

# Entwicklung eines Webshop-Konfigurators für Medaillenhalter mit React

Pandza, Luca Maurice

22. März 2025

Fachhochschule Südwestfalen

Studienfach: Moderne Webframeworks

Betreuer: Prof. Dr. Christian Gawron

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>4</b>
1.1 Motivation . . . . .	4
1.2 Zielsetzung . . . . .	5
<b>2 Theoretischer Hintergrund</b>	<b>6</b>
2.1 DXF und die Python Bibliothek ezdxf . . . . .	6
2.2 FastAPI und Uvicorn . . . . .	6
2.3 React . . . . .	7
<b>3 Projektstruktur</b>	<b>9</b>
3.1 Unterstruktur des Frontends . . . . .	10
3.2 Unterstruktur des Backends . . . . .	10
<b>4 Einrichtung und Ausführen der Entwicklungsumgebung</b>	<b>11</b>
4.1 Setup . . . . .	11
4.1.1 Klonen des Projekts . . . . .	11
4.1.2 Backend vorbereiten . . . . .	11
4.1.3 Frontend vorbereiten . . . . .	12
4.2 Überblick der Webseite . . . . .	12
4.2.1 Konfigurationsseite . . . . .	13
4.2.2 Warenkorbseite . . . . .	15
4.2.3 WebAuthn-Registrierung und Login . . . . .	17
<b>5 Implementierung</b>	<b>20</b>
5.1 Projektbeginn . . . . .	20
5.2 Sequenzdiagramm . . . . .	20
5.3 Frontend . . . . .	22
5.3.1 Home.jsx – Startseite mit Call-to-Action . . . . .	22
5.3.2 Navbar.jsx – Navigationsleiste mit dynamischem Warenkorb-Icon . . . . .	22
5.3.3 CartContext.jsx – Globale Verwaltung des Warenkorbs . . . . .	23
5.3.4 Configurator.jsx – Zentrale Konfigurationslogik . . . . .	23
5.3.5 MedalHolder.jsx – SVG-Vorschau des Medaillenhalters . . . . .	23
5.3.6 Warenkorb.jsx – Produktübersicht und Bestellabwicklung . . . . .	24
5.3.7 Kontaktformular.jsx – Validierung und Kontaktdaten für die Bestellung . . . . .	24
5.3.8 ContactContext.jsx – Globale Zustandsverwaltung für Nutzerdaten . . . . .	25
5.4 Backend . . . . .	26
5.4.1 generate_buchstaben.py – Buchstabenerzeugung als DXF-Dateien . . . . .	26
5.4.2 generate_medaillenhalter_dxf_png_svg.py - Erzeugung des Endprodukts . . . . .	26
5.4.3 main.py - Serverlogik . . . . .	28
<b>6 Ausblick</b>	<b>29</b>
<b>7 Fazit</b>	<b>30</b>

# Abbildungsverzeichnis

1	Der im Jahre 2021 entworfene Medaillenhalter aus gelasertem Edelstahl. Als persönliche Motivationsquelle an der Wand montiert. . . . .	4
2	Einfaches Use-Case-Diagramm für das Projekt eines Online-Shops mit Medaillenhalter-Konfigurator . . . . .	5
3	Wichtigste Datei- und Verzeichnisstruktur des Projekts <i>MedaillenKonfigurator</i> mit Trennung in Frontend und Backend (ohne CSS-Dateien) . . . . .	9
4	Startseite mit einem Marketingspruch . . . . .	12
5	Designauswahl „Radfahren“, Eingabe „CYCLING“ und vier Balken . . . . .	13
6	Design „Laufen“ mit drei Balken – Warenkorb zeigt Produktanzahl . . . . .	14
7	Design „Schwimmen“ mit fünf Balken und maximaler Breite von 1000 mm . . . . .	14
8	Tabellarische Auflistung der konfigurierten Produkte im Warenkorb . . . . .	15
9	Vergrößerte Vorschau in der Frontend der serverseitig generierten DXF-Datei als PNG-Datei . . . . .	15
10	Generierte Vorschau für „Laufen“ mit drei Balken . . . . .	16
11	Generierte Vorschau für Design „Schwimmen“ mit fünf Balken . . . . .	16
12	Kontaktformular mit Validierung und optionaler Rechnungsadresse . . . . .	16
13	Kaufbestätigung und automatisches Leeren des Warenkorbs . . . . .	17
14	Kundendatenbank mit gespeicherten Kontaktinformationen . . . . .	17
15	Produktdatenbank mit Konfiguration und Referenz auf Kunden . . . . .	17
16	Registrierungsdialog zur Erstellung eines Passkeys . . . . .	18
17	Registrierung erfolgreich abgeschlossen . . . . .	18
18	WebAuthn-Datenbankeintrag nach erfolgreicher Registrierung . . . . .	18
19	Aufforderung zur Einrichtung eines Passkeys auf einem anderen Gerät . . . . .	19
20	Fehler beim Login trotz vorheriger Registrierung . . . . .	19
21	Einfaches Sequenzdiagramm des Konfigurations- und Bestellprozesses. . . . .	21
22	Warenkorb-Icon mit rotem Zähler für in den Warenkorb abgelegte Produkte . . . . .	22
23	Konsolenausgabe beim Ausführen der Funktion <code>get_medaillehalter(...)</code> . . . . .	27
24	Einfügen der drei Grundformen des Medaillenhalters . . . . .	27
25	Nach Anwendung der Funktion <code>insert_and_mirror()</code> . . . . .	28
26	Endergebnis der Funktion <code>get_medaillehalter(...)</code> . . . . .	28

## Abkürzungsverzeichnis

**API** Application Programming Interfaces

**CAD** Computer Aided Design

**DXF** Drawing Exchange Format

**JSX** JavaScript Syntax Extension

**SPA** Single-Page-Applications

**UI** User Interface

# 1 Einleitung

## 1.1 Motivation

Während der Corona-Pandemie wurden weltweit nahezu alle Sportveranstaltungen abgesagt. Gerade im Jahr 2020 hatte ich mir vorgenommen, sportlich durchzustarten. Ich war für mehrere Laufveranstaltungen angemeldet, doch viele wurden aufgrund der Pandemie kurzfristig gestrichen. Als einzige Alternative blieb mir das sogenannte virtuelle Laufen: Ich absolvierte die geplanten Distanzen alleine zu Hause oder im Freien – ohne Zuschauer, ohne Mitläufer und ohne die sonst so motivierende Atmosphäre.

Trotzdem erhielt ich nach diesen Läufen die jeweiligen Medaillen zugesendet. Mit der Zeit sammelten sich einige davon an. Ich betrachtete sie oft mit gemischten Gefühlen – einerseits stolz, die Distanzen trotzdem geschafft zu haben, andererseits etwas wehmütig, weil das echte Erlebnis mit hunderten Mitläufern und stimmungsvollen Zuschauern gefehlt hatte. Um meine sportliche Motivation aufrechtzuerhalten, beschloss ich mithilfe meiner Kenntnisse aus dem Mechatronik-Studium, einen eigenen Medaillenhalter zu entwerfen.

Mit Fusion360 von Autodesk (ein Programm für Computer Aided Design (**CAD**)) entwickelte ich ein individuelles Design. Ziel war es, einige ausgewählte Medaillen stilvoll aufzuhängen. Nachdem der Entwurf stand, ließ ich den Halter aus Edelstahl laseren. Er misst etwa einen Meter in der Breite und hat nun einen festen Platz an meiner Wand – als tägliche Erinnerung und neue Motivation (siehe Abbildung 1).

Aus dieser persönlichen Erfahrung heraus wuchs die Idee, diesen Prozess zu automatisieren. Ich stellte mir die Frage, wie ein Online-Konfigurator aussehen könnte, mit dem man ganz einfach einen eigenen Medaillenhalter gestalten kann. Besonders spannend fand ich die Idee, dass dabei automatisch die für die Fertigung notwendige Datei als Drawing Exchange Format (**DXF**) im Backend generiert wird. Allerdings hatte ich bisher kaum Erfahrung in der Webentwicklung. Mein letzter Kontakt mit HTML lag einige Jahre zurück und stammt aus dem Schulunterricht. Der Gedanke an einen eigenen kleinen Online-Shop oder Konfigurator reizte mich schon länger, blieb aber bisher Theorie. In meinem beruflichen Alltag arbeite ich mehr mit maschinennaher Software im Bereich Mechatronik und Maschinenbau – Webtechnologien waren Neuland.

Im Rahmen des Moduls *Moderne Webframeworks* bot sich nun endlich die Gelegenheit, diese Idee in einem passenden Umfeld umzusetzen. Mit React im Frontend und FastAPI im Backend standen zwei moderne und gut dokumentierte Technologien zur Verfügung, die auch für Einsteiger geeignet sind. React überzeugte mich durch seinen komponentenbasierten Aufbau und die intuitive Wiederverwendbarkeit von Elementen. FastAPI machte es mir mit meinen Python-Kenntnissen leicht, schnell eine funktionsfähige Backend-Struktur aufzubauen.



Abbildung 1: Der im Jahre 2021 entworfene Medaillenhalter aus gelasertem Edelstahl. Als persönliche Motivationsquelle an der Wand montiert.

## 1.2 Zielsetzung

Ziel dieser Arbeit ist die Entwicklung eines webbasierten Konfigurators für Medaillenhalter. Nutzerinnen und Nutzer sollen die Möglichkeit haben, ihr individuelles Design direkt im Browser zu gestalten. Der gesamte Prozess – von der Eingabe des Wunschtextes über die Auswahl der Halterform bis hin zur Vorschau und finalen Dateigenerierung – soll vollständig digital ablaufen.

Die technische Umsetzung erfolgt im Frontend mit dem JavaScript-Framework React, das eine dynamische und benutzerfreundliche Oberfläche bereitstellt. Im Backend wird FastAPI eingesetzt, um die Application Programming Interfaces ([API](#)) bereitzustellen und die automatische Generierung der entsprechenden CAD-Dateien im .dxf-Format zu ermöglichen. Langfristig könnte der Konfigurator als Grundlage für einen kleinen Online-Shop dienen, in dem individuelle Halter bestellt werden können.

Im Rahmen dieser Arbeit soll ein webbasiertes System zur Konfiguration individueller Medaillenhalter entwickelt werden. Abbildung 2 zeigt die zentralen Anwendungsfälle im Überblick: Nutzerinnen und Nutzer interagieren über das Frontend mit dem System, indem sie Parameter wie Wunschtext, Breite (bis 1000 mm) und Anzahl der Aufhängungen (2, 3 oder 4) eingeben. Die Konfiguration wird anschließend an das Backend übermittelt, welches die Kunden- und Produktdaten speichert und daraus automatisch eine CAD-Datei im .dxf-Format erzeugt. Die klare Trennung zwischen Frontend und Backend sowie der automatisierte Ablauf bilden die Grundlage für eine durchgängig digitale Fertigungskette.

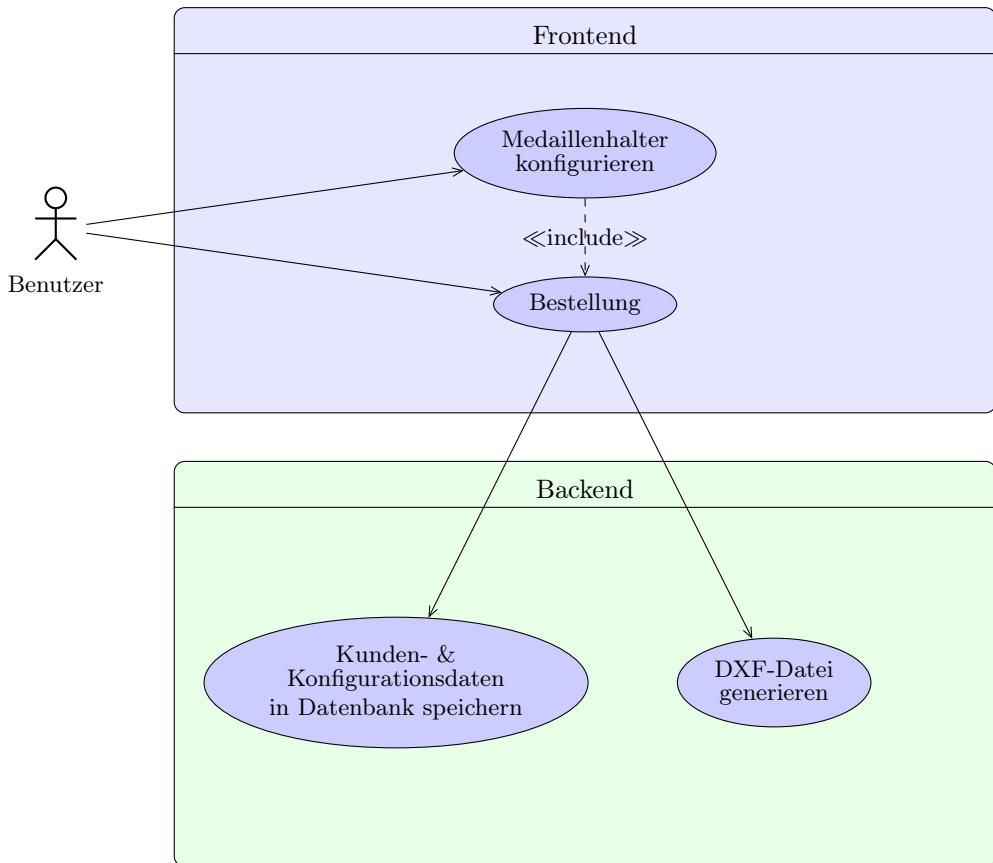


Abbildung 2: Einfaches Use-Case-Diagramm für das Projekt eines Online-Shops mit Medaillenhalter-Konfigurator

## 2 Theoretischer Hintergrund

### 2.1 DXF und die Python Bibliothek `ezdxf`

Im Rahmen der Konstruktion des Medaillenhalters kam das CAD-Dateiformat `.dxf` (Drawing Exchange Format) zum Einsatz. Bereits während der ersten Entwurfsphasen in Autodesk Fusion360 wurde dieses Format verwendet, um die Zeichnungen direkt für den Fertigungsprozess – das Laserschneiden des Edelstahlblechs – bereitzustellen. Die Möglichkeit, DXF-Dateien unkompliziert an Fertigungsdienstleister zu übergeben, erwies sich dabei als großer Vorteil.

Technisch gesehen handelt es sich bei DXF um ein von Autodesk entwickeltes und offen dokumentiertes Austauschformat für CAD-Zeichnungen. Ursprünglich wurde es für die Verwendung in AutoCAD entwickelt und ist heute ein verbreiteter Standard zur Weitergabe von Zeichnungsdaten an andere CAD-Programme oder Fertigungssysteme. Eine DXF-Datei kann wahlweise im ASCII- oder im Binärformat gespeichert werden. Während die ASCII-Version lesbar und editierbar ist, überzeugt das Binärformat durch eine geringere Dateigröße und schnellere Verarbeitung (vgl. [autodesk2025](#)).

