HTL Leonding
next level

HÖHERE TECHNISCHE
BUNDESLEHRANSTALT
LEONDING

Smartbill Converter - Rechnungszuordnung mittels KI

DIPLOMARBEIT

verfasst im Rahmen der

Reife- und Diplomprüfung

an der

Höheren Abteilung für Informatik

Eingereicht von:
Sebastian Radić
Luis Schörgendorfer

Betreuer:
Gerald Unterrainer

Projektpartner:
PROGRAMMIERFABRIK GmbH

Leonding, 4. April 2025

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die Arbeit wurde bisher in gleicher oder ähnlicher Weise keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Die vorliegende Diplomarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Leonding, 4. April 2025 Sebastian Radić & Luis Schörgendorfer

Abstract

Brief summary of our amazing work. In English. This is the only time we have to include a picture within the text. The picture should somehow represent your thesis. This is untypical for scientific work but required by the powers that are. Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Zusammenfassung

Zusammenfassung unserer genialen Arbeit. Auf Deutsch. Das ist das einzige Mal, dass eine Grafik in den Textfluss eingebunden wird. Die gewählte Grafik soll irgendwie eure Arbeit repräsentieren. Das ist ungewöhnlich für eine wissenschaftliche Arbeit aber eine Anforderung der Obrigkeit. *Bitte auf keinen Fall mit der Zusammenfassung verwechseln, die den Abschluss der Arbeit bildet!* Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Inhaltsverzeichnis

1. Einleitung

1.1. Ausgangssituation und Motivation

[Wird gemeinsam verfasst]

1.2. Problemstellung und Rahmenbedingungen

[Wird gemeinsam verfasst]

1.3. Zielsetzung und Erfolgskriterien

[Wird gemeinsam verfasst]

1.4. Umfang, Annahmen und Abgrenzungen

[Wird gemeinsam verfasst]

1.5. Beiträge (Team- und Individualleistungen)

[Wird gemeinsam verfasst]

1.6. Aufbau der Arbeit und Leseführung

[Wird gemeinsam verfasst]

Teil I.

Theoretische Grundlagen

2. E-Rechnungsstandards und Compliance

Die Transformation von papierbasierten Geschäftsprozessen hin zu vollautomatisierten digitalen Workflows ist einer der wesentlichen Treiber der modernen IT-Ökonomie. Im Zentrum dieser Entwicklung steht die elektronische Rechnung (E-Rechnung). Anders als eine bloße PDF-Datei, die lediglich ein digitales Abbild eines Papierdokuments darstellt, definiert eine echte E-Rechnung strukturierte Daten, die von Maschinen ohne menschliches Zutun verarbeitet, geprüft und verbucht werden können.

In diesem Kapitel werden die theoretischen Fundamente und spezifischen Standards analysiert, die für die Entwicklung des *SmartBillConverter* maßgeblich sind. Der Fokus liegt dabei auf dem österreichischen Standard *ebInterface* sowie dem deutsch-europäischen Hybridformat *ZUGFeRD*, eingebettet in den Kontext der europäischen Norm EN 16931.

2.1. ebInterface 6.1 (Österreich)

2.1.1. Zweck und Historie

ebInterface ist der nationale österreichische Standard für die elektronische Rechnungslegung. Er wurde von der AUSTRIAPRO, einer Initiative der Wirtschaftskammer Österreich (WKO), entwickelt, um den Datenaustausch zwischen Unternehmen sowie zwischen Unternehmen und der öffentlichen Verwaltung zu standardisieren. Die Entwicklung von ebInterface reicht bis in die frühen 2000er Jahre zurück und hat sich über mehrere Versionen (4.0, 5.0, 6.0) bis zur aktuellen Version 6.1 weiterentwickelt.

