, die später vom Go-Server ausgeliefert werden. Der Ordner `node_modules` enthält alle installierten Pakete, die für die Entwicklung benötigt werden. Unter `public` können statische Dateien wie Logos oder Favicons abgelegt werden, die beim Build unverändert übernommen werden. Der wichtigste Teil befindet sich im Ordner `src`. Dort sind die eigentlichen Quellcodes abgelegt. In den Unterordnern `api`, `create` und `edit` befinden sich die Hilfsfunktionen für den Zugriff auf das Backend sowie die Komponenten zum Anlegen und Bearbeiten von Bannern, Kategorien, Services und Cookies. Die Datei `App.tsx` bildet den Einstiegspunkt der Anwendung und steuert das Layout sowie den übergeordneten Zustand. Die dazugehörige `App.css` kümmert sich um globale Styles. Mit `details.tsx` wird die Detailansicht für einen bestimmten Banner umgesetzt, während `icons.ts` eine zentrale Verwaltung der verwendeten Icons bereitstellt. In `global.d.ts` und `vite-env.d.ts` liegen TypeScript-Deklarationen, damit der Code korrekt typisiert ist. Die `index.html` dient als Einstieg für Vite und bindet `main.tsx` ein, welches die React-App startet. Zusätzlich gibt es

noch die Datei `embed.go`, die das Frontend mit dem Go-Backend verbindet, sodass die Anwendung später direkt über den Echo-Server erreichbar ist.

Im Projektverzeichnis selbst finden sich die Konfigurationsdateien. `Eslint.config.js` enthält Regeln für sauberen Code, während `package.json` die Abhängigkeiten und Skripte definiert. Die verschiedenen `tsconfig`-Dateien regeln die Einstellungen für TypeScript, aufgeteilt nach Anwendungen und Node-Umgebung. Die Datei `vite.config.ts` enthält die zentrale Konfiguration des Bundlers Vite. Dort wird festgelegt, wie das Projekt gebaut wird, welche Plugins eingebunden sind und über welche Proxy-Einstellungen die Verbindung zum Backend während der Entwicklung erfolgt. Schließlich sorgt die `.gitignore` dafür, dass unnötige Dateien wie Build-Ordner oder Cache nicht ins Git-Repository gelangen.

Im Zusammenspiel funktioniert das Projekt so: Während der Entwicklung wird mit `yarn dev` der Vite-Server gestartet, der Änderungen sofort sichtbar macht. Das Backend läuft dabei parallel über Echo und wird bei Bedarf über den Proxy in der Vite-Konfiguration angesprochen. Mit `yarn build` wird ein finaler Build erzeugt, der im Ordner `dist` landet. Der Go-Server liefert diesen Ordner zusammen mit den API-Routen aus, sodass die Admin-Oberfläche direkt auf die Daten zugreifen und alle CRUD-Operationen durchführen kann.

### 5.3.2 Admin Tool

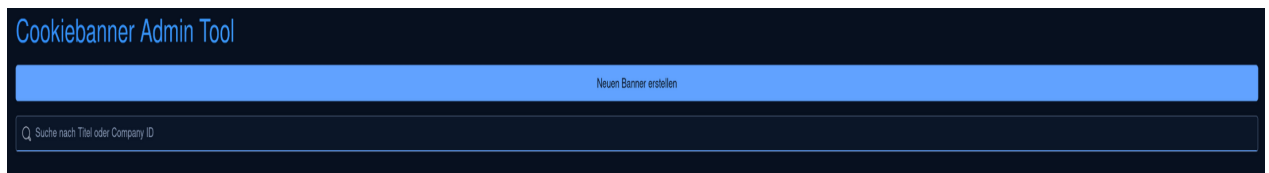


Abb. 20 - Cookiebanner suchen

Nun möchte ich kurz darauf eingehen, was man im Admin Tool alles machen kann. Als aller erstes kann man einen Neuen Banner erstellen, falls es schon welche gibt kann man nach der ID oder nach dem Namen filtern. Wir gehen nun davon aus dass es keinen Cookiebanner gibt.