Für den Medaillenhalter reichen zweidimensionale Zeichnungen vollständig aus. Die Dateien enthalten die Umrisse von Buchstaben sowie symbolischen Figuren wie Läufern, Radfahrern oder Schwimmern. Auch Bohrungen für die Wandbefestigung werden in DXF abgebildet. Bereits bei der Konstruktion muss jedoch beachtet werden, dass der Laserschneider bestimmte minimale Radien benötigt, um Formen exakt ausführen zu können. Zu kleine Details können unter Umständen nicht gefertigt werden. Auch die Materialdicke spielt eine entscheidende Rolle – diese wird beim Fertigungsdienstleister separat angegeben und ist nicht direkt Bestandteil der DXF-Datei.

Neben geometrischen Anforderungen sind auch statische Aspekte zu berücksichtigen: Da Medaillen ein gewisses Eigengewicht mitbringen, sollte die Grundform des Halters bestimmte Mindestbreiten nicht unterschreiten. Bereiche mit hoher mechanischer Belastung – etwa Stellen mit mehreren Haken – müssen ausreichend stabil konstruiert sein. Knickstellen oder unnötig dünne Verbindungen sollten vermieden werden. Zusätzlich sollten Optionen zur Wandbefestigung bereits in der Zeichnung berücksichtigt werden.

Im Projektverlauf wurden zur fehlerfreien Gestaltung des Medaillenhalters vorkonstruierte DXF-Bausteine modular zusammengesetzt. Die Grundform des Halters besteht aus drei Einzellementen, die abhängig von der gewünschten Breite über einen definierten Versatz miteinander verbunden werden. Die dazwischenliegenden Lücken werden durch automatisch erzeugte Linien geschlossen. Zusätzlich existieren mehrere Varianten der Grundform mit unterschiedlichen Höhen, die jeweils zwei, drei oder vier Aufhängeebenen für Medaillen bieten. Für ein harmonisches Gesamtbild wird der Halter schließlich symmetrisch gespiegelt.

Für die automatisierte Generierung von DXF-Dateien im Backend dieser Anwendung wird die Python-Bibliothek `ezdxf` verwendet. Diese erlaubt es, DXF-Dateien zu erstellen, zu bearbeiten und zu speichern – ohne ein grafisches CAD-Interface. Die Bibliothek richtet sich explizit an Entwicklerinnen und Entwickler und abstrahiert viele technische Details des DXF-Formats, ohne dabei an Flexibilität zu verlieren (vgl. [ezdxf2025](#)).

Die Bibliothek unterstützt sowohl das Lesen als auch das Schreiben in verschiedenen Versionen des DXF-Formats (ASCII- und Binärformat), darunter R12, R2000, R2018 und viele weitere. Neben dem Schreiben von DXF-Dateien bietet `ezdxf` auch Add-ons zur Umwandlung in andere Formate wie SVG oder PNG, was insbesondere für die Vorschau im Webbrowser genutzt werden kann. Die Nutzung dieser Bibliothek ermöglichte es, ohne tiefgehende Kenntnisse der DXF-Spezifikation, eine funktionierende und erweiterbare Backend-Lösung zu implementieren.

Auch die Schriftzüge für die Medaillenhalter werden aus vorgefertigten DXF-Buchstaben zusammengesetzt. Diese wurden zuvor von einer Schriftart zu einzelne DXF-Einzeldateien umgewandelt und in das Verzeichnis `Buchstaben` abgelegt. Zur Laufzeit erfolgt dann die Kombination der entsprechenden Buchstaben zu einem Gesamttext. Figuren wie Läufer oder Schwimmer wurden ebenfalls manuell vorgezeichnet und in das Verzeichnis `dxf_vorlagen` abgelegt. Das Zusammensetzen dieser Einzelbestandteile übernimmt das Python-Backend mithilfe von `ezdxf` automatisiert.

### 2.2 FastAPI und Uvicorn

Die serverseitige Zusammensetzung der Medaillenhalter, inklusive Schriftzügen und Symbolen, erfolgt mithilfe der Python-Bibliothek `ezdxf`. Während die DXF-Dateien für Buchstaben und Figuren vorab im Backend abgelegt werden, ist für das dynamische Generieren, Verarbeiten und Ausliefern dieser Daten eine geeignete serverseitige Infrastruktur notwendig. Diese Aufgabe übernimmt im

vorliegenden Projekt das Webframework *FastAPI* in Kombination mit dem asynchronen Server *Uvicorn*.

FastAPI ist ein modernes, auf Python 3.6+ basierendes Framework, das sich insbesondere durch seine Typhintweise, integrierte Validierung mit Pydantic sowie automatische API-Dokumentation (via Swagger UI) auszeichnet. Durch diese Merkmale eignet sich FastAPI besonders gut für Projekte, in denen strukturierte Nutzereingaben verarbeitet und als Grundlage für algorithmisch generierte Inhalte genutzt werden sollen – wie es bei der DXF-Komposition der Fall ist (vgl. **fastapi2025**). Im Projekt übernimmt FastAPI unter anderem die Entgegennahme der Konfigurations- und Kundendaten, die Validierung der Eingaben sowie die Rückgabe von Preisinformationen, Mindestgrößenangaben für den Medaillenhalter in Abhängigkeit verwendeter Textzeichen und generierter Vorschau- oder Produktionsdateien im DXF-, PNG- oder SVG-Format. Die Schnittstellen wurden so gestaltet, dass sie modular erweitert werden können.

Zum Einsatz kommt dabei Uvicorn als ASGI-kompatibler Server. Der ASGI-Standard (Asynchronous Server Gateway Interface) bildet die Grundlage für moderne Python-Webanwendungen mit Unterstützung für parallele Prozesse, WebSockets und Hintergrundaufgaben. Uvicorn ermöglicht eine hohe Laufzeiteffizienz und eine schnelle Serverreaktion – insbesondere im Zusammenspiel mit FastAPI, das vollständig asynchron aufgebaut ist (vgl. **uvicorn2025**). Dadurch lassen sich auch größere Datenmengen, etwa beim Absenden umfangreicher Konfigurationen oder bei der Übergabe von Benutzerinformationen, performant verarbeiten.

Die Kombination aus FastAPI und Uvicorn stellt im Projekt einen optimalen Kompromiss zwischen Einfachheit und Leistungsfähigkeit dar. Während Frameworks wie Flask oder Django häufig entweder zu minimalistisch (Flask) oder zu komplex (Django) für schlanke, datengetriebene APIs sind, bietet FastAPI bereits „out of the box“ viele relevante Funktionen – inklusive Typsicherheit, Datenvalidierung und automatisierter Dokumentation. Dies ermöglicht nicht nur eine saubere Strukturierung des Codes, sondern auch eine effizientere Fehlersuche und Weiterentwicklung (vgl. **datascientest2025**).

Auch aus didaktischer Perspektive ist FastAPI besonders interessant: Die klare Trennung von Endpunkten, Logik und Datenstruktur sowie die explizite Typisierung mittels Python Typannotationen erhöhen nicht nur die Lesbarkeit des Codes, sondern tragen auch dazu bei, komplexe Anwendungen besser zu strukturieren. Dies ist insbesondere für Einsteiger hilfreich, da die zugrunde liegende REST-Architektur von FastAPI konsequent auf eine saubere Abbildung von Ressourcen auf HTTP-Methoden ausgelegt ist – ein Ansatz, der sich in der modernen Backend-Entwicklung etabliert hat (vgl. **gawron2020**, S. 38f.).

Wie Gawron beschreibt, ermöglichen RESTful APIs eine entkoppelte Kommunikation zwischen Frontend und Backend, wodurch etwa Bestellungen oder Konfigurationen als JSON-Daten transportiert und unabhängig verarbeitet werden können. Die Verwendung von standardisierten URIs und HTTP-Methoden schafft dabei nicht nur Übersichtlichkeit, sondern vereinfacht auch die Testbarkeit der Schnittstellen. Gerade Letzteres stellt in der Praxis einen Vorteil gegenüber klassischen User Interface (**UI**)-Tests dar, da definierte Schnittstellen automatisiert überprüft werden können (vgl. **gawron2020**, S. 39).

FastAPI nutzt dieses Paradigma auf effektive Weise, indem es nicht nur eine einfache Definition der Endpunkte erlaubt, sondern auch direkt validierbare Datenstrukturen über sogenannte Pydantic-Modelle bereitstellt. Diese Integration von Datenprüfung und Endpunktdefinition in einem konsistenten Konzept ist exemplarisch für die Prinzipien moderner Webframeworks, die laut Gawron durch eine einfache Verbindung zwischen Datenmodell, Darstellung und Steuerlogik gekennzeichnet sind (vgl. **gawron2020**, S. 38).

Die im Projekt verwendete `main.py` definiert dabei alle Endpunkte der Anwendung: von der Generierung visueller Vorschauen über die Speicherung von Bestellungen bis hin zur Authentifizierung mit WebAuthn. Die zentrale FastAPI-Instanz wird über Uvicorn lokal auf Port 8000 betrieben und ist damit für das React-Frontend, das auf Port 3000 läuft, via HTTP erreichbar. Durch diese Architektur ergibt sich eine klare Trennung zwischen Datenverarbeitung und Benutzeroberfläche – bei gleichzeitig reibungsloser Kommunikation über standardisierte REST-APIs.

## 2.3 React

Da die serverseitige Verarbeitung somit zuverlässig gekapselt ist, liegt der Fokus der Benutzerinteraktion auf der Clientseite. Für deren Umsetzung wurde React als zentrale Frontend-Technologie eingesetzt.

React ist eine weit verbreitete JavaScript-Bibliothek zur Entwicklung komponentenbasierter Benutzeroberflächen. Sie wurde ursprünglich von Facebook (heute Meta) entwickelt und hat sich seither zu einem de facto Standard für moderne Frontend-Entwicklung etabliert (vgl. **legacyreact**).

Besonders hervorzuheben ist die hohe Flexibilität von React, die unter anderem durch den unidirektionalen Datenfluss und die enge Integration von Logik und Darstellung in Komponenten ermöglicht wird (vgl. [gawron2020](#)).

Im Gegensatz zu anderen Frameworks wie Angular oder Vue verfolgt React eine andere Interpretation des Konzepts der Separation of Concerns: Anstelle einer Trennung von HTML, CSS und JavaScript erfolgt die Trennung auf Ebene der Komponenten selbst. Eine React-Komponente kapselt sowohl die Darstellung mittels JavaScript Syntax Extension ([JSX](#)) als auch die Logik ihrer Funktionseinheit. Dies ermöglicht eine saubere Gliederung in wiederverwendbare und testbare Einheiten, die sich auch für komplexe Anwendungen eignen (vgl. [gawron2020](#)).

Ein zentrales Element von React ist die Verwendung von JSX – einer Syntaxerweiterung für JavaScript, mit der HTML-Elemente direkt im JavaScript-Code definiert werden können. Diese Integration ermöglicht eine deklarative Beschreibung des UI und verbessert die Lesbarkeit sowie die Wartbarkeit der Komponentenstruktur. Der sogenannte Virtual DOM, eine Repräsentation der Benutzeroberfläche im Speicher, sorgt zusätzlich für eine performante Aktualisierung der Anzeige, da nur die tatsächlich veränderten Komponenten erneut gerendert werden (vgl. [hygraph2024](#)).

Der Datenfluss in React ist grundsätzlich unidirektional. Das bedeutet, dass Daten ausschließlich von übergeordneten zu untergeordneten Komponenten weitergereicht werden. Interaktionen von unten nach oben erfolgen über Callbacks, die beispielsweise als Props weitergegeben werden. Dieses Prinzip erhöht die Vorhersehbarkeit und Nachvollziehbarkeit der Datenflüsse in einer Anwendung und wird insbesondere bei größeren Projekten geschätzt (vgl. [gawron2020; radix2025](#)).

Für Zustandsverwaltung innerhalb von Komponenten verwendet React sogenannte Hooks, insbesondere `useState`. Diese ermöglichen eine einfache und zugleich effektive Implementierung reaktiver UI-Elemente. Für globale Zustände, wie etwa Benutzerdaten oder Warenkorbinhalte, bietet React zusätzlich die `Context API`. In der vorliegenden Umsetzung wurde diese Funktion gezielt eingesetzt: Die Datei `CartContext.jsx` verwaltet sämtliche Informationen rund um den Warenkorb, während `ContactContext.jsx` für die Erfassung und Validierung von Kontaktinformationen verantwortlich ist. Durch die Bereitstellung dieser Daten in Kontextobjekten entfällt das sogenannte „Prop Drilling“ – also die mühsame Weitergabe von Daten durch viele verschachtelte Komponenten hinweg.

Zur Navigation innerhalb der Anwendung kommt in diesem Projekt die Bibliothek `react-router-dom` zum Einsatz. Sie ermöglicht ein clientseitiges Routing zwischen verschiedenen Seiten wie Start, Konfigurator, Warenkorb oder Nutzerprofil – ohne dass dabei ein vollständiges Neuladen der Seite erforderlich ist. Dieses Verhalten verbessert nicht nur die Nutzererfahrung, sondern unterstützt auch eine klare logische Trennung der funktionalen Bereiche im Frontend.

Die Entscheidung für React fiel dabei nicht nur aus technischer Perspektive. Auch unter didaktischen Gesichtspunkten stellte sich React als besonders einsteigerfreundlich heraus. Im Vergleich zu komplexeren Frameworks wie Angular bietet React eine flachere Lernkurve, insbesondere für Entwicklerinnen und Entwickler mit grundlegenden JavaScript-Kenntnissen (vgl. [kinsta2023; radix2025](#)). Durch das einfache Setup mit `create-react-app`, die umfangreiche Dokumentation sowie eine große Entwickler-Community konnte das Projekt effizient und verständlich realisiert werden – trotz begrenzter Vorerfahrung in der Webentwicklung. Insgesamt war React für die Umsetzung des Medaillenhalter-Konfigurators eine geeignete Wahl. Die Kombination aus modularer Komponentenstruktur, zentraler Zustandsverwaltung über Context und flexibler Seitenarchitektur durch Routing bildete eine stabile Grundlage für eine moderne, interaktive und gut wartbare Webanwendung.