Der primäre Zweck von ebInterface liegt in der Schaffung einer semantischen Interoperabilität. Während EDIFACT-Standards in großen Industriekonzernen dominierten, fehlte für KMUs (Kleine und mittlere Unternehmen) lange Zeit ein einfach zu implementieren-

des, XML-basiertes Format. ebInterface füllt diese Lücke. Seit 2014 ist die E-Rechnung an den Bund in Österreich verpflichtend, was die Verbreitung von ebInterface massiv gefördert hat. Rechnungen an den Bund müssen über das Unternehmensserviceportal (USP) eingebracht werden, welches ebInterface als primäres Format akzeptiert.[1]

2.1.2. Kernelemente und Struktur

Technisch gesehen ist ebInterface eine reine XML-Struktur. Im Gegensatz zu ZUGFeRD gibt es keine eingebettete PDF-Datei; die XML-Datei *ist* die Rechnung. Dies bedeutet, dass für die visuelle Darstellung (z.B. für die manuelle Prüfung durch einen Sachbearbeiter) ein Stylesheet (XSLT) oder ein Viewer notwendig ist.

Die Struktur eines ebInterface 6.1 Dokuments ist hierarchisch aufgebaut und umfasst folgende Hauptbereiche:

- **Root-Element**: `<Invoice>` definiert die Grundeigenschaften wie Währung, Sprache und Dokumententitel.
- **Header-Daten**: Hierzu zählen die eindeutige Rechnungsnummer (`InvoiceNumber`), das Rechnungsdatum (`InvoiceDate`) und der Leistungszeitraum (`Delivery`).
- **Biller (Rechnungssteller)**: Enthält detaillierte Informationen zum Leistenden, inklusive der gesetzlich vorgeschriebenen UID-Nummer (VAT Identification Number), Anschrift, Kontaktdaten und Bankverbindung.
- **InvoiceRecipient (Rechnungsempfänger)**: Analog zum Biller die Daten des Leistungsempfängers. Hier ist oft die Auftragsreferenz (Order Reference) entscheidend für die automatische Zuordnung.
- **Details (Positionen)**: Der Kern der Rechnung. Hier werden in einer Liste (`ItemList`) die einzelnen Positionen (`ListLineItem`) aufgeführt. Jede Position enthält Menge, Einheit, Einzelpreis, Beschreibung, Steuerreferenz und Zeilensumme.
- **Tax (Steuern)**: Eine Zusammenfassung der Steuerbeträge, gruppiert nach Steuersätzen. Dies ist essenziell für die Vorsteuerabzugs-Prüfung.
- **PaymentConditions**: Zahlungsziele, Skonto-Informationen und Fälligkeitsdaten.

[1] Vgl. AUSTRIAPRO: *ebInterface - Der österreichische Standard für die elektronische Rechnung*, https://www.ebinterface.at/, letzter Zugriff am 19.12.2025

2.1.3. Pflichtfelder und Validierung

Die Validität einer ebInterface-Rechnung wird durch ein XML Schema (XSD) definiert. Pflichtfelder sind jene, die laut Umsatzsteuergesetz (UStG) für eine ordnungsgemäße Rechnung erforderlich sind. Dazu gehören in Österreich unter anderem:

- Name und Anschrift des liefernden und empfangenden Unternehmers.
- Menge und handelsübliche Bezeichnung der Gegenstände.
- Tag der Lieferung oder sonstigen Leistung.
- Entgelt (Netto) und der darauf entfallende Steuerbetrag.
- Der anzuwendende Steuersatz.
- Ausstellungsdatum und fortlaufende Nummer.
- UID-Nummer des Leistenden (und ab 10.000 Euro Brutto auch des Empfängers).

Eine Besonderheit von ebInterface 6.1 ist die strikte Typisierung. Datumsfelder müssen dem ISO 8601 Format entsprechen, Beträge sind als Dezimalzahlen mit definierter Genauigkeit anzugeben. Dies verhindert Interpretationsspielräume, die bei OCR-basierten Verfahren oft zu Fehlern führen.

2.1.4. Beispielhafte XML-Struktur

Um die Struktur zu verdeutlichen, zeigt das folgende Listing einen gekürzten Ausschnitt einer validen ebInterface 6.1 Rechnung. Man erkennt deutlich die hierarchische Gliederung und die sprechenden Tag-Namen, die eine Implementierung erleichtern.