**Neuen Cookiebanner erstellen**

Company ID: z.B. 1

Position: z.B. Top | Center | Bottom

Kundendaten:

Logo URL:

Privacy Link:

Custom CSS:

CallBack: Head | Body

Abb. 21 - Cookiebanner erstellen

Beim Klicken auf den Button öffnet sich das Modal welches die ganzen Punkte des Cookiebanner besitzt. Ich erinnere noch einmal, dass ein cookiebanner mehrere Kategorien haben kann, er wiederum aber nur zu einem Cookiebanner dazu gehört. Nach dem wir nun alle Felder ausgefüllt haben für den Cookiebanner, können wir nun beliebig viele Kategorien erstellen.

Custom-Text:

Custom-Text:

Custom-Text:

Kategorien  
 > Kategorie 1: Neu

Abb. 22 - Kategorie Erstellen

In dem Beispiel werde ich nur eine Kategorie erstellen. Das Ganze funktioniert so, dass es alles ineinander verschachtelt ist. Bedeutet, wenn ich die Kategorie erstelle, habe ich erst dann die Möglichkeit einen Service zu erstellen, es sollte ja genau deshalb so sein, da man ja eben ein Service erstellen könnte ohne Kategorie, was nutzlos wäre.

Kategorien

✓ Kategorie 1: Statistik


**Name:** z.B Statistik

Statistik

**Beschreibung:** z.B ...

Erfassung von anonymisierten Daten für Statistiken. Mithilfe dieser Cookies können wir beispielsweise die Besucherzahlen und den Effekt bestimmter Seiten unseres Web-Auftritts ermitteln und unsere Inhalte laufend optimieren.

Immer Akzeptieren?: Ja / Nein

☒ 

 Always Accepted

> Service 1: Neu

Service hinzufügen

*Abb. 23 Service erstellen*

Nach dem wir nun die Kategorie hinzugefügt haben, erscheint der Button für den Service. Damit wir ein Service erstellen müssen wir mindestens eine Sache in unsere Kategorie schreiben. Das Beziehung zwischen Kategorie und Service ist identisch, wie die Beziehung Cookie und Service, weswegen ich darauf weiter nicht eingehe. Nach dem wir also unser Cookiebanner erstellt haben, sieht es folgendermaßen aus:

Cookiebanner Admin Tool

Neuen Banner erstellen

Q Suche nach Titel oder Company ID

Technische Hochschule Ulm, Company ID: 2

Abb. 24 - Filterung nach einem Cookiebanner

Pro Cookiebanner erscheint eine Kachel mit der ID und dem Titel. Wir können den Cookiebanner bearbeiten und Löschen, dabei ruft er einfach nur die Put und Delete Routen von meinem Backend auf. Wenn wir das ganze editieren, so editieren wir nur die Oberfläche, also nicht die Kategorie, Service oder Cookie.

Mit einem Klick auf dem Cookiebanner listen wir uns alle Punkte zu dem Cookiebanner und seinen Kategorien auf:

Cookiebanner					
UUID	COMPANY ID	TITLE	LOGO	PRIVACY LINK	CUSTOM TEXTS
adee4629-f8d5-41ef-b975-34ec5a3bcdd5	2	Technische Hochschule Ulm	Hier der Link zur URL	https://www.thu.de/de/Seiten/Homepage.aspx ()	<div> <div>{</div> <div>"default-id": "Williams.com"</div> <div>"id": "Williams.com"</div> <div>"name": "Name Video Two"</div> <div>"post-title": "Williams.com"</div> <div>"privacy-policy": "Privacy Policy"</div> <div>"referrer-url": "referrer"</div> <div>"cookie-consent-type": "html"</div> <div>"show-widget-button": "change"</div> </div>

Neue Kategorie erstellen	

Kategorien					
UUID	NAME	DESCRIPTION	ALWAYS ACCEPTED	Aktionen	
64424546-62d5-4025-b1f8-6c3713e0071	Statusk	Erfassung von anonymisierten Daten für Statistiken. Mit Hilfe dieser Cookies können wir beispielsweise die Besucherzahlen und den Effekt bestimmter Seiten unseres Web-Auftritts ermitteln und unsere Inhalte laufend optimieren.	Nein	Services	[edit] [delete]

Abb. 25 - Cookiebanner übersicht

Dabei können wir immer weitere Kategorien erstellen und bestehende bearbeiten und löschen. Um nun nicht alle Services anzeigen zu lassen (was bei einem Realen Kunden



viel sein könnte), wollte mein Chef das jede Kategorie ein Button hat, und erst wenn man auf Kategorie xy drückt, erscheinen nur die Services zu dieser Kategorie.