### 3 Projektstruktur

Die Anwendung *Medaillenhalter* ist in zwei zentrale Bestandteile gegliedert: das *Backend* und das *Frontend*. Diese klare Trennung folgt dem Prinzip der Aufgabenverteilung in modernen Webanwendungen und erleichtert sowohl Entwicklung als auch Wartung. Das Backend für die serverseitige Funktionalitäten befindet sich im Verzeichnis `backend`, während sich das Frontend für die browserbasierte Benutzeroberfläche im Verzeichnis `frontend` befindet. Abbildung 3 zeigt die wichtigsten Dateien und Verzeichnisse der Anwendung. Einige Dateien wie CSS-Dateien und generische Build-Ordner wurden zur besseren Lesbarkeit ausgeblendet.

```
Medaillenhalter/
├── .git/
└── frontend/
    ├── node_modules/
    ├── public/
    │   ├── index.html
    │   └── robots.txt
    ├── src/
    │   ├── img/
    │   ├── components/
    │   │   ├── Home.jsx
    │   │   ├── Navbar.jsx
    │   │   ├── Configurator.jsx
    │   │   ├── MedalHolder.jsx
    │   │   ├── About.jsx
    │   │   ├── Warenkorb.jsx
    │   │   ├── Kontakformular.jsx
    │   │   ├── Login.jsx
    │   │   └── Profil.jsx
    │   ├── contexts/
    │   │   ├── CartContext.jsx
    │   │   └── ContactContext.jsx
    │   ├── App.js
    │   └── index.js
    └── package.json
└── backend/
    ├── Buchstaben/
    │   ├── A.dxf
    │   ├── letter_widths.txt
    │   └── sf-florencesans-sc-exp.ttf
    ├── Warenkorb/
    │   ├── DXF/
    │   │   └── TEXT_Design_AnzahlEbenen.dxf
    │   ├── PNG/
    │   │   └── TEXT_Design_AnzahlEbenen.png
    │   └── SVG/
    │       └── TEXT_Design_AnzahlEbenen.svg
    ├── extern/
    ├── venv/
    ├── main.py
    ├── generate_buchstaben_dxf.py
    ├── generate_medaillenhalter_dxf_png_svg.py
    ├── MedaillenDatenbank.db
    └── requirements.txt

```

README.md

Abbildung 3: Wichtigste Datei- und Verzeichnisstruktur des Projekts *MedaillenKonfigurator* mit Trennung in Frontend und Backend (ohne CSS-Dateien)

### 3.1 Unterstruktur des Frontends

Das Frontend der Anwendung befindet sich im Verzeichnis `frontend/` und wurde mit dem Framework *React* realisiert. Es folgt einer typischen Struktur moderner Single-Page-Applications ([SPA](#)), bei der die funktionale Gliederung in Komponenten eine zentrale Rolle spielt. Die Ordnerstruktur lässt sich in drei Hauptebenen unterteilen:

- `public/`: Dieses Verzeichnis enthält statische Dateien, die direkt vom Browser geladen werden können. Dazu gehören unter anderem die zentrale HTML-Datei `index.html` sowie Hintergrundbilder. Diese Dateien werden nicht durch Webpack verarbeitet und stehen unmittelbar beim Laden der Anwendung zur Verfügung.
- `src/`: Dies ist der zentrale Arbeitsbereich der Anwendung. Hier befinden sich alle Quelltexte und Ressourcen, die für das React-Frontend benötigt werden. Die Struktur ist dabei wie folgt untergliedert:
  - `img/`: Beinhaltet zusätzliche Bilddateien, die zur Gestaltung der Benutzeroberfläche verwendet werden, z. B. für Hintergründe oder Icons.
  - `components/`: In diesem Ordner befinden sich sämtliche React-Komponenten, welche die logischen und visuellen Bausteine der Benutzeroberfläche darstellen. Dazu zählen unter anderem `Home.jsx` (Startseite), `Configurator.jsx` (Konfiguratoransicht), `Navbar.jsx` (Navigationsleiste), `MedalHolder.jsx` (Darstellung des SVG-Halters) und `Warenkorb.jsx` (Bestellübersicht). Zusätzlich sind zu jeder dieser Komponenten zugehörige CSS-Dateien hinterlegt, die das visuelle Erscheinungsbild definieren.
  - `contexts/`: Dieses Verzeichnis enthält die beiden zentralen Context-Dateien `CartContext.jsx` und `ContactContext.jsx`. Über die React Context API ermöglichen sie eine globale Zustandsverwaltung innerhalb der Anwendung, insbesondere für Warenkorb-Inhalte und die im Kontaktformular erfassten Nutzerdaten.
  - `App.js`, `App.css`, `index.js`: Diese Dateien definieren die Grundstruktur der Anwendung. `App.js` enthält das Routing zwischen den Seiten, `App.css` globale CSS-Klassen (etwa für Hintergrundbilder) und `index.js` initialisiert die gesamte React-Anwendung im Browser.
- `node_modules/`: Dieses automatisch generierte Verzeichnis beinhaltet sämtliche Abhängigkeiten, die über den Node Package Manager (npm) installiert wurden. Es ist nicht Teil der aktiven Entwicklung, wird jedoch zur Laufzeit benötigt.

Die gewählte Struktur ist klar gegliedert und entspricht bewährten Standards in der React-Entwicklung. Sie erlaubt eine modulare Weiterentwicklung sowie eine einfache Wartbarkeit. Eine detaillierte Betrachtung der einzelnen Komponenten erfolgt im Abschnitt [5.3](#).

### 3.2 Unterstruktur des Backends

Das Backend der Anwendung befindet sich im Verzeichnis `backend/` und wurde mit dem Webframework *FastAPI* in Kombination mit dem Server *Uvicorn* umgesetzt. Es übernimmt die gesamte serverseitige Logik: von der Generierung der Medaillenhalter-Dateien über die Verarbeitung von Bestellungen bis hin zur Verwaltung der Benutzerdatenbank. Die Backendstruktur ist funktional gegliedert und erlaubt so eine klare Trennung der Aufgabenbereiche:

- `main.py`: Zentrale Steuerdatei der FastAPI-Anwendung. Hier werden sämtliche Routen definiert, darunter die API-Endpunkte zur Vorschau-Generierung (SVG, PNG, DXF), zum Speichern von Bestellungen sowie zur Authentifizierung via WebAuthn. Die FastAPI-Instanz wird hier erstellt und über Uvicorn auf Port 8000 ausgeführt.
- `generate_buchstaben_dxf.py`: Dieses Skript dient der Vorbereitung der Buchstaben-DXF-Dateien. Es konvertiert eine installierte Schriftart automatisiert in einzelne .dxf-Dateien für jeden der 26 Großbuchstaben (A–Z) und speichert diese sowie Informationen über ihre jeweilige Größe als Textdatei im Unterverzeichnis `Buchstaben/`.
- `generate_medaillenhalter_dxf_png_svg.py`: Hauptskript zur dynamischen Generierung der Medaillenhalter. Basierend auf Benutzereingaben (z. B. Name, Design, Anzahl der Ebenen) meldet es je nach Länge des vom Nutzer eingegebenen Texts Abmessungen des Medaillenhalters zurück oder erzeugt eine DXF-, PNG- und SVG-Datei, die anschließend im jeweiligen Unterverzeichnis gespeichert werden.

- **Buchstaben/**: Enthält die vorbereiteten DXF-Buchstaben sowie Zusatzinformationen:
  - **A.dxf – Z.dxf**: Einzelbuchstaben als DXF-Dateien.
  - **letter\_widths.txt**: Enthält Informationen über die Breite jedes Buchstabens, zur Berechnung der Mindestgröße des Medaillenhalters und korrekten Platzierung.
  - **sf-florencesans-sc-exp.ttf**: Die zur Generierung verwendete Schriftart.
- **Warenkorb/**: Hier werden die generierten Produktdaten je Bestellung abgelegt:
  - **DXF/, PNG/, SVG/**: Enthalten die Medaillenhalter-Dateien im jeweiligen Format. Die Dateinamen kodieren relevante Informationen zur Bestellung.
- **MedaillenDatenbank.db**: SQLite-Datenbank zur Speicherung der Bestellungen. Sie enthält neben Kundendaten auch Verweise auf die zugehörigen Konfigurationsdateien.
- **extern/**: Beinhaltet externe Quellcodes, insbesondere für die Integration von WebAuthn. Dieses Verzeichnis wurde dem Open-Source-Repository von *duo-labs* ([https://github.com/duo-labs/py\\_webauthn.git](https://github.com/duo-labs/py_webauthn.git)) übernommen.
- **venv/**: Virtuelle Python-Umgebung mit allen projektspezifischen Abhängigkeiten. Sie wird automatisch durch `python -m venv` generiert und ist nicht Bestandteil der Quellcodeverwaltung.
- **requirements.txt**: Listet alle benötigten Python-Bibliotheken auf, u. a. **fastapi**, **uvicorn**, **pydantic**, **ezdxf** und **sqlalchemy**. Diese Datei dient zur Reproduktion der Entwicklungsumgebung.

## 4 Einrichtung und Ausführen der Entwicklungsumgebung

### 4.1 Setup

Die lokale Ausführung der Anwendung erfolgt in zwei getrennten Schritten: Zunächst wird das Python-Backend eingerichtet und gestartet, danach folgt das Starten des React-Frontends. Die beiden Teile kommunizieren über HTTP, wobei das Backend standardmäßig auf Port 8000 und das Frontend auf Port 3000 läuft.

#### 4.1.1 Klonen des Projekts

1. Zunächst wird das Projekt lokal geklont:

```
git clone https://github.com/Luca99410/MedaillenhalterKonfigurator.git
```

#### 4.1.2 Backend vorbereiten

2. Nach dem Klonen des Projekts in das Verzeichnis für das Backend wechseln:

```
cd MedaillenhalterKonfigurator/backend
```

3. Für die WebAuthn-Funktionalität ist ein zusätzliches Modul in dem Ordner `backend/extern/` erforderlich, das nicht automatisch durch das Klonen des Projekts Medaillenhalter mitgeladen wird. Es muss daher manuell ergänzt werden:

```
git clone https://github.com/duo-labs/py_webauthn.git extern
```

4. Die benötigten Python-Abhängigkeiten werden über die Datei `requirements.txt` installiert:

```
pip install -r requirements.txt
```

5. Anschließend wird eine virtuelle Umgebung aktiviert:

```
# Unter Linux/macOS:  
source venv/bin/activate  
  
# Unter Windows:  
.\\venv\\Scripts\\activate
```

6. Zum Starten des Servers genügt folgender Befehl:

```
uvicorn main:app --reload
```

Die Option `--reload` sorgt dafür, dass der Server bei jeder Codeänderung automatisch neu gestartet wird – was die Entwicklung deutlich vereinfacht.

#### 4.1.3 Frontend vorbereiten

7. Für das React-Frontend wird zunächst sichergestellt, dass das Backend bereits aktiv ist. Zudem müssen `Node.js` und `npm` lokal installiert sein. Danach wechselt man in das Frontend-Verzeichnis:

```
cd ../frontend
```

8. Die Abhängigkeiten werden installiert mit:

```
npm install
```

9. Der Entwicklungsserver wird gestartet mit:

```
npm start
```

10. Anschließend ist die Anwendung über `http://localhost:3000` im Browser erreichbar.

## 4.2 Überblick der Webseite

Nachdem alle Schritte aus Abschnitt 4.1 erfolgreich durchlaufen sind, kann die Anwendung über `http://localhost:3000` im Browser genutzt werden. Der Einstieg erfolgt über die Startseite, auf der ein großformatiger Marketingspruch potenzielle Nutzer zur Gestaltung ihres individuellen Medaillenhalters animieren soll.



Abbildung 4: Startseite mit einem Marketingspruch

#### 4.2.1 Konfigurationsseite

Über den Button der Startseite oder der Navigationsleiste gelangt man zur zentralen Komponente der Anwendung – dem Konfigurator. Hier wird standardmäßig ein Medaillenhalter im Design „Laufen“ mit dem Text TEXT und drei Balken angezeigt. Die grafische Vorschau wird direkt im Browser als SVG gerendert. Diese visuelle Vorschau dient der Orientierung, basiert jedoch nicht auf der finalen DXF-Datei. Letztere wird erst erzeugt, sobald der Nutzer das Produkt dem Warenkorb hinzufügt.

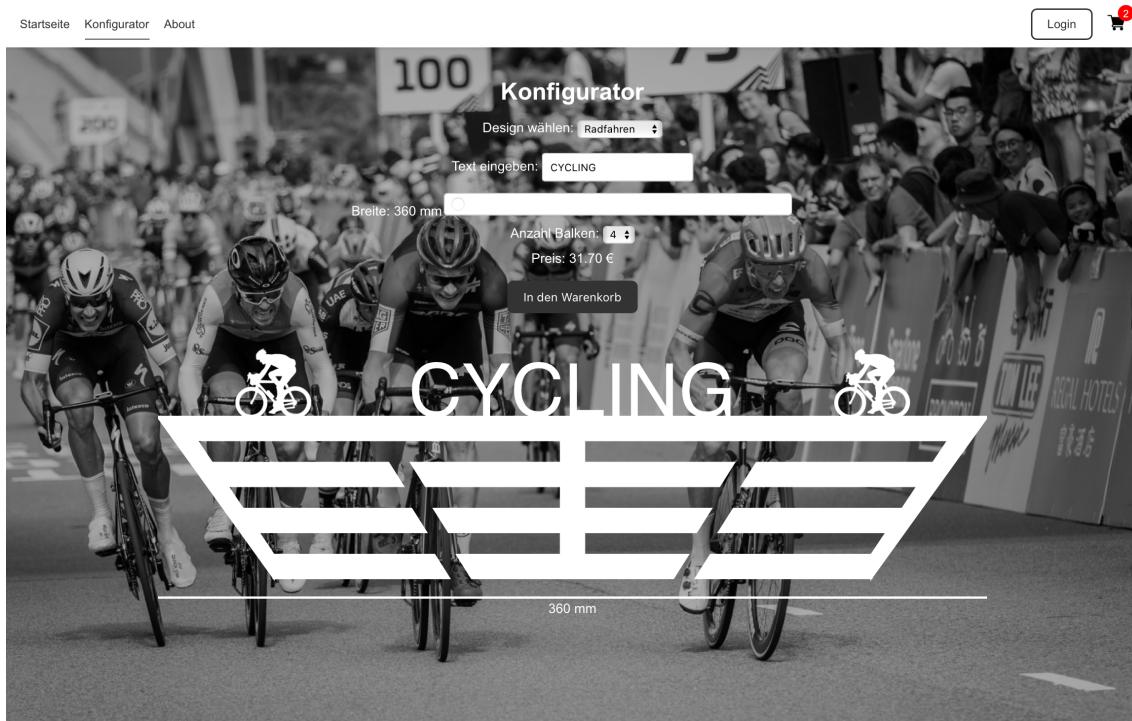


Abbildung 5: Designauswahl „Radfahren“, Eingabe „CYCLING“ und vier Balken

Die Konfiguration ist interaktiv. Änderungen an Text, Design, Breite oder Balkenzahl führen unmittelbar zu einer Aktualisierung der Vorschau. Besonders bemerkenswert ist die Preisberechnung, die vollständig im Backend erfolgt. Damit wird vermieden, dass sensible Preisdaten im Browser einsehbar sind. Gleichzeitig wird im Backend auch die technisch notwendige Mindestbreite des Halters bestimmt, basierend auf den Buchstabenbreiten in der Schriftart. Diese Logik verhindert zu enge Gestaltungen, erlaubt jedoch eine Breitenvergrößerung bis maximal 1000 mm.