Listing 1: Ausschnitt einer ebInterface 6.1 Rechnung

```
<Invoice xmlns="http://www.ebinterface.at/schema/6p1/"
         GeneratingSystem="SmartBillConverter">
  <InvoiceNumber>2024-001</InvoiceNumber>
  <InvoiceDate>2024-01-15</InvoiceDate>
  <Delivery>
    <Date>2024-01-10</Date>
  </Delivery>
  <Biller>
    <VATIdentificationNumber>ATU12345678</VATIdentificationNumber>
    <Address>
      <Name>Musterfirma GmbH</Name>
      <Street>Hauptstrasse 1</Street>
      <Town>Wien</Town>
      <ZIP>1010</ZIP>
      <Country>Austria</Country>
    </Address>
  </Biller>
  <Details>
    <ItemList>
      <ListLineItem>
        <Description>Software Entwicklung</Description>
        <Quantity Unit="h">10.00</Quantity>
        <UnitPrice>100.00</UnitPrice>
```

```
24          <TaxItem>
25            <TaxPercent>20</TaxPercent>
26          </TaxItem>
27          <LineItemAmount>1000.00</LineItemAmount>
28        </ListLineItem>
29      </ItemList>
30    </Details>
31    <Tax>
32      <VAT>
33        <TaxedAmount>1000.00</TaxedAmount>
34        <TaxPercent>20</TaxPercent>
35        <Amount>200.00</Amount>
36      </VAT>
37    </Tax>
38    <TotalGrossAmount>1200.00</TotalGrossAmount>
39  </Invoice>
```

Dieses Beispiel illustriert, wie essenziell die korrekte Verschachtelung ist. Ein häufiger Fehler bei der Generierung ist die Diskrepanz zwischen den berechneten Zeilensummen (`LineItemAmount`) und den Steuer-Aggregaten im `Tax`-Block.

2.2. ZUGFeRD 2.3 und EN 16931 (Deutschland/EU)

2.2.1. Das hybride Konzept

ZUGFeRD (Zentraler User Guide des Forums elektronische Rechnung Deutschland) verfolgt einen anderen Ansatz als ebInterface. Es ist ein hybrides Format, das die Vorteile der menschlichen Lesbarkeit mit der maschinellen Verarbeitbarkeit kombiniert. Eine ZUGFeRD-Rechnung ist technisch gesehen eine PDF/A-3 Datei.

PDF/A-3 ist eine Erweiterung des PDF-Standards, die es erlaubt, beliebige Dateien als Anhang in das PDF einzubetten. Bei ZUGFeRD wird eine XML-Datei (genannt `factur-x.xml` oder `zugferd-invoice.xml`) in das PDF eingebettet.

- **Sichtbare Ebene**: Das PDF zeigt das Rechnungsbild, wie man es von Papier kennt.
- **Unsichtbare Ebene**: Das eingebettete XML enthält die gleichen Daten in strukturierter Form.

Dieses Konzept löst das Akzeptanzproblem bei kleineren Empfängern: Wer keine Software zur automatischen Verarbeitung hat, behandelt die Datei wie ein normales PDF. Wer automatisieren will, extrahiert das XML.

2.2.2. Profile und Konformität

ZUGFeRD 2.3 implementiert die europäische Norm EN 16931. Da die Anforderungen je nach Unternehmensgröße und Branche variieren, definiert ZUGFeRD verschiedene Profile:

1. **MINIMUM**: Enthält nur rudimentäre Daten, um eine Buchungshilfe zu bieten. Es erfüllt nicht die Anforderungen an eine steuerrechtliche Rechnung.
2. **BASIC WL (Without Lines)**: Enthält Kopfdaten und Summen, aber keine Positionsdaten. Nützlich für einfache Verbuchung, aber eingeschränkt in der Prüfung.
3. **BASIC**: Erfüllt die Anforderungen des deutschen UStG für Rechnungen unterhalb bestimmter Grenzen, ist aber eine Untermenge der EN 16931.
4. **EN 16931 (COMFORT)**: Das Standardprofil. Es bildet die europäische Norm vollständig ab und ist für den grenzüberschreitenden Verkehr sowie für B2G (Business to Government) geeignet. Dies ist das Zielformat für den *SmartBillConverter*.
5. **EXTENDED**: Enthält zusätzliche branchenspezifische Erweiterungen, die über die Norm hinausgehen.