Das Ganze funktioniert identisch mit den Tabellen Cookies und Service.

UUID	NAME	PARENT	DESCRIPTION	PRIVACY LINK	CALLBACK ON ACCEPT	CALLBACK ON DECLINE	INTERNAL FUNCTION CALL ON ...	INTERNAL FUNCTION CALL ON ...
f5ad9c5-d39f-43af-9dc7-e3b8a1220ea7	google.com	---	Google nutzt hier das und das ...	https://www.planta.de/hartene/pdubertstf/1	Console.log("Test1")	Console.log("Test2")	Console.log("Test3")	Console.log("Test4")

Abb. 26 - Service anzeigen

So ist es nun sehr übersichtlich, sparen uns das Scrollen, weil alles immer kompakt auf der einen Seite geladen wird.

Durch das Framework Elastic UI können wir Tabellen in Fullscreen anzeigen lassen, können Felder sortieren, entfernen und ab 5 Kategorien, Services oder Cookies entsteht hier eine Paginierung.

### 5.3.3 Einbinden eines Cookie Banners

```
<script src="http://localhost:21001/js/cookie/banner.js" data-bite-cookie-consent-id="b6230c7d-5404-4d1b-a0f2-92fae50705e5"></script>
```

Derzeit verweist die Quelle noch auf localhost, was sich natürlich ändert, sobald meine beiden Vorgesetzten das Merge Request geprüft und freigegeben haben. Anschließend wird das gesamte Projekt auf unseren statischen Server hochgeladen. Ab diesem Moment müssen wir nur noch die jeweilige UUID im HTML mitgeben, um den passenden Cookiebanner aufzurufen.

### 5.3.4 Ergebnisse

Nach dem ich am Anfang erklärt hatte, dass das Frontend schon erstellt worden ist von meiner Kollegin, habe ich dann die Info erhalten, dass ich hier auch etwas Anpassen muss, da Sie ihr Kind bekommen hat und für mindestens 1 Jahr nicht mehr arbeiten kann. Hier musste ich sehr wenig ändern, welches ich ganz grob erkläre: Bei meiner Umsetzung habe ich die Konfiguration nicht mehr über eine globale Variable eingebunden, sondern in einer eigenen JSON-Datei hinterlegt. Dadurch lässt sich die Backend-URL zentral verwalten und einfacher anpassen. Um beim Zusammensetzen der Anfrage-URL keine doppelten oder fehlenden Slashes zu erzeugen, habe ich zusätzlich eine kleine Hilfsfunktion geschrieben, die den Basis-Pfad mit der Banner-ID korrekt verbindet. Außerdem habe ich die Fehlerausgabe so erweitert, dass im Falle eines Problems nicht nur eine allgemeine Fehlermeldung erscheint, sondern der konkrete HTTP-Status

zurückgegeben wird. Damit lassen sich Fehler schneller erkennen und gezielt beheben. Abschließend habe ich die Instanz des Cookiebanners sauber über TypeScript an das globale Fensterobjekt gebunden, sodass sie auch außerhalb des Skripts verfügbar ist. Zum Schluss müssen wir ganz am Ende unseres Templates folgenden Code einbinden, damit wir einen Cookiebanner haben:

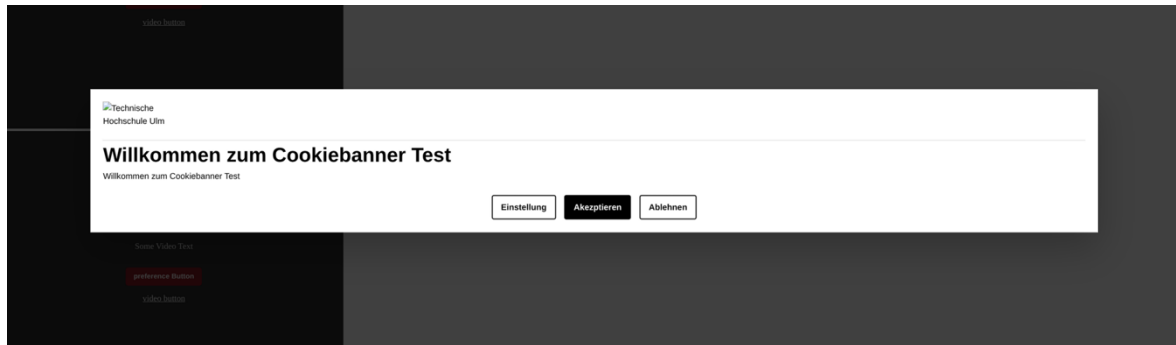


Abb. 27 - Live Darstellung Cookiebanner

So würde das Ganze dann aussehen, wobei ich hier natürlich kein Logo eingefügt habe, da wir das erstmal alles auf unseren Statischen Server hochladen müssen. Den Cookiebanner können wir dann über dem Admin Tool oben mittig oder ganz unten anzeigen lassen. Alle Texte sind auch über den Cookiebanner anpassbar. Klicken wir auf den „Einstellung“ Button, sehen wir unsere ganzen Kategorien, in dem Fall wie ich oben erklärt habe, habe ich nur nur eine Kategorie, Services und Cookie erstellt.

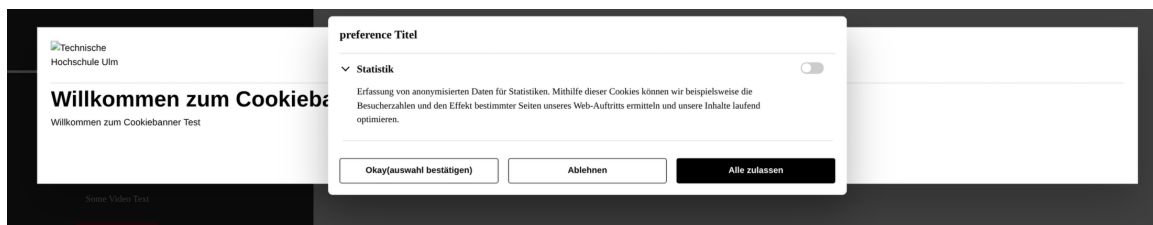


Abb. 28 - Settings Live - Cookiebanner

Klicken wir dann die Kategorie an (den Pfeil), so erscheinen dann die Services zu der jeweiligen Kategorie und klicken wir auf den Service erscheinen dann auch die Cookies dazu.

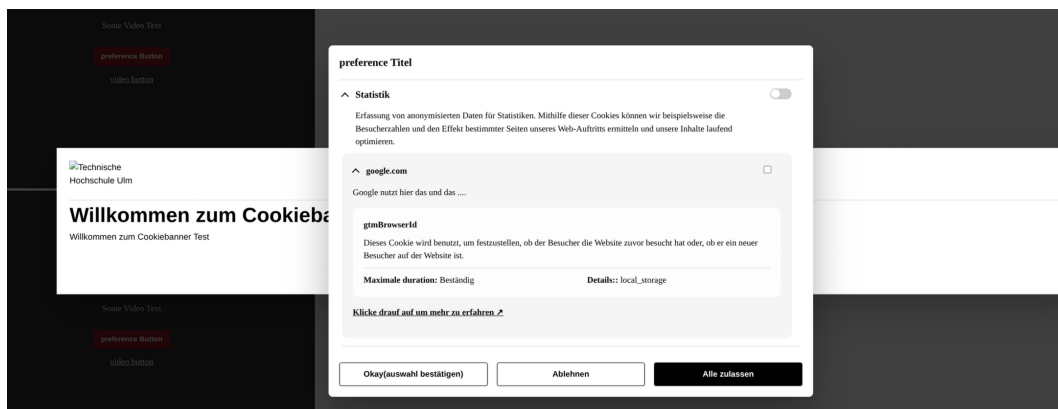


Abb. 29 - Services und Cookie Live

## 6. Kleine Projekte

Neben den drei großen Projekten durfte ich auch kleinere Aufgaben übernehmen. Ein Beispiel dafür war die Erstellung von berechneten Datenfeldern. Zur Erklärung: Wenn ein Kunde unser System nutzt und darin Bewerbungen verwaltet, kann er zusätzlich zahlreiche individuelle Datenfelder anlegen. Diese Felder sind nicht nur einfache Eingaben, sondern können auch mit bestimmten Funktionen verknüpft werden. Dadurch lassen sich Abläufe automatisieren und Informationen dynamisch berechnen, was den Einsatz des Systems noch benutzerfreundlicher und effizienter macht.