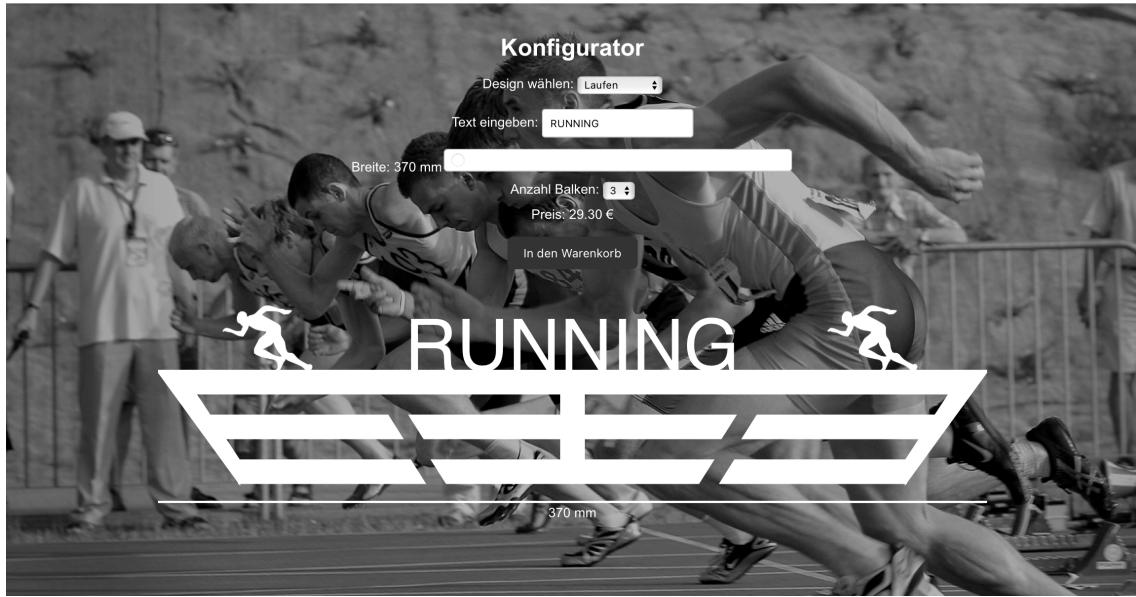


Abbildung 6: Design „Laufen“ mit drei Balken – Warenkorb zeigt Produktanzahl

Im Beispiel aus Abbildung 6 sieht man, dass bei Eingabe des Textes RUNNING ein dreibalkiger Halter konfiguriert wurde. Im roten Kreis rechts oben ist zudem sichtbar, dass der Warenkorb um ein Produkt erhöht wurde.

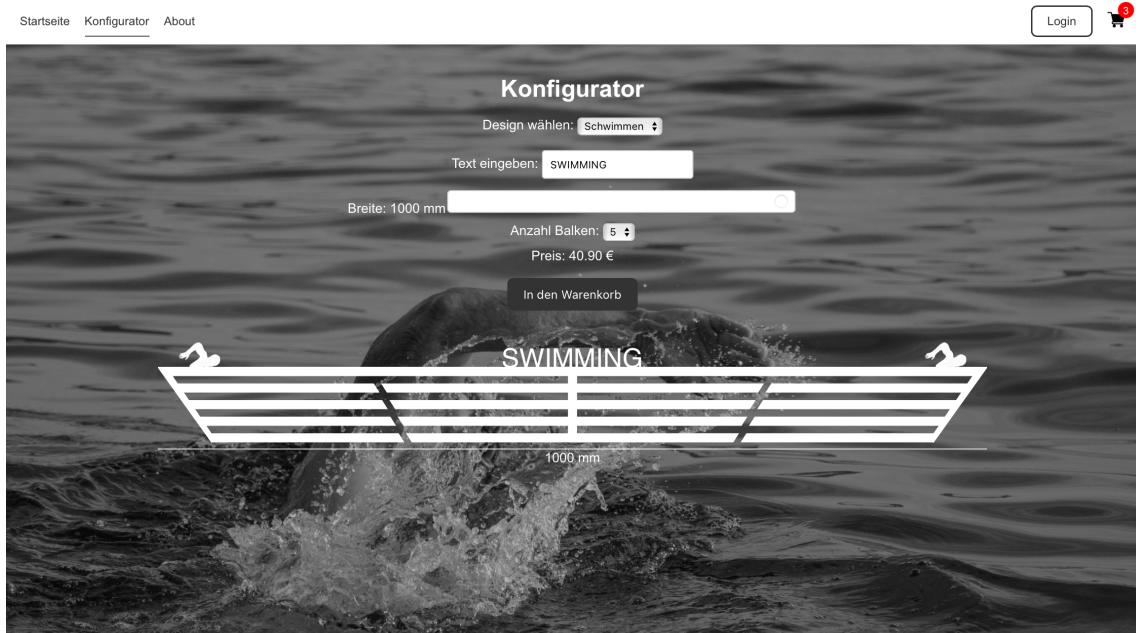


Abbildung 7: Design „Schwimmen“ mit fünf Balken und maximaler Breite von 1000 mm

Abbildung 7 zeigt ein Beispiel für das maximale Maß: Ein Medaillenhalter mit dem Design „Schwimmen“, fünf Balken und der größtmöglichen Breite von 1000 mm. Trotz der hohen Breite bleibt die Darstellung flüssig und interaktiv – ein Vorteil der SVG-Darstellung in Kombination mit React.

Nach erfolgreicher Konfiguration lassen sich Produkte dem Warenkorb hinzufügen. Dieser listet alle gespeicherten Konfigurationen tabellarisch auf. Jede Zeile zeigt Text, Design, Breite, Balkenzahl und eine Miniaturvorschau. Die Vorschauen basieren nun auf serverseitig generierten DXF-Dateien, die als PNG gerendert und an das Frontend übermittelt wurden (vgl. Abschnitt 3 zur Dateistruktur).

#### 4.2.2 Warenkorbseite

Warenkorb							
ID	Bild	Design	Name	Breite	Anzahl Balken	Preis	Aktionen
1		Laufen	RUNNING	370	3	29.3 €	<button>Entfernen</button>
2		Radfahren	CYCLING	360	4	31.7 €	<button>Entfernen</button>
3		Schwimmen	SWIMMING	1000	5	40.9 €	<button>Entfernen</button>

Lieferadresse

Anrede  
Vorname  
Nachname  
E-Mail  
Straße

Abbildung 8: Tabellarische Auflistung der konfigurierten Produkte im Warenkorb

Ein Klick auf eine dieser Vorschauen führt zu einer vergrößerten Ansicht der gerenderten DXF-Datei. Diese Vorschau ist visuell identisch mit dem, was später in der Produktion verwendet wird. Damit erhält der Nutzer ein realistisches Bild seines Produktes.

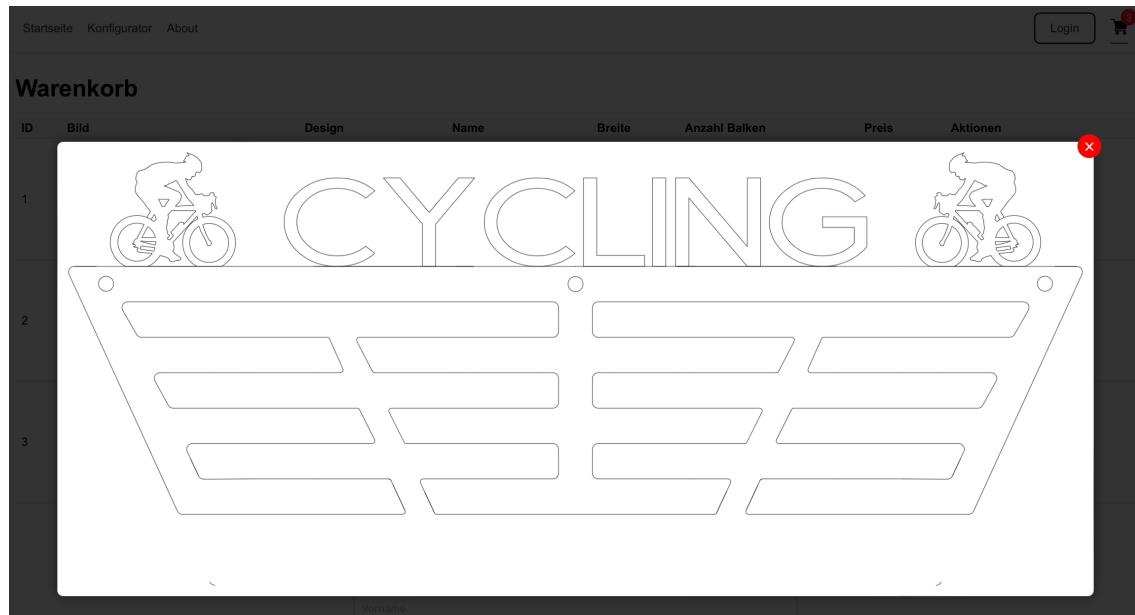


Abbildung 9: Vergrößerte Vorschau in der Frontend der serverseitig generierten DXF-Datei als PNG-Datei

Nachfolgend zwei Beispiele solcher Vorschauen, jeweils aus dem Backend gerendert. Abbildung 10 zeigt das Produkt „RUNNING“ im Design „Laufen“ mit drei Balken. Abbildung 11 entspricht exakt der Konfiguration aus Abbildung 7.

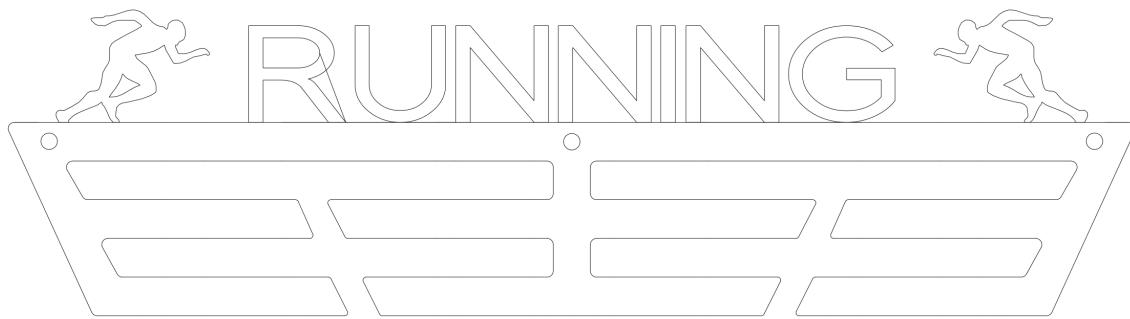


Abbildung 10: Generierte Vorschau für „Laufen“ mit drei Balken

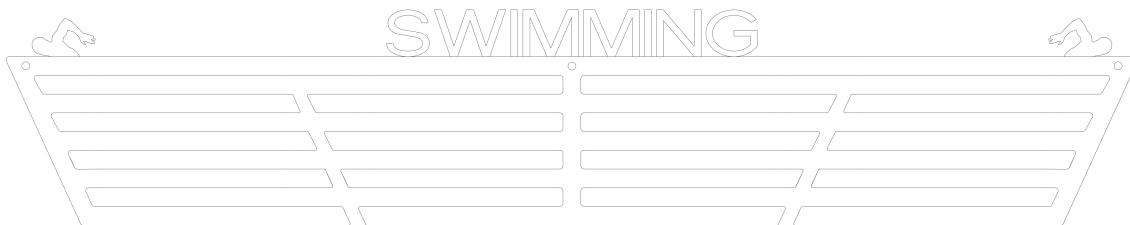


Abbildung 11: Generierte Vorschau für Design „Schwimmen“ mit fünf Balken

Bevor ein Kauf abgeschlossen werden kann, muss der Nutzer ein Kontaktformular ausfüllen. Dieses ist vollständig validiert – erst bei vollständiger Eingabe aller Felder wird der Kauf-Button aktiviert. Optional lässt sich die Rechnungsadresse von der Lieferadresse übernehmen, um Doppeleingaben zu vermeiden.

The screenshot shows a contact form with validation and optional billing address. At the top right are "Login" and a shopping cart icon with a red notification bubble containing the number "3". The form has two main sections: "Lieferadresse" and "Rechnungsadresse".

Lieferadresse
Herr
Luca
Pandza
luca.pandza@email.com
Musterstrasse
10
58135
Hagen

Rechnungsadresse	
Rechnungsadresse aus Lieferadresse übernehmen	
Frau	
Sabine	
Schmidt	
sabine.schmidt@email.de	
Andererstrasse	
20	
58640	

Abbildung 12: Kontaktformular mit Validierung und optionaler Rechnungsadresse

Nach dem Absenden erscheint eine Bestätigungsmeldung, und der Warenkorb wird automatisch geleert. Das erleichtert das erneute Konfigurieren weiterer Produkte und signalisiert dem Nutzer klar den Abschluss des Vorgangs.

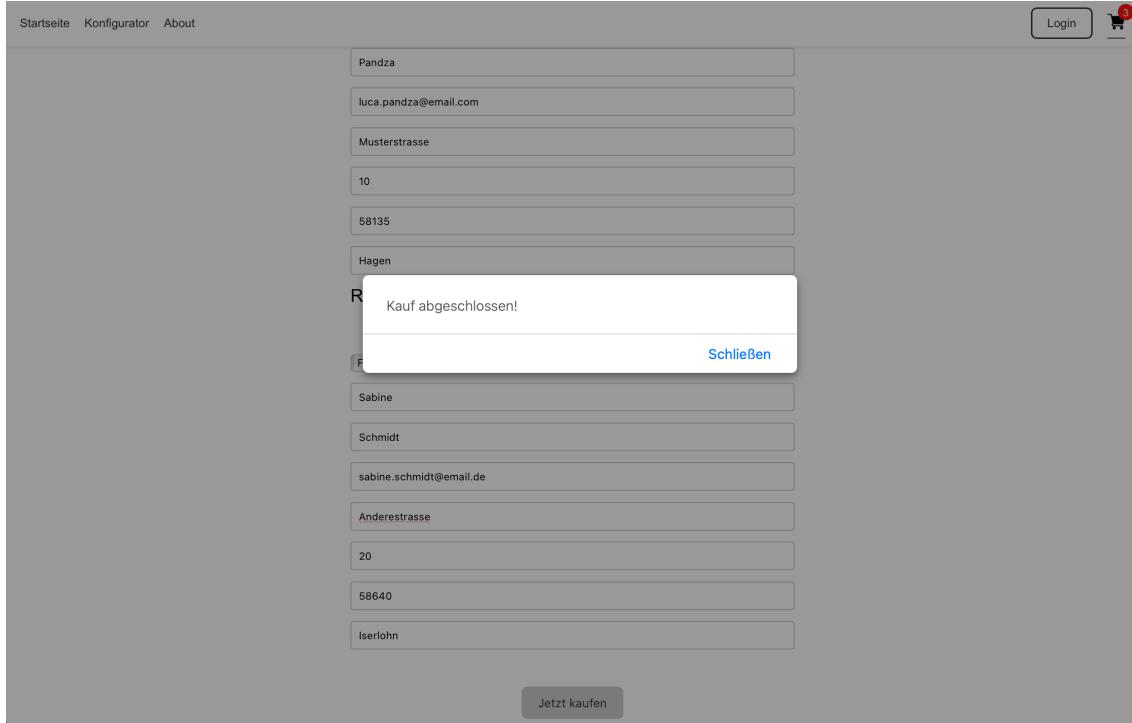


Abbildung 13: Kaufbestätigung und automatisches Leeren des Warenkorbs

Die eingegebenen Informationen werden dauerhaft in einer SQLite-Datenbank gespeichert. In Abbildung 14 sind die gespeicherten Kundeninformationen zu sehen – jede Zeile entspricht einem abgeschlossenen Bestellvorgang. Diese Daten umfassen Name, Adresse, E-Mail und ggf. abweichende Rechnungsadresse.

DB Browser for SQLite - /Users/lucapandza/dev/MedaillenhalterKonfigurator/backend/MedaillenDatenbank.db															
Database Structure Browse Data Edit Pragmas Execute SQL															
Table: customers															
id	reated_a	esse_i	sse_f	sse_k	lieferadresse_email	adresse_street	z_hc	sse_p	dress	rechnungsadresse_gender	dress	adresse	rechnungsadresse_email	rechnungsadresse_street	rechnungsadresse
1	2026-03-15	Herr	Luca	Panda	luca.pandza@email.com	Musterstrasse	10	68136	Hagen	Frau	Sabine	Schmidt	sabine.schmidt@email.de	Andererstrasse	20

Abbildung 14: Kundendatenbank mit gespeicherten Kontaktinformationen

In der Produkttabelle (Abbildung 15) sind alle gespeicherten Konfigurationen abgelegt – jeweils mit Bezug auf den zugehörigen Kunden über eine `customer_id`. Dort sind Breite, Balkenzahl, Textinhalt und gewähltes Design enthalten. Für die spätere Fertigung können die zugehörigen DXF-Dateien anhand des Dateinamens eindeutig zugeordnet werden.