2.2.3. CII-Struktur (Cross Industry Invoice)

Das XML-Format innerhalb von ZUGFeRD basiert auf der *Cross Industry Invoice* (CII) Syntax der UN/CEFACT. Dies ist ein globaler Standard für Lieferketten-Daten. Die Struktur ist extrem granular und tief verschachtelt. Ein einfacher Preis ist nicht nur eine Zahl, sondern ein komplexes Objekt mit Betrag, Währung, Basismenge und Einheitencode.

Beispielhafte Struktur-Tiefe in CII: `SupplyChainTradeTransaction` → `IncludedSupplyChainTra`
→ `SpecifiedLineTradeAgreement` → `NetPriceProductTradePrice` → `ChargeAmount`.
Diese Komplexität macht die Implementierung eines Mappers anspruchsvoll, da hunderte von Pfaden korrekt befüllt werden müssen, um Validierungsfehler zu vermeiden.[2] Im folgenden Listing ist ein Ausschnitt einer ZUGFeRD-XML dargestellt, der die Komplexität im Vergleich zu ebInterface verdeutlicht. Man beachte die tiefen Verschachtelungen für einfache Informationen wie den Steuerbetrag.

[2]Vgl. FeRD e.V.: *ZUGFeRD 2.3 - Das Datenformat für elektronische Rechnungen*, https://www.ferd-net.de/standards/zugferd-2.3/index.html, letzter Zugriff am 19.12.2025

Listing 2: Ausschnitt einer ZUGFeRD 2.3 (CII) Rechnung

```
<rsm:CrossIndustryInvoice
    xmlns:rsm="urn:un:unece:uncefact:data:standard:CrossIndustryInvoice:100" ...>
  <rsm:SupplyChainTradeTransaction>
    <ram:IncludedSupplyChainTradeLineItem>
      <ram:SpecifiedLineTradeAgreement>
        <ram:NetPriceProductTradePrice>
          <ram:ChargeAmount>100.00</ram:ChargeAmount>
        </ram:NetPriceProductTradePrice>
      </ram:SpecifiedLineTradeAgreement>
      <ram:SpecifiedLineTradeSettlement>
        <ram:ApplicableTradeTax>
          <ram:TypeCode>VAT</ram:TypeCode>
          <ram:CategoryCode>S</ram:CategoryCode>
          <ram:RateApplicablePercent>19.00</ram:RateApplicablePercent>
        </ram:ApplicableTradeTax>
      </ram:SpecifiedLineTradeSettlement>
    </ram:IncludedSupplyChainTradeLineItem>
    <ram:ApplicableHeaderTradeSettlement>
      <ram:SpecifiedTradeSettlementHeaderMonetarySummation>
        <ram:LineTotalAmount>100.00</ram:LineTotalAmount>
        <ram:TaxBasisTotalAmount>100.00</ram:TaxBasisTotalAmount>
        <ram:TaxTotalAmount currencyID="EUR">19.00</ram:TaxTotalAmount>
        <ram:GrandTotalAmount>119.00</ram:GrandTotalAmount>
      </ram:SpecifiedTradeSettlementHeaderMonetarySummation>
    </ram:ApplicableHeaderTradeSettlement>
  </rsm:SupplyChainTradeTransaction>
</rsm:CrossIndustryInvoice>
```

2.3. UBL vs. CII: Mapping-Überlegungen und Trade-offs

Die europäische Norm EN 16931 lässt zwei XML-Syntaxen zu: UBL (Universal Business Language) und UN/CEFACT CII. Während Netzwerke wie Peppol (Pan-European Public Procurement OnLine) historisch stark auf UBL (ISO/IEC 19845) setzen, hat sich das Forum elektronische Rechnung Deutschland (FeRD) bei ZUGFeRD für CII entschieden.

2.3.1. Strukturelle Unterschiede

UBL ist dokumentenzentriert. Es gibt eigene Schemas für `Order`, `Invoice`, `DespatchAdvice`. Die Struktur ist oft flacher und pragmatischer. **CII** ist prozesszentriert. Es versucht, alle Aspekte der Lieferkette in einem universellen Modell abzubilden. Dies führt zu einer höheren Abstraktion und Verschachtelung.