Um es besser zu erklären, hier ein einfaches Ticket. Die Anforderung würde so aussehen:

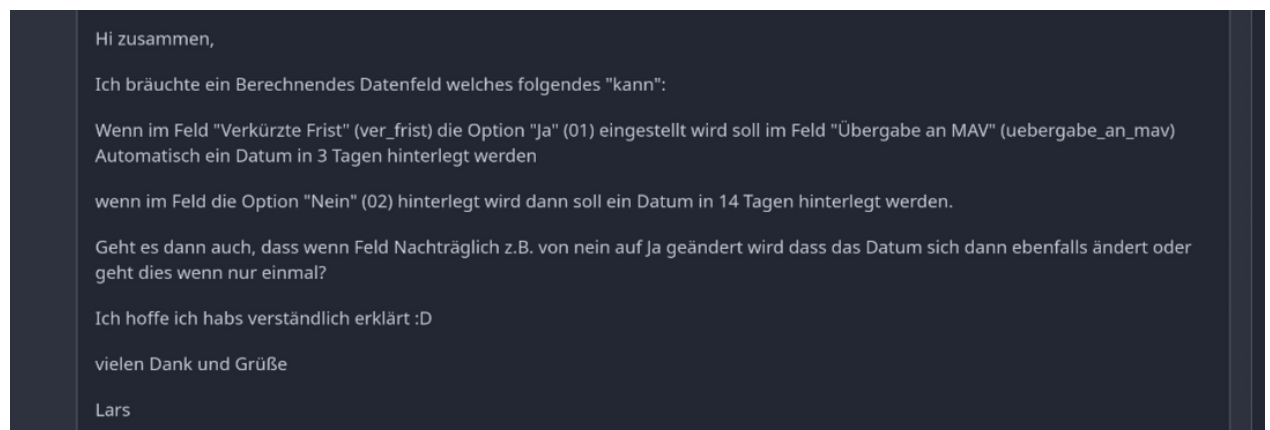


Abb. 30 - Projektsupport Ticket

Dabei habe ich im System berechnete Datenfelder angelegt, die mit JavaScript-Funktionen verknüpft wurden. Zusätzlich habe ich kleinere Aufgaben wie die Bearbeitung von Social-Media-Tickets übernommen, damit unsere Stellenanzeigen auf Plattformen wie LinkedIn korrekt dargestellt werden.

## 7. Fazit

Rückblickend kann ich sagen, dass mir das Praxissemester in vielerlei Hinsicht sehr gutgetan hat und ich persönlich wie auch fachlich enorm davon profitiert habe. Während dieser Zeit durfte ich nicht nur kleinere Aufgaben übernehmen, sondern auch an umfangreichen Projekten mitarbeiten, die für das Unternehmen von echter Relevanz sind. Besonders wertvoll war für mich die Kombination aus Teamarbeit und eigenverantwortlichem Arbeiten. Im Team konnte ich beispielsweise bei API-Integrationen oder der Weiterentwicklung von Templates viel lernen, mich mit Kolleginnen und Kollegen austauschen und gemeinsam Lösungen erarbeiten. Gleichzeitig hatte ich die Möglichkeit, eigenständig ein größeres Projekt wie den Cookiebanner von Grund auf zu planen und umzusetzen. Dieses Projekt war für mich ein Highlight, da ich nicht nur die technische Umsetzung mit Go, React, Vite als Bundle, PostgreSQL, Echo und Elastic UI als Framework übernommen habe, sondern auch ein Gefühl dafür bekommen habe, wie ein Projekt über mehrere Monate hinweg Schritt für Schritt entsteht und wächst. Neben der fachlichen Erfahrung war für mich auch die Arbeitsatmosphäre sehr bereichernd. Ich habe mich in das Team sehr gut integriert gefühlt, konnte jederzeit Fragen stellen und habe viel Unterstützung, aber auch Vertrauen in meine Fähigkeiten bekommen. Besonders gefreut hat mich, dass meine Arbeit sichtbar zum Fortschritt beigetragen hat und von meinem Chef und meinen Vorgesetzten sehr geschätzt wurde. Das hat mir nicht nur Sicherheit, sondern auch Motivation gegeben, immer mein Bestes zu geben und mich neuen Herausforderungen zu stellen. Ein weiterer positiver Aspekt war, dass ich während des Praxissemesters bereits ein Gespräch mit meinem Chef führen durfte, in dem mir eine mögliche Übernahme nach meinem Studium in Aussicht gestellt wurde. Für mich ist das eine große Anerkennung meiner bisherigen Arbeit und zeigt, dass das Unternehmen Vertrauen in meine Fähigkeiten hat. Gleichzeitig motiviert es mich, mein letzten 2 Semester im Studium weiterhin mit voller Energie zu verfolgen und mich noch stärker in komplexe Themen wie Softwarearchitektur, Datenbankdesign, moderne Frontend-Frameworks und Backend Entwicklung einzuarbeiten und mich weiter zu verbessern. Alles in allem blicke ich mit großer Zufriedenheit auf die letzten Monate zurück. Ich habe nicht nur viele neue Technologien kennengelernt und praktische Erfahrungen gesammelt, sondern auch einen Einblick in die Strukturen, Abläufe und Anforderungen eines Softwareunternehmens bekommen. Besonders wichtig ist für mich die Erkenntnis, dass mir die Arbeit in diesem Umfeld großen Spaß macht und ich mir sehr gut vorstellen kann, nach meinem Studium in diesem Bereich meine berufliche Zukunft zu gestalten.