DB Browser for SQLite - /Users/lucapandza/dev/MedaillenhalterKonfigurator/backend/MedaillenDatenbank.db						
Database Structure Browse Data Edit Pragmas Execute SQL						
Table: products						
id	reated_a	design	name	width	barCount	price
1	2026-03-15	Laufen	RUNNING	370	3	29.3
2	2026-03-15	Radfahren	CYCLING	360	4	81.7
3	2026-03-15	Schwimmen	SWIMMING	1000	6	40.9
						bestellt
						1

Abbildung 15: Produktdatenbank mit Konfiguration und Referenz auf Kunden

#### 4.2.3 WebAuthn-Registrierung und Login

Zusätzlich zur Kernfunktionalität des Medaillenhalter-Konfigurators wurde im Projekt ein modernes Login-System über **WebAuthn** integriert. Dabei handelt es sich um eine passwortlose Authentifizierungsmethode, bei der sogenannte Passkeys erstellt und lokal auf einem Gerät gespeichert werden.

Die Registrierung erfolgt über die Eingabe der E-Mail-Adresse. Im Anschluss wird ein Passkey erzeugt, wie in Abbildung 16 zu sehen. Der Prozess wurde mithilfe der JavaScript-API

`navigator.credentials.create()` angestoßen und funktionierte im Test reibungslos. Die entsprechenden Schlüsselpaare werden automatisch auf dem Gerät gespeichert.

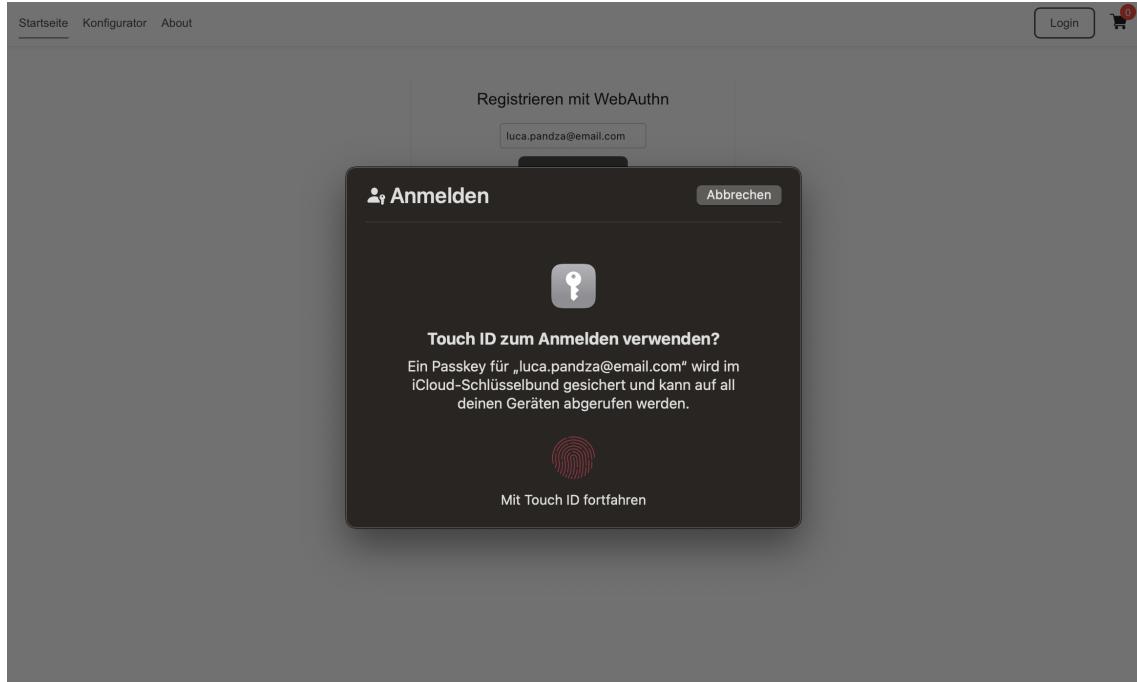


Abbildung 16: Registrierungsdialog zur Erstellung eines Passkeys

Nach erfolgreicher Registrierung erscheint eine Bestätigungsmeldung, die den Nutzer auffordert, sich anschließend einzuloggen (siehe Abbildung 17).

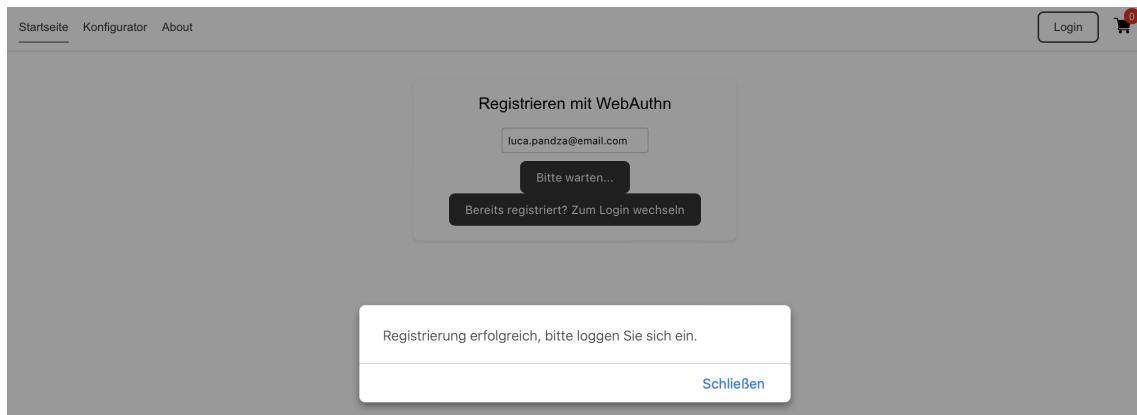


Abbildung 17: Registrierung erfolgreich abgeschlossen

Obwohl die Registrierung erfolgreich abgeschlossen wurde und die erzeugten Credentials korrekt in der Datenbank gespeichert wurden (vgl. Abbildung 18), traten beim Login über verschiedene Geräte Probleme auf.

id	credential_id	public_key
1	mxR6SKvD7cf...	pQECAyYgASFY...

Abbildung 18: WebAuthn-Datenbankeintrag nach erfolgreicher Registrierung

Beim Login-Versuch mit einem iPhone nach dem Scannen des QR-Codes kam es zu einer Aufforderung, ein anderes Gerät zu verwenden, um den Passkey für `localhost` anzulegen – siehe Abbildung 19. Dieses Verhalten war aus technischer Sicht unerwartet.



Abbildung 19: Aufforderung zur Einrichtung eines Passkeys auf einem anderen Gerät

Auch bei direkter Anmeldung auf dem selben Gerät kam es teilweise zu Fehlermeldungen, wie in Abbildung 20 dargestellt.

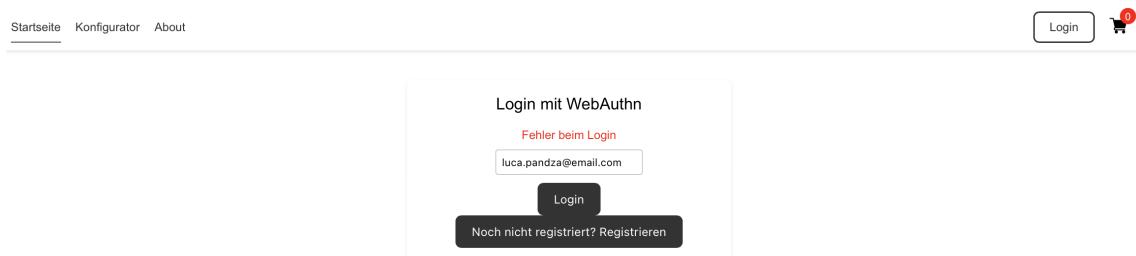


Abbildung 20: Fehler beim Login trotz vorheriger Registrierung

Da sich die Probleme auf das Login beschränkten und die restliche Funktionalität der Webanwendung im Fokus stand, wurde aus zeitlichen Gründen keine tiefere Fehleranalyse oder Behebung vorgenommen. Mögliche Ursachen könnten in der lokalen Entwicklungsumgebung (`localhost`) oder in Einschränkungen der verwendeten Plattformen liegen.

Schließlich wurde aus zeitlichen Gründen auf eine vollständige Integration und Fehlerbehebung bewusst verzichtet, weil der Login kein zentraler Bestandteil des Projekts war. Die Grundstruktur für WebAuthn wurde jedoch erfolgreich implementiert und in der Datenbank hinterlegt.

## 5 Implementierung

### 5.1 Projektbeginn

Die technische Umsetzung des Medaillenhalter-Konfigurators erfolgte in mehreren aufeinander abgestimmten Schritten. Dabei wurde das Projektverzeichnis zunächst manuell vorbereitet und anschließend um das React-Frontend sowie das Python-Backend ergänzt. Die Implementierung orientierte sich an einem iterativen Vorgehen mit wiederholten Tests und Verbesserungen während der Entwicklung.

Zunächst wurde ein neues Projektverzeichnis `Medaillenhalter` angelegt und mit Git versiert. Für das Frontend wurde mithilfe von `npx create-react-app` eine React-Anwendung im Unterordner `frontend/` generiert. Dieser Befehl legt die grundlegende Struktur für ein modernes React-Projekt an, inklusive Build-Skripten, Konfigurationen und einem lokalen Entwicklungsserver.

```
mkdir Medaillenhalter
cd Medaillenhalter
git init
npx create-react-app frontend
```

Parallel dazu wurde das Verzeichnis `backend/` erstellt und darin eine virtuelle Python-Umgebung eingerichtet. Anschließend erfolgte die Installation der benötigten Python-Bibliotheken wie `fastapi`, `uvicorn`, `ezdxf` und `pillow`. Die verwendeten Pakete wurden in einer `requirements.txt`-Datei dokumentiert.

```
mkdir backend
cd backend
python3 -m venv venv
source venv/bin/activate
pip install fastapi uvicorn ezdxf pillow ...
pip freeze > requirements.txt
```

Daraufhin wurde im Frontend die Komponentenhierarchie manuell aufgebaut. React-Komponenten wie `Configurator.jsx`, `Navbar.jsx` oder `Warenkorb.jsx` wurden im Verzeichnis `src/components/` erstellt. Für ein einheitliches Styling wurden CSS-Dateien wie `Button.css` ergänzt. Die Navigation innerhalb der Anwendung wurde mit Hilfe von `react-router-dom` umgesetzt:

```
npm install react-router-dom
npm start
```

Im Backend erfolgte in einem zweiten Schritt die Implementierung der zentralen API in der Datei `main.py`. Diese verarbeitet Anfragen des Frontends, z. B. zur Erstellung von Medaillenhalter-Vorschauen oder zum Abspeichern von Bestellungen. Unterstützt wird dies durch separate Skripte wie `generate_medaillenhalter_dxf_png_svg.py`, welche mithilfe von `ezdxf` die eigentlichen Dateien generieren.

Die Anwendung wurde iterativ erweitert: neue Komponenten im Frontend wurden direkt eingebunden, mit der API verbunden und getestet. Ebenso wurden neue Funktionen im Backend ergänzt, Anforderungen aus dem Frontend angepasst und die Datenverarbeitung schrittweise optimiert. Die Synchronisation zwischen Frontend und Backend erfolgte stets über REST-API-Aufrufe.

Zum Abschluss wurde das Projekt versioniert und über Git verwaltet. Änderungen wurden regelmäßig über `git add`, `git commit` und `git status` dokumentiert. Auf diese Weise konnte der Stand der Entwicklung transparent und nachvollziehbar gesichert werden.

```
git add .
git commit -m "Frontend: Warenkorb hinzugefügt"
```

Wie das Projekt lokal ausgeführt werden kann, ist bereits in Abschnitt 4.1 beschrieben.

### 5.2 Sequenzdiagramm

In nachfolgender Abbildung 21 ist das Zusammenspiel der wichtigsten Komponenten von Frontend und Backend bei einer Konfiguration bis hin zur Bestellung dargestellt.

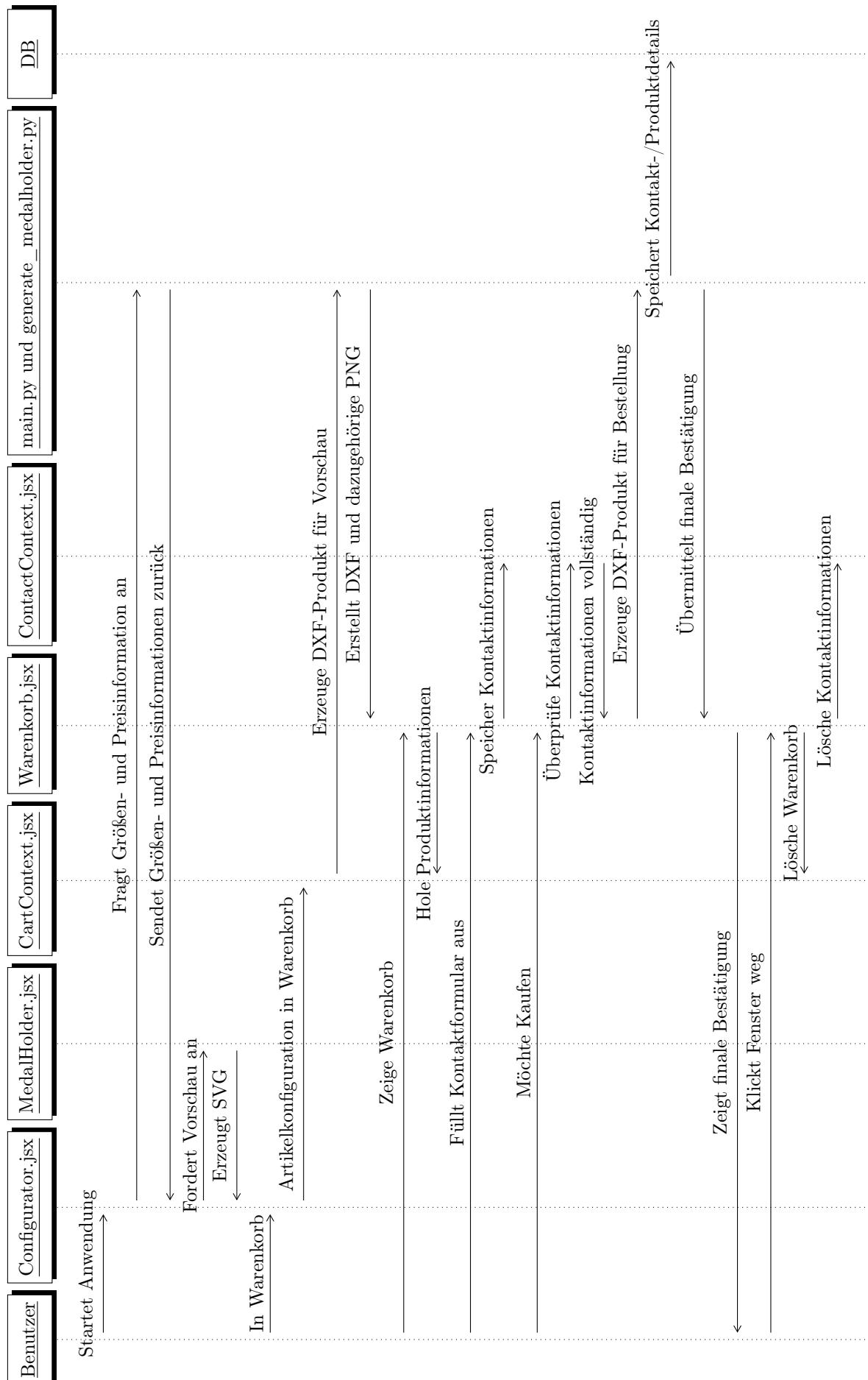


Abbildung 21: Einfaches Sequenzdiagramm des Konfigurations- und Bestellprozesses.