Ein konkretes Beispiel ist die Handhabung von Steuern:

- In **UBL** werden Steuern oft direkt auf Zeilenebene referenziert (`ClassifiedTaxCategory`).
- In **CII** gibt es komplexe `ApplicableTradeTax`-Strukturen, die sowohl auf Dokumentenebene (Summen) als auch auf Zeilenebene (Referenzen) konsistent gehalten werden müssen.

2.3.2. Herausforderungen für den Konverter

Für den *SmartBillConverter* bedeutet dies, dass das interne Datenmodell (das C# `Invoice`-Objekt) als "Intermediate Representation" fungieren muss. Es darf nicht zu stark an ebInterface gekoppelt sein, da sonst das Mapping auf CII extrem schwierig wird. Das Mapping von ebInterface (XML) direkt auf ZUGFeRD (CII) ist nicht trivial, da die semantischen Bäume unterschiedlich sind. ebInterface gruppiert oft logisch (z.B. "Biller"), während CII funktional gruppiert (z.B. "SupplyChainTradeTransaction"). Die Entscheidung, eine eigene interne Repräsentation zu nutzen, die von der KI befüllt wird und dann in beide Formate serialisiert werden kann, ist daher architektonisch notwendig. Tabelle 1 fasst die wesentlichen Unterschiede zusammen, die bei der Implementierung berücksichtigt wurden.

Merkmal	**ebInterface 6.1**
Basis-Standard	National (AUSTRIAPRO)
Dateiformat	Reines XML
Struktur-Tiefe	Flach bis Mittel
Steuer-Logik	Zentraler Tax-Block Header)
Pflichtfelder	Fokus auf UStG (AT)
Visualisierung	Benötigt Stylesheet

Tabelle 1.: Vergleich zwischen ebInterface und ZUGFeRD

2.4. XSD-Validierung vs. Geschäftsregeln (BR-S-08 etc.)

Die Qualitätssicherung einer generierten E-Rechnung erfolgt in zwei Stufen. Ein Dokument kann technisch valide sein, aber inhaltlich falsch.

2.4.1. Syntaktische Validierung (XSD)

Die XML Schema Definition (XSD) prüft die Grammatik der Datei.

- Sind alle Pflicht-Tags vorhanden?
- Haben Datumsfelder das Format YYYY-MM-DD?
- Sind numerische Werte korrekt formatiert?

- Stimmt die Reihenfolge der Elemente? (Besonders in CII ist die Reihenfolge strikt vorgegeben).

Ein Verstoß gegen das XSD führt dazu, dass die Datei vom Empfängersystem meist gar nicht erst eingelesen werden kann ("Technical Rejection").

2.4.2. Semantische Validierung (Business Rules)

Selbst wenn das XML wohlgeformt ist, kann der Inhalt unlogisch sein. Die EN 16931 definiert daher eine Liste von Geschäftsregeln (Business Rules), die eingehalten werden müssen. Diese werden oft mittels *Schematron* geprüft.

Ein prominentes und im Projektverlauf problematisches Beispiel ist die Regel **BR-S-08** (Value Added Tax Breakdown). Diese Regel besagt: *Für jeden unterschiedlichen Steuercode und Steuersatz, der in den Rechnungspositionen verwendet wird, muss genau eine Zusammenfassung auf Dokumentenebene existieren, und die Summe der Steuerbeträge muss rechnerisch korrekt sein.*

Das Problem bei der Generierung durch KI (LLMs) ist, dass LLMs statistische Modelle sind, keine Taschenrechner. Ein LLM kann problemlos schreiben:

- Position 1: 100 Euro, 20% MwSt
- Position 2: 200 Euro, 20% MwSt
- Steuer-Summe: 55 Euro (statt korrekt 60 Euro)

Das LLM "schätzt" oder "halluziniert" die Summe oft, anstatt sie zu berechnen. Daher darf die Berechnung der Summen und Steuern **niemals** der KI überlassen werden. Die KI extrahiert die Einzelwerte (Menge, Preis, Steuersatz), aber die Aggregation und die Erstellung des `TaxBreakdown` für die Regel BR-S-08 muss durch deterministische Algorithmen im Backend (C#) erfolgen. Nur so kann die Validierung gegen Schematron-Regeln bestanden werden.[3]