## 8.Abbildungsverzeichnis

Abb. 1 - Ordner Struktur API .....	5
Abb. 2 - Export function Api .....	5
Abb. 3 - Listing.....	6
Abb. 4 - Build Api .....	7
Abb. 5 - Beispiel für eine API-Client und Suchkonfiguration .....	7
Abb. 6 - Beispiel für eine ControlConfig in der Jobs Api .....	8
Abb. 7 - Template Class .....	8
Abb. 8 - App .....	9
Abb. 9 - Live Api Vertical Cloud Solution.....	10
Abb. 10 - Tickets APIs.....	10
Abb. 11- Template Struktur .....	11
Abb. 12 - HTML Aufbau.....	12
Abb. 13 - CSS .....	13
Abb. 14 - Import CSS Datei in Bewerbermanagement .....	14
Abb. 15 - Tickets Templates .....	15
Abb. 16 - HTML Live Kunden Vertical Cloud Solution.....	15
Abb. 17 - UML Diagramm Cookiebanner mit StarUML .....	16
Abb. 18 - Cookiebanner backend Struktur .....	17
Abb. 19 - Struktur Web User Interface .....	19
Abb. 20 - Cookiebanner suchen.....	20
Abb. 21 - Cookiebanner erstellen.....	21
Abb. 22 - Kategorie Erstellen.....	21
Abb. 23 Service erstellen.....	22
Abb. 24 - Filterung nach einem Cookiebanner .....	22
Abb. 25 - Cookiebanner übersicht .....	22
Abb. 26 - Service anzeigen .....	23
Abb. 27 - Live Darstellung Cookiebanner.....	24
Abb. 28 - Settings Live - Cookiebanner .....	24
Abb. 29 - Services und Cookie Live .....	24
Abb. 30 - Projektsupport Ticket.....	25

## 9. Quellenverzeichnis

### Bildquellen:

1. Technische Hochschule Ulm Logo

[https://de.wikipedia.org/wiki/Datei:THU\\_Logo\\_Subline\\_deutsch\\_RGB.png](https://de.wikipedia.org/wiki/Datei:THU_Logo_Subline_deutsch_RGB.png)

2. Logo vom Unternehmen B-ite GmbH

[https://rathaus.io/unsere-integrationen/bite-integration-stellenanzeigen-und-bewerbermanagement\\_id1082](https://rathaus.io/unsere-integrationen/bite-integration-stellenanzeigen-und-bewerbermanagement_id1082)

### Informationsquellen:

Webdriver.io: Component Testing mit Preact, verfügbar unter (13.09.2025):

<https://webdriver.io/de/docs/component-testing/preact/>

Ionos: Golang Knowhow, verfügbar unter (13.09.2025):

<https://www.ionos.de/digitalguide/server/knowhow/golang/>

Ionos: Arch linux, verfügbar unter (10.09.2025):

<https://www.ionos.de/digitalguide/server/konfiguration/arch-linux/>

Abgesehen von einzelner Erklärung zu Arch linux, Preact und Go, die lediglich zur Auffrischung dienten, habe ich keine weiteren Quellen aus dem Internet verwendet.