## 5.3 Frontend

Wie aus dem Sequenzdiagramm hervorgeht, sind die zentrale Bausteine für die Konfiguration des Frontends `Configurator.jsx` und die Vorschaukomponente `MedalHolder.jsx`. Die globale Zustandsverwaltung über `CartContext.jsx` und `ContactContext.jsx` sorgen für eine reaktive und konsistente Nutzererfahrung. Darauf hinaus ermöglichen Komponenten wie `Navbar.jsx` und `Warenkorb.jsx` eine nahtlose Navigation und einfache Kaufabwicklung. Im Folgenden werden diese und weitere Komponenten des Frontends im Detail vorgestellt.

### 5.3.1 Home.jsx – Startseite mit Call-to-Action

Die Komponente `Home.jsx` stellt die Startseite der Anwendung dar und dient als erster Kontaktpunkt für Besucherinnen und Besucher des Konfigurators. Sie verfolgt ein einfaches, aber effektives Ziel: Aufmerksamkeit erzeugen und zur Nutzung des Konfigurators motivieren. Dafür werden ein großflächiges Hintergrundbild und ein prägnanter Werbeslogan verwendet, ergänzt durch einen zentral positionierten Button mit direkter Weiterleitung zum Konfigurator.

Das Hintergrundbild wird per CSS-Klasse `bg-image` eingebunden und über ein Inline-Style mit verminderter Helligkeit versehen (`-brightness: 0.7`), um die weiße Schrift und den Button besser zur Geltung zu bringen.

Technisch gesehen ist `Home.jsx` eine rein präsentative Komponente ohne externe Abhängigkeiten oder Zustandsverknüpfungen. Es gibt weder API-Calls noch Props von außen. Der Call-to-Action-Button ist als `<Link>` zur Route `/konfigurator` umgesetzt und nutzt damit das clientseitige Routing von `react-router-dom`, ohne die Seite neu zu laden.

Durch diese klare Fokussierung auf Benutzerführung und visuelles Storytelling erfüllt `Home.jsx` seine Rolle als Einstiegspunkt in die Anwendung konsequent. Die Gestaltung lässt sich bei Bedarf leicht erweitern oder durch zusätzliche Inhalte wie Produktbeispiele ergänzen.

### 5.3.2 Navbar.jsx – Navigationsleiste mit dynamischem Warenkorb-Icon

Die Komponente `Navbar.jsx` stellt die zentrale Navigationsleiste der Anwendung dar. Sie ermöglicht es, schnell zwischen den Hauptseiten der Single-Page-Application (Startseite, Konfigurator, About, Warenkorb, Login/Profil) zu wechseln.

Technisch basiert die Navigation auf der Bibliothek `react-router-dom`, welche das clientseitige Routing übernimmt. Jeder `<NavLink>` wird mit einem dynamischen Klassenwechsel versehen, sodass aktive Links optisch hervorgehoben werden können. Zusätzlich wurde eine animierte Linie integriert, die unter dem aktuell aktiven Link eingeblendet wird. Dieser sogenannte „Sliding Indicator“ wird über den Hook `useEffect` beim Laden und beim Fenster-Resize aktualisiert und basiert auf `offsetLeft` sowie `offsetWidth` des aktiven Links, da diese Links unterschiedlich groß sind.

Ein weiteres UI-Element der Navigationsleiste ist das Warenkorb-Symbol mit rotem Zähler (siehe Abbildung 22). Dieses Icon wird über die Datei `cart-icon.jpg` aus dem Projektverzeichnis `img/` eingebunden. Das rote Badge mit der Produktanzahl wird dynamisch über den React Context `CartContext.jsx` gespeist. Dabei wird direkt auf die Länge der global gespeicherten Warenkorb-Einträge zugegriffen: `cartItems.length`.

Die konkrete Gestaltung dieses Icons basiert auf einem Beispiel aus einem YouTube-Tutorial<sup>1</sup>, das zur visuellen Inspiration diente. Das Icon besteht aus einem einfachen Bild (`<img>`), über das mittels CSS ein absolut positionierter roter Kreis gelegt wird. Der Zähler selbst ist mittig darin ausgerichtet und passt sich automatisch der Zahl der Produkte im Warenkorb an.

Zusätzlich bietet die rechte Seite der Navigation kontextabhängige Login-Optionen: Falls der Nutzer bereits eingeloggt ist, wird sein Vor- und Nachname angezeigt. Beim Hovern ändert sich der Button-Text zu „Ausloggen“. Wird ausgeloggt, wird der Kontextzustand `setLoggedInUser(null)` aufgerufen und die Seite auf die Startansicht umgeleitet.

Das zugehörige CSS wurde in der Datei `Navbar.css` abgelegt. Es trennt visuell die linke und rechte Seite der Navigation, regelt die Animation des Indikators und gestaltet das Zähler-Overlay. Damit fügt sich die `Navbar.jsx` nahtlos in das UI-Konzept der Anwendung ein – sowohl funktional als auch visuell.



Abbildung 22:  
Warenkorb-  
Icon mit rotem  
Zähler für in  
den Waren-  
korb abgelegte  
Produkte

<sup>1</sup><https://www.youtube.com/watch?v=jbfuzcrfjqQ>

### 5.3.3 CartContext.jsx – Globale Verwaltung des Warenkorbs

Die Datei `CartContext.jsx` ist für die zentrale Verwaltung des Warenkorbs zuständig und basiert auf der Context API von React. Sie stellt einen globalen Zustand bereit, der von allen Komponenten genutzt werden kann, ohne dass dieser Zustand manuell über Props durchgereicht werden muss. Dies erhöht die Wartbarkeit und reduziert die Komplexität insbesondere bei wachsendem Projektumfang.

Im Projekt wird der Context über den `CartProvider` bereitgestellt. Dieser kapselt die Zustandslogik in einem eigenständigen Modul und stellt vier zentrale Funktionen zur Verfügung:

- `addToCart(item)` – fügt ein neues Produkt mit eindeutiger ID hinzu.
- `removeFromCart(id)` – entfernt gezielt ein Produkt anhand seiner ID.
- `clearCart()` – leert den gesamten Warenkorb.
- `cartItems` – enthält die aktuelle Liste aller Konfigurationen.

Der Ordner `contexts/` wurde bewusst gewählt, um Zustandsverwaltung und UI-Komponenten voneinander zu trennen. Dies ermöglicht eine modulare und leicht erweiterbare Struktur. Der Warenkorb kann dadurch an beliebiger Stelle der Anwendung eingebunden werden, etwa in der `Navbar.jsx` zur Anzeige der Artikelanzahl oder im `Warenkorb.jsx` zur tabellarischen Darstellung aller Konfigurationen.

Ein großer Vorteil dieser Struktur liegt in der Entkoppelung der Komponentenlogik: Änderungen im Warenkorb (etwa das Hinzufügen oder Löschen von Items) müssen nicht an mehreren Stellen manuell synchronisiert werden. Stattdessen greift jede Komponente auf den selben globalen Zustand zu, wodurch eine konsistente Darstellung garantiert ist.

Da React Context Teil des Frameworks ist, wird keine zusätzliche Bibliothek benötigt. Dennoch erhält man durch diese Technik eine mächtige Lösung zur Zustandsverwaltung – ohne den Overhead komplexerer Alternativen wie Redux. Insgesamt stellt `CartContext.jsx` eine essentielle Komponente dar, um den Bestellprozess technisch robust und nutzerfreundlich umzusetzen.

### 5.3.4 Configurator.jsx – Zentrale Konfigurationslogik

Die Komponente `Configurator.jsx` bildet das funktionale Herzstück der Anwendung. Sie erlaubt es Nutzerinnen und Nutzern, ihren Medaillenhalter individuell zu gestalten – inklusive Auswahl von Design, Wunschtext, Breite und Anzahl der Balken.

Die Interaktivität basiert auf mehreren React States, die per `useState`-Hook verwaltet werden: `design`, `name`, `width`, `barCount`, `price` sowie `minWidth` und `maxReached`, um die technischen Begrenzungen der Halterbreite zu berücksichtigen.

Sobald der Nutzer einen Eingabewert verändert, wird mithilfe eines `fetch`-Requests an das Backend eine Neuberechnung der Mindestbreite und des Preises angestoßen. Dadurch bleibt die Vorschau stets konsistent mit den tatsächlichen Produktionsdaten. Die Abfrage erfolgt über die API `/medaillenhalter_info`, welche auf Basis des eingegebenen Textes, gewählten Designs und der Balkenanzahl eine Mindestbreite und den Preis zurückgibt.

Optisch wird die Komponente durch eine Hintergrundgrafik sowie durch das eingebettete SVG-Vorschaumodul `MedalHolder.jsx` ergänzt. Der Nutzer erhält so in Echtzeit ein visuelles Feedback über das spätere Produkt – eine intuitive und zugleich technisch valide Umsetzung der Konfigurationslogik. Diese Vorschau wird im Browser erzeugt und entlastet somit den Server im Backend.

Ein weiteres zentrales Feature ist der Button „In den Warenkorb“. Hierbei wird zusätzlich ein API-Call an `/generate_medaillenhalter` ausgelöst, um die DXF-, SVG- und PNG-Dateien serverseitig zu erzeugen. Erst danach wird die Konfiguration über den `CartContext` zur globalen Warenkorbstruktur hinzugefügt.

Durch die Kombination aus Live-Rückmeldung, serverseitiger Validierung und persistenter Speicherung der Konfiguration im Warenkorb erfüllt `Configurator.jsx` eine Schlüsselfunktion innerhalb der Webanwendung.

### 5.3.5 MedalHolder.jsx – SVG-Vorschau des Medaillenhalters

Die Komponente `MedalHolder.jsx` generiert eine dynamische SVG-Vorschau des konfigurierten Medaillenhalters direkt im Browser. Ziel ist es, Nutzerinnen und Nutzern ein realistisches visuelles Feedback ihres Produkts zu geben - allerdings ist stellt zur einfacheren Implementierung nicht das exakte Produkt dar. Eine exakte Darstellung erfolgt erst nachdem das Produkt in den Warenkorb hinzugefügt wurde und die DXF-Dateien zusammengebaut werden.

Dabei wird ein parametrisiertes SVG erzeugt, das sowohl die Formgebung des Halters (inkl. Balken, Schrägelemente, Mittelträger) als auch den konfigurierten Text und das Designsymbol (z. B. Läufer, Schwimmer, Radfahrer) abbildet. Die Vorschau basiert auf mathematischen Transformationen, darunter trigonometrische Berechnungen für die Schrägelemente der Halterform. Die horizontalen Balken werden ausgehend von einem festen Winkel über eine Rotation berechnet, sodass sie realitätsnah im SVG dargestellt werden können.

Besonders hervorzuheben ist die Spiegelung der linken Halterhälfte zur rechten Seite. Dies wird über ein `<g>` mit `transform="translate + scale(-1,1)"` umgesetzt – ein Trick zur symmetrischen Gestaltung (welcher auch ähnlich im Backend für die DXF-Generierung angewendet wird).

Die Darstellung des Wunschtextes erfolgt mit der Komponente `<text>`, die mittig oberhalb der Balken platziert ist. Ergänzt wird das Design durch links und rechts eingebettete Bildsymbole, die in Abhängigkeit vom gewählten Design angezeigt werden.

Technisch gesehen ist `MedalHolder.jsx` vollständig gekapselt: Sie erhält über Props die Parameter `name`, `width`, `barCount` und `design`, rendert daraus das entsprechende SVG und erfordert keine interne Zustandsverwaltung. Dadurch ist die Komponente hochgradig wiederverwendbar und flexibel.

### 5.3.6 Warenkorb.jsx – Produktübersicht und Bestellabwicklung

Die Komponente `Warenkorb.jsx` übernimmt die Rolle des Bestellzentrums innerhalb der Anwendung. Sie zeigt alle bisher vom Nutzer konfigurierten Produkte tabellarisch an, inklusive Design, Abmessungen, Preis und Vorschau. Die Vorschau des exakten Produkts wird dabei über ein serverseitig generiertes Bild aus der jeweiligen DXF-Datei dargestellt, das als Dateiname durch Parameter wie Text, Breite und Design eindeutig auf das entsprechende Produkt verweist. Durch einen Klick auf das Vorschaubild öffnet sich ein Zoom-Overlay.

Neben der Anzeige ermöglicht die Komponente auch das Entfernen einzelner Einträge sowie das vollständige Leeren des Warenkorbs nach erfolgreichem Kauf. Die dazugehörige Zustandsverwaltung erfolgt über den `CartContext`, auf den über `useContext` zugegriffen wird. Für jede Interaktion – sei es das Entfernen eines Produkts oder das Bestätigen eines Kaufs – stehen zentrale Methoden bereit: `removeFromCart()`, `clearCart()` und `addToCart()`.

Zentraler Bestandteil dieser Komponente ist der Button `Jetzt kaufen`, der die gesamte Konfiguration zusammen mit der Liefer- und Rechnungsadresse an das Backend sendet. Die Datenstruktur umfasst sowohl die Produktinformationen (`items`) als auch beide Adressen. Vor dem Absenden prüft die Anwendung, ob alle Pflichtfelder ausgefüllt sind. Dies geschieht über die Funktion `isDeliveryAddressValid()` aus dem `ContactContext`. Sollte diese Prüfung fehlschlagen, erscheint eine Fehlermeldung im Frontend.

Die Verknüpfung mit dem Kontaktformular ist eng integriert: Dieses wird direkt unterhalb der Produktübersicht eingebunden und in die Validierung einbezogen. Auf diese Weise entsteht ein nahtloser Übergang von der Konfiguration zur Kaufabwicklung. Nach dem erfolgreichen Kauf wird der Zustand zurückgesetzt – sowohl der Warenkorb als auch die Adressdaten werden geleert. Damit ist `Warenkorb.jsx` ein zentrales Bindeglied zwischen Benutzerinteraktion, Kontextzustand und Serverlogik.

### 5.3.7 Kontaktformular.jsx – Validierung und Kontaktdaten für die Bestellung

Die Komponente `Kontaktformular.jsx` bietet ein mehrteiliges Formular zur Eingabe der Liefer- und Rechnungsadresse. Sie arbeitet eng mit dem `ContactContext` zusammen, aus dem sie sowohl aktuelle Zustände als auch Änderungsfunktionen bezieht. Jeder Eingabewert – etwa Vorname, Nachname, E-Mail oder Stadt – ist dabei mit einem entsprechenden Feld im Context verknüpft. Änderungen im Formular werden über gezielte Setter-Funktionen wie `updateDeliveryAddress()` oder `updateInvoiceAddress()` direkt in den zentralen Zustand übernommen.

Ein wesentliches Merkmal ist die Validierung der Pflichtfelder. Diese wird automatisch ausgelöst, wenn ein Kaufversuch stattfindet. Fehlerhafte oder fehlende Eingaben werden visuell hervorgehoben, sodass der Nutzer eindeutig erkennt, welche Felder noch auszufüllen sind. Dies erhöht nicht nur die Benutzerfreundlichkeit, sondern verhindert auch unvollständige Kaufanfragen an das Backend.

Darüber hinaus bietet das Formular die Möglichkeit, die Rechnungsadresse aus der Lieferadresse zu übernehmen. Durch eine Checkbox kann der Nutzer festlegen, ob beide Adressen identisch sind. Bei Aktivierung werden die Werte automatisch synchronisiert – Änderungen an der Lieferadresse übertragen sich dann direkt auf die Rechnungsadresse. Optional kann die Rechnungsadresse auch

manuell bearbeitet werden, ohne dass die Synchronisierung verloren geht. Dies wird über ein lokal verwaltetes Flag gesteuert, das das Sichtbarmachen des zusätzlichen Formularabschnitts erlaubt.

Die gesamte Logik dieser Komponente ist stark modularisiert: Alle Felder sind wiederverwendbar aufgebaut, das Validierungssystem ist erweiterbar und der visuelle Zustand passt sich dynamisch an. `Kontaktformular.jsx` stellt damit eine robuste Grundlage für die Erfassung und Validierung von Kontaktdaten dar – ein elementarer Bestandteil des gesamten Kaufprozesses.

### 5.3.8 ContactContext.jsx – Globale Zustandsverwaltung für Nutzerdaten

Der `ContactContext` übernimmt die zentrale Verwaltung aller kontaktbezogenen Informationen – insbesondere der Liefer- und Rechnungsadresse. Diese Strukturierung in einem eigenen Kontextmodul folgt dem Prinzip der Separation of Concerns: Zustandslogik wird bewusst aus der UI ausgelagert, um die Wiederverwendbarkeit und Wartbarkeit des Codes zu erhöhen. Ein eigener Context bietet sich insbesondere bei wachsenden Anwendungen an, um sogenannte „Prop Drilling“-Probleme zu vermeiden **geeksforgeeks2025**.

Im Projekt wird der Context über den `ContactProvider` bereitgestellt. Dieser dient der zentralisierten Verwaltung von Benutzerdaten, die im Bestellprozess – insbesondere im Kontaktformular – erhoben werden. Durch die Verwendung des React Context API werden diese Zustände global verfügbar gemacht, ohne dass Props über mehrere Komponenten hinweg weitergereicht werden müssen.

Die Datei definiert dabei zwei Hauptobjekte: `deliveryAddress` und `invoiceAddress`, jeweils mit Feldern für Anrede, Vorname, Nachname, E-Mail, Straße, Hausnummer, PLZ und Stadt.

Neben reinen Getter/Setter-Funktionen zur Aktualisierung einzelner Felder über `updateDeliveryAddress()` bzw. `updateInvoiceAddress()` stellt der Context weitere zentrale Funktionen bereit:

- `isDeliveryAddressValid()` – überprüft, ob alle Pflichtfelder in der Lieferadresse ausgefüllt sind. Diese Methode wird beim Kaufabschluss aufgerufen, um fehlerhafte Eingaben zu verhindern.
- `toggleUseDeliveryForInvoice()` – aktiviert oder deaktiviert die automatische Übernahme der Lieferadresse als Rechnungsadresse. Wird diese Option aktiviert, so wird der Zustand von `invoiceAddress` synchron mit der Lieferadresse gehalten.
- `clearContactData()` – setzt alle gespeicherten Kontaktinformationen auf ihre Ausgangswerte zurück. Diese Methode wird nach erfolgreichem Abschluss eines Kaufs aufgerufen, um das Formular für eine neue Bestellung zu leeren.
- `useDeliveryForInvoice` – ein boolescher Schalter, der angibt, ob Rechnungs- und Lieferadresse identisch sind.
- `logged In User`, `setLogged In User()` – speichert zusätzlich optional einen eingeloggten Benutzer, falls Authentifizierung über WebAuthn genutzt wird.

Wie auch beim `CartProvider` sorgt dieser modulare Aufbau dafür, dass UI-Komponenten entkoppelt von der Logik arbeiten können. Dadurch wird die Wiederverwendbarkeit erhöht und die gesamte Anwendung bleibt besser testbar und wartbar. So kann dieser Context von jeder beliebigen Komponente im Projekt verwendet werden – sei es im `Kontaktformular`, im `Warenkorb` oder im `Profil`.

### Login und Profilansicht (nicht funktionsfähig)

Die Komponenten `Login.jsx` und `Profil.jsx` bilden das Frontend-Pendant zur serverseitigen WebAuthn-Authentifizierung. Auch wenn der Login über mehrere Geräte nicht zuverlässig funktionierte (siehe Abschnitt 4.2.3), wurden beide Komponenten für zukünftige Weiterentwicklungen implementiert und im Projektkontext eingebunden.

`Login.jsx` bietet sowohl Registrierung als auch Login über WebAuthn und greift dabei direkt auf die native `navigator.credentials`-API zu. Die E-Mail-Adresse dient als eindeutiger Identifier für die Passkey-Registrierung. Die Komponente unterscheidet zwischen Registrierungs- und Login-Modus und wechselt dynamisch per Button.

Die Login-Logik gliedert sich in folgende Schritte:

1. Anfordern der Login-/Registrierungsoptionen vom Backend

2. Ausführen der Authentifizierung über den Browser (z. B. TouchID, FaceID)
3. Rückgabe eines `Credential`-Objekts und dessen Weiterleitung an das Backend zur Validierung

Trotz erfolgreicher Registrierung und korrekt abgespeicherter Schlüssel in der Datenbank (vgl. Abb. 18) war der Login-Versuch nach Scannen des QR-Codes erfolglos.

`Profil.jsx` hingegen stellt eine Nachlade-Komponente dar, die nach erfolgreichem Login die letzten Bestellungen des eingeloggten Nutzers aus der Datenbank abruft. Die Daten werden über den API-Endpunkt `/profile/{id}` geladen und tabellarisch angezeigt. Als Basis dienen die Produktdaten (ID, Datum, Design, Breite, Balkenzahl, Preis, Status), die dem jeweiligen Nutzer zugeordnet sind. Für Styling und Tabellenaufbau wurde weitgehend auf die bereits existierenden Klassen aus `Warenkorb.css` zurückgegriffen.

Da die Authentifizierung aktuell nicht zuverlässig funktioniert, ist das Profil nur theoretisch nutzbar. Die zugrunde liegende Struktur ist jedoch bereits vorbereitet und könnte bei späterer Fehlerbehebung eingesetzt werden.

## 5.4 Backend

Das Backend bildet das technische Rückgrat der Anwendung und ist verantwortlich für die Kommunikation mit dem Frontend, die Dateigenerierung und Speicherung von Bestellungen. Es besteht aus mehreren Modulen, die unterschiedliche Aufgaben übernehmen: Die Datei `generate_buchstaben.py` wurde einmalig zur Erzeugung der Buchstaben-DXF-Dateien verwendet und liefert die Basis für die spätere Schriftplatzierung. Das Modul `generate_medaillenhalter_dxf_png_svg.py` übernimmt die eigentliche Geometrie- und Dateierstellung auf Basis der Benutzereingaben. Die zentrale API-Logik, inklusive Datenbankbindung und REST-Endpunkte, ist in der Datei `main.py` implementiert. Diese ruft gezielt Funktionen aus den zuvor genannten Modulen auf und stellt so eine modulare, aber eng verzahnte Architektur sicher. In den folgenden Abschnitten werden die Backend-Komponenten vorgestellt.

### 5.4.1 `generate_buchstaben.py` – Buchstabenerzeugung als DXF-Dateien

Die Datei `generate_buchstaben.py` dient zur automatisierten Generierung von Vektorpfaden für Buchstaben (A–Z) auf Basis eines TrueType-Fonts. Die resultierenden Buchstaben werden mithilfe der Python Bibliothek `ezdxf` in das DXF-Format überführt, das für die spätere Verwendung im Konfigurationsprozess von Medaillenhaltern notwendig ist.

Zunächst wird ein beliebiger TrueType-Font geladen (`sf-florencesans-sc-exp.ttf`), welche in Pfade umgewandelt werden. Die Klasse `DXFPen` übernimmt dabei das Übersetzen der Kurven der Schrift in eine von `ezdxf` verarbeitbare Struktur (`Path`). Hierbei treten noch Bugs bei Buchstaben mit "Inseln" (A, B, D, O, P und Q) auf.

Nach dem Zeichnen der Pfade erfolgt eine automatische Skalierung auf eine standardisierte Höhe (z. B. 50 mm), gefolgt von einer Transformation zum Ursprung (Drehung) für eine korrekte Ausrichtung. Dies stellt sicher, dass alle Buchstaben dieselbe Höhe und eine einheitliche Platzierung besitzen. Anschließend wird ein DXF-Dokument erzeugt, in dem sowohl Außen- als auch Innenkonturen jedes Buchstabens geschlossen und gespeichert werden.

Das Skript beinhaltet zusätzlich eine Breitenberechnung aller Buchstaben. Die ermittelten Werte werden in einer Datei `letter_widths.txt` abgelegt. Diese Datei wird später von anderen Skripten wie `get_medaillenhalter()` verwendet, um dynamisch die Breite eines Medaillenhalters aus einem beliebigen Text zu bestimmen.

Damit bildet `generate_buchstaben.py` die Grundlage für die individualisierte Gestaltung der Medaillenhalter-Schriftzüge. Durch die Vorab-Generierung und Speicherung der Buchstaben als DXF-Dateien, muss das Skript nur einmalig durchlaufen werden, solange sich die Schriftart nicht ändern sollte.

### 5.4.2 `generate_medaillenhalter_dxf_png_svg.py` - Erzeugung des Endprodukts

Dieses Modul ermöglicht die Erzeugung fertigungsggeeigneter Medaillenhalter mithilfe der Python Bibliothek `ezdxf`. Es übersetzt Benutzerkonfigurationen in DXF-, SVG- und PNG-Formate, die sowohl für Visualisierung als auch für Fertigungsprozesse geeignet sind. Dabei werden vorgefertigte DXF-Dateien eingelesen, welche modular miteinander kombiniert werden.

Ziele des Moduls:

- Einlesen und Kombinieren vordefinierter DXF-Bausteine (einzelne Buchstaben und 3-teilige Grundform des Medaillenhalters).
- Dynamische Positionierung und Spiegelung der Elemente (z. B. für Balken).
- Zeichnen des Benutzertexts aus Einzelbuchstaben-Dateien.
- Exportieren in drei Zielformate: DXF (CAD), PNG (Vorschau), SVG (Vektor).

Funktion `get_width_of_medaillenhalter(text, user_breite)`: Berechnet die Breite des Schriftzugs anhand einzelner Buchstabenbreiten, die zuvor aus DXF-Dateien extrahiert und in einer Textdatei gespeichert wurden. Dabei:

1. wird jedes Zeichen als Breite ausgelesen (Leerzeichen werden gesondert behandelt),
2. wird ein Mindestabstand zwischen Zeichen addiert,
3. wird eine Verlängerung berechnet, falls die Gesamtlänge den Basisbereich überschreitet.

Die einzelnen Schritte des Generierungsprozesses lassen sich anhand der Abbildungen 24 bis 26 nachvollziehen. Aus der gewählten Konfiguration wurde zunächst die Gesamtbreite von 600 mm und die dazugehörigen notwendigen Verlängerungen (Offsets der Grundformen) berechnet. Zunächst werden die drei Grundelemente (`3_1.dxf`, `3_2.dxf`, `3_3.dxf`) geladen und mit einem Offset positioniert (vgl. Abbildung 24). Diese Bauteile werden als Blöcke mit konkreten Offsets eingefügt, um den gewünschten Abstand und die Gesamtbreite korrekt abzubilden. Im Beispiel beträgt der Offset des zweiten Blocks -117,5 und der des dritten Blocks -305,0 (vgl. Abbildung 23).

Anschließend erfolgt eine Spiegelung zur linken Seite mithilfe der Funktion `insert_and_mirror()`, um eine symmetrische Gesamtform zu erzeugen (vgl. Abbildung 25). Die linke Seite wird gespiegelt und automatisch anhand der Offsets ausgerichtet. Im letzten Schritt werden die Verbindungsstreben zwischen den Blöcken, die beiden Designsymbole (z. B. Läufer links und rechts) sowie der zentrale Schriftzug eingefügt (vgl. Abbildung 26).

Sobald alle Elemente gesetzt wurden, exportiert das Skript drei finale Dateien, deren Pfade ebenfalls aus der Backend-Ausgabe hervorgehen: Eine `.dxf`-Datei zur Fertigung, eine `.png`-Datei zur visuellen Vorschau und eine `.svg`-Datei für skalierbare Vektordarstellung im Frontend. Der Dateiname enthält automatisch den Designnamen, die Anzahl der Balken sowie die Gesamtbreite (z. B. `LAUFEN_Laufen_3_600.dxf`).

```
Importiere './dxf_vorlagen/3_1.dxf' als Block '3_1' mit Offset (0, 0) ...
```

Abbildung 23: Konsolenausgabe beim Ausführen der Funktion `get_medaillenhalter(...)`: Detallierte Angaben zu den geladenen DXF-Vorlagedateien, Offsets, berechneter Gesamtbreite sowie Dateipfaden der erzeugten Ausgabedateien.

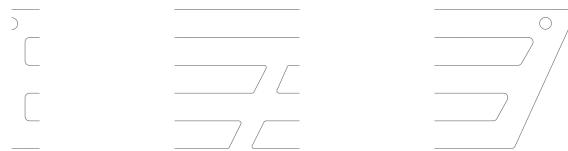


Abbildung 24: Einfügen der drei Grundformen des Medaillenhalters mit jeweils 3 Ebenen (`3_1.dxf`, `3_2.dxf`, `3_3.dxf`). Das Einfügen erfolgt mit Offsets in Abhängigkeit der festgelegten Breite durch Nutzer oder notwendiger Mindestbreite, damit der Text vollständig auf die Grundform passt. In diesem Fall handelt es bei 600 mm sich um die vom Nutzer angegebene Breite, da die Mindestbreite für den Text kleiner als 600 mm wäre. So hat der Nutzer die Form verlängert und nicht der Text. Die rechte Hälfte ist vollständig aufgebaut, die linke Seite fehlt noch und wird später gespiegelt.

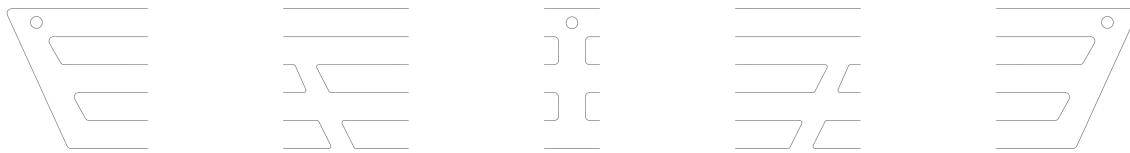


Abbildung 25: Nach Anwendung der Funktion `insert_and_mirror()`: Die linke Seite wurde gespiegelt hinzugefügt, wodurch eine vollständige symmetrische Grundform entsteht. Die Linien werden im nächsten Schritt in Abhängigkeit der Offsets der Grundelemente auf beiden Seiten gespiegelt hinzugefügt.

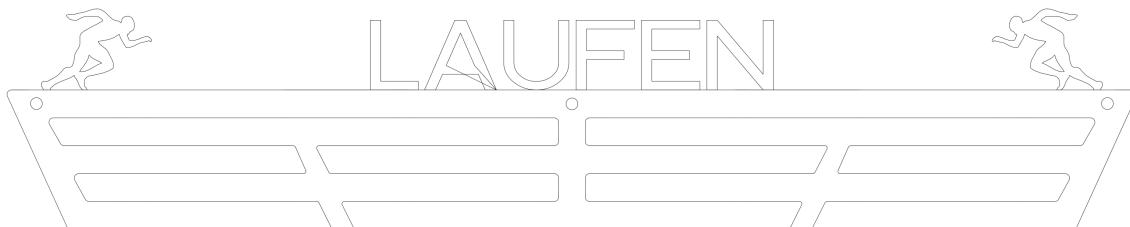


Abbildung 26: Endergebnis der Funktion `get_medaillenhalter(...)`: Der vollständige Medaillenhalter inklusive Text, Designsymbolen und dreifacher Balkenstruktur. Alle Komponenten wurden korrekt positioniert, mit Linien verbunden, gespiegelt und zusammengesetzt. Hierbei ist auch die „Inselproblematik“ beim A ersichtlich, welche in der Funktion in Abschnitt 5.4.1 auftritt.

#### 5.4.3 main.py - Serverlogik

Die Datei `main.py` bildet den zentralen Einstiegspunkt für das Backend der Anwendung. Sie basiert auf dem Framework **FastAPI**, das moderne, asynchrone REST-APIs ermöglicht und gleichzeitig eine automatische OpenAPI/Swagger-Dokumentation erzeugt. In dieser Datei sind sowohl die Datenbanklogik, als auch die gesamte API-Struktur und die Integration von WebAuthn enthalten.

#### Datenspeicherung mit SQLite

Die Anwendung verwendet eine SQLite-Datenbank (`MedaillenDatenbank.db`), die über SQLAlchemy asynchron eingebunden wird. Dieses Setup ist bewusst leichtgewichtig gehalten, da SQLite keine zusätzliche Serverinstallation erfordert und sich hervorragend für Entwicklungs- oder Demonstrationszwecke eignet.

In der Datenbank sind drei Tabellen definiert:

- `Customer` – speichert Liefer- und Rechnungsadressen sowie E-Mail-Adressen.
- `Product` – enthält Produktkonfigurationen (Design, Name, Breite, Balkenzahl, Preis).
- `WebAuthnCredential` – speichert Public Keys und Signaturen für die passwortlose Anmeldung.

Jeder Eintrag in `Product` ist über einen ForeignKey mit einem Kunden in `Customer` verknüpft. Diese Beziehungen ermöglichen eine saubere Historie von Bestellungen pro Nutzer.

#### FastAPI-Struktur und Endpunkte

Die API folgt einem modularen Aufbau und bietet unter anderem folgende Endpunkte:

- `/medaillenhalter_info` – berechnet Preis einer Konfiguration sowie dessen Breite mithilfe der Funktionsaufrufe `get_width_of_medaillenhalter(...)` aus Abschnitt 5.4.2.
- `/generate_medaillenhalter` – erzeugt alle zugehörigen Dateien (DXF, PNG, SVG) mithilfe des Funktionsaufrufs `get_medaillenhalter(...)` aus Abschnitt 5.4.2.
- `/get_image_url` – liefert das zugehörige Bild einer Konfiguration.
- `/purchase` – verarbeitet Kaufvorgänge, legt Kunden- und Produktdaten in der DB an.
- `/profile/{id}` – zeigt Bestellhistorie eines Nutzers.

Alle Routen sind CORS-fähig und akzeptieren Anfragen vom React-Frontend auf `localhost:3000`.

### **WebAuthn: Login-System mit Passkeys**

Das Login-System über WebAuthn konnte noch nicht erfolgreich implementiert werden (siehe Abschnitt 4.2.3). Es soll sowohl die Registrierung als auch den Login mittels Passkeys durchführen und nutzt hierbei das externe Repo von *duo-labs* ([https://github.com/duo-labs/py\\_webauthn.git](https://github.com/duo-labs/py_webauthn.git)).

Registrierung:

1. `/webauthn/register/options` – erzeugt Registrierungschallenge und Optionen.
2. `/webauthn/register/complete` – verifiziert den Response und speichert das Credential.

Login:

1. `/webauthn/login/options` – erzeugt Authentifizierungsoptionen.
2. `/webauthn/login/complete` – verifiziert Signatur und setzt SignCount hoch.

Ein zusätzlicher Endpunkt `/webauthn/is_registered` prüft, ob ein Nutzer bereits WebAuthn-registriert ist.

## **6 Ausblick**

Die Umsetzung des Medaillenhalter-Konfigurators in React und FastAPI zeigt, wie sich moderne Webtechnologien mit CAD-Prozessen kombinieren lassen, um individualisierte Produkte digital erfassbar und darstellbar zu machen. Dabei wurde eine klare Trennung zwischen Frontend und Backend realisiert, die sowohl die Wartbarkeit als auch die Erweiterbarkeit des Systems erleichtert. Trotzdem ergeben sich aus der aktuellen Umsetzung zahlreiche sinnvolle Erweiterungsmöglichkeiten.

Ein offensichtlicher nächster Schritt ist die Verbesserung der SVG-Vorschau. Derzeit stellt diese nur eine abstrahierte Version des tatsächlichen Produkts dar. Gleichzeitig sollte die SVG-Datei weiterhin bewusst reduziert bleiben, um eine direkte Kopierbarkeit zu vermeiden. Die Vorschau könnte noch informativer gestaltet werden – etwa durch Höhenlinien für die Bemaßung.

Auch die Auswahlmöglichkeiten könnten deutlich erweitert werden. Aktuell steht nur eine einzige Schriftart zur Verfügung, deren Umwandlung in DXF-Dateien teilweise fehleranfällig ist – besonders bei komplexeren Buchstaben mit „Inseln“ wie „A“, „B“ oder „D“. Hier könnten zusätzliche, klar konstruierte Schriften mit besserer Lesbarkeit eingebunden werden. Auch eine Umschaltfunktion zwischen mehreren Schriftarten innerhalb des Konfigurators wäre denkbar. Darüber hinaus wäre es sinnvoll, die Umwandlung von Schriftarten für alle Buchstaben in Groß- und Kleinschreibung sowie Sonderzeichen in DXF langfristig robuster zu gestalten.

Ein weiteres naheliegendes Feature ist die Materialauswahl. Derzeit sind alle Halter standardmäßig aus Metall gedacht. In Zukunft könnten Nutzerinnen und Nutzer zwischen verschiedenen Materialien wie Holz oder Edelstahl wählen, wobei sich diese Entscheidung auch optisch in der Vorschau widerspiegeln sollte. Dafür müsste die SVG-Visualisierung um Farbgebungen oder Texturen ergänzt werden, die jeweils typische Eigenschaften des gewählten Materials widerspiegeln – beispielsweise Holzmaserungen oder metallisch glänzende Flächen.

Auch auf Backend-Seite ergeben sich Erweiterungspotenziale. Momentan ist das System stark auf eine feste Auswahl an Designs und Konfigurationen beschränkt. Zukünftig wäre eine modularere Struktur denkbar, in der Designs als dynamisch ladbare Bausteine hinterlegt sind. So könnten neue Varianten schnell ergänzt und im Frontend automatisch eingebunden werden. Auch die Flexibilität beim Einfügen von Schriftzügen oder Symbolen ließe sich erhöhen – etwa durch eine automatische Zentrierung langerer Texte oder ein intelligentes Layout, das die Position der Designelemente anpasst.

Die Integration von WebAuthn als passwortlose Login-Methode ist noch nicht ganz gelungen, konnte in der Praxis noch nicht zuverlässig über verschiedene Geräte hinweg genutzt werden. Nach Behebung der bestehenden Fehler – insbesondere bei der Anmeldung über mobile Geräte – ließe sich die Funktionalität deutlich ausbauen. Denkbar wäre etwa eine persönliche Profilseite, auf der Bestellstatus („offen“, „in Produktion“, „versendet“) angezeigt und Kontaktinformationen aktualisiert werden können. Auch eine Übersicht vergangener Bestellungen, die Möglichkeit zur Nachbestellung oder zur Verwaltung gespeicherter Konfigurationen wäre sinnvoll. Die technische Basis hierfür ist bereits vorhanden, muss jedoch weiter stabilisiert und ausgebaut werden.

Schließlich wäre auch die Vorschaufunktion für generierte Produkte ausbaufähig. Die aktuell eingesetzten PNG-Dateien basieren auf einer einfachen Darstellung der DXF-Datei. In Zukunft könnten diese Vorschauen realitätsnäher gestaltet werden – beispielsweise durch Rendering mit

Schatten, perspektivischer Verzerrung oder durch ein 3D-Modell, das per WebGL interaktiv drehbar ist. Alternativ könnte auch die SVG-Vorschau um Funktionen wie Maßlinien oder optional einblendbare Bohrungen erweitert werden.

Insgesamt zeigt sich, dass auf Basis des aktuellen Prototyps viele sinnvolle Erweiterungen realisierbar sind – sowohl hinsichtlich der Benutzeroberfläche, der Produktdarstellung als auch der Funktionalität und Individualisierungsmöglichkeiten. Diese Optionen bilden eine fundierte Grundlage für die kontinuierliche Weiterentwicklung der Anwendung.

## 7 Fazit

Im Rahmen dieser Arbeit wurde ein funktionierender Webshop-Konfigurator für Medaillenhalter entwickelt, der auf modernen Technologien wie React im Frontend und FastAPI im Backend basiert. Der modulare Aufbau hat sich als sinnvoll erwiesen: Durch die konsequente Trennung von Client- und Serverlogik konnten beide Teile unabhängig voneinander aufgebaut und getestet werden. Besonders positiv fiel auf, dass React eine hohe Flexibilität in der Komponentenstruktur bietet, gleichzeitig aber durch die Context API auch eine robuste Zustandsverwaltung für komplexe Anwendungsfälle wie Warenkorb oder Formularvalidierung ermöglicht.

Die automatisierte Generierung von DXF-Dateien über `ezdxf` war ebenfalls ein zentraler Bestandteil des Projekts. Dabei wurden vorgefertigte Geometrien und Schriftzüge dynamisch zusammengesetzt, um eine individuelle Fertigungsdatei für jedes Produkt zu erzeugen. Die Kombination aus Textlayout, Designelementen und Dateiexport konnte erfolgreich umgesetzt werden – trotz einiger technischer Herausforderungen bei der Umwandlung von Schriftarten in geometrische Formen.

Neben der funktionalen Umsetzung ergaben sich auch viele praktische Erkenntnisse über den Aufbau und die Limitierungen moderner Webframeworks. Die Integration von WebAuthn etwa zeigte, wie wichtig standardisierte Schnittstellen und sauberes Fehlerhandling sind – insbesondere bei sicherheitskritischen Anwendungen wie der passwortlosen Anmeldung. Die bestehende Implementierung legt hier bereits eine gute Grundlage, ist aber noch nicht vollumfänglich einsatzbereit.

Zusammenfassend lässt sich sagen, dass durch das Projekt sowohl technisches Wissen in der Webentwicklung vertieft als auch ein vollständiger und funktionaler Prototyp erstellt wurde. Die Anwendung erlaubt bereits heute eine vollständige Konfiguration, Visualisierung und serverseitige Verarbeitung individueller Produktwünsche. Gleichzeitig bestehen zahlreiche konkrete Möglichkeiten zur Weiterentwicklung – angefangen bei einer erweiterten Vorschau über zusätzliche Materialien und Schriftarten bis hin zu einer verbesserten Nutzerkontoverwaltung mit Statusanzeige.

Diese Perspektiven machen das System nicht nur für den aktuellen Anwendungsfall relevant, sondern eröffnen auch Wege für eine potenzielle Weiterentwicklung in Richtung eines vollwertigen Produktkonfigurators mit professionellem Anspruch.

## Literatur

- [1] Autodesk Help. *Über DXF-Dateien*. Online verfügbar unter: <https://help.autodesk.com/view/ACDLT/2022/DEU/?guid=GUID-11AEFE3C-D05A-45C3-8604-7B6A594440F3>, aufgerufen am 12.03.2025.
- [2] Cart Icon. *Cart Icon on Pinterest*. Online verfügbar unter: <https://de.pinterest.com/pin/689613761726531899/>.
- [3] DataScientest. *FastAPI: Das Hochleistungs-Framework für APIs mit Python*. Online verfügbar unter: <https://datascientest.com/de/fastapi>, aufgerufen am 15.03.2025.
- [4] ezdxf Documentation. *What is ezdxf?* Online verfügbar unter: <https://ezdxf.readthedocs.io/en/stable/introduction.html>, aufgerufen am 12.03.2025.
- [5] FastAPI Documentation. *What is FastAPI?* Online verfügbar unter: <https://fastapi.tiangolo.com/>, aufgerufen am 15.03.2025.
- [6] Christian Gawron. *Moderne Webframeworks*. 1. Auflage. Hanser Fachbuch, 2020.
- [7] GeeksForGeeks. *What is Prop Drilling and How to Avoid it?*, aktualisiert am 07.02.2025. Online verfügbar unter: <https://www.geeksforgeeks.org/what-is-prop-drilling-and-how-to-avoid-it/>, aufgerufen am 11.03.2025.
- [8] Chidi. *Angular vs React*. Hygraph Blog, 08.05.2024. Online verfügbar unter: <https://hygraph.com/blog/angular-vs-react>, aufgerufen am 10.03.2025.
- [9] Zadhid Powell. *Angular vs. React: Ein detaillierter Vergleich*. Kinsta Blog, 27.07.2023. Online verfügbar unter: <https://kinsta.com/de/blog/angular-vs-react/>, aufgerufen am 10.03.2025.
- [10] 2025 Meta Platforms, Inc. *React Documentation. Thinking in React*. Online verfügbar unter: <https://react.dev/learn/thinking-in-react>, aufgerufen am 11.03.2025.
- [11] 2025 Meta Platforms, Inc. *ReactJS Legacy Docs. A JavaScript library for building user interfaces*. Online verfügbar unter: <https://legacy.reactjs.org/>, aufgerufen am 11.03.2025.
- [12] Vishal Siddhpara. *React Documentation. React vs Angular: Which JS Framework to choose for Front-end Development?* Radixweb Blog, aktualisiert am 13.02.2025. Online verfügbar unter: <https://radixweb.com/blog/react-vs-angularWeb>, aufgerufen am 10.03.2025.
- [13] Uvicorn Documentation. *ASGI server for Python*. Online verfügbar unter: <https://www.uvicorn.org/>, aufgerufen am 15.03.2025.