[3]Vgl. CEN - European Committee for Standardization: *EN 16931-1:2017 Electronic invoicing - Semantic data model*, https://standards.cen.eu/dyn/www/f?p=204:110:0::::FSP_PROJECT:60602&cs=1B61B766636F9FB34B7DBD72CE9026C72, letzter Zugriff am 19.12.2025

3. Dokumentenverarbeitung und KI-Grundlagen

Die automatisierte Verarbeitung von Rechnungen ist ein klassisches Problem der Informatik, das durch moderne KI-Ansätze neu gelöst wird. Traditionelle Ansätze basierten auf Templates (Schablonen), bei denen für jeden Lieferanten definiert wurde, an welchen Koordinaten die Rechnungsnummer steht. Dieser Ansatz ist jedoch fragil und skaliert nicht. Der *SmartBillConverter* setzt daher auf einen generischen Ansatz mittels Large Language Models (LLMs). Dieses Kapitel beleuchtet die technologischen Grundlagen.

3.1. PDFs und Textextraktion

3.1.1. Die Natur des PDF-Formats

Das Portable Document Format (PDF) wurde von Adobe entwickelt, um Dokumente geräteunabhängig darzustellen. Es ist ein Seitenbeschreibungssprache. Das bedeutet, ein PDF speichert primär Anweisungen wie "Zeichne den Buchstaben "A" an Koordinate (100, 200) in Schriftart Helvetica". Es speichert *nicht* notwendigerweise die Information, dass dieses "A" Teil des Wortes "Rechnung" ist oder dass dieses Wort Teil einer Tabelle ist.

Für die Textextraktion ergeben sich daraus massive Probleme:

- **Verlust der Lesereihenfolge**: In einem PDF-Stream können die Zeichenbefehle in beliebiger Reihenfolge stehen. Ein zweispaltiger Text kann im Stream so gespeichert sein, dass erst die erste Zeile der linken Spalte, dann die erste Zeile der rechten Spalte kommt. Ein naiver Extraktor liest dann "Rechnungs Datum: 01.01.2024" als "Rechnungs 01.01.2024 Datum:".

- **Fehlende Wortgrenzen**: Oft werden Wörter nicht als String gespeichert, sondern jeder Buchstabe einzeln positioniert (Kerning). Leerzeichen sind oft gar keine Zeichen, sondern einfach Lücken in den Koordinaten.
- **Encoding-Probleme**: Manchmal nutzen PDFs benutzerdefinierte Encodings, sodass der Buchstabe "A" im Code als "X" gespeichert ist, aber visuell als "A" dargestellt wird. Ohne korrekte ToUnicode-Map ist nur "Datensalat" extrahierbar.

3.1.2. Lösungsansatz mit PdfPig

Im Projekt wird die Bibliothek *PdfPig* eingesetzt. Sie versucht, die logische Struktur aus den geometrischen Informationen zu rekonstruieren. PdfPig analysiert die "Bounding Boxes" aller Buchstaben. Durch Heuristiken (Abstandsanalyse) werden Buchstaben zu Wörtern und Wörter zu Zeilen zusammengefügt. Ein spezifisches Problem bei Rechnungen sind Tabellen. Da Tabellenlinien im PDF nur Vektorgrafiken sind und keine logische Verbindung zum Text haben, ist es für Algorithmen schwer zu erkennen, welche Zahl zu welcher Spalte gehört, besonders wenn Spalten rechtsbündig sind und der Textabstand variiert. Dennoch liefert die direkte Extraktion aus dem PDF-Stream (wenn möglich) immer präzisere Daten als der Umweg über OCR, da keine Zeichenverwechslungen (z.B. "8" vs "B") auftreten können.[4]

3.2. OCR mit Tesseract

Wenn Rechnungen nicht als digitales PDF, sondern als Scan (Rastergrafik) vorliegen, greift die Textextraktion ins Leere. Hier ist Optical Character Recognition (OCR) erforderlich.

3.2.1. Funktionsweise von Tesseract

Tesseract ist eine der führenden Open-Source-OCR-Engines, ursprünglich von HP entwickelt und heute von Google gepflegt. Moderne Versionen (ab 4.0) basieren auf LSTM (Long Short-Term Memory) neuronalen Netzen, einer Form von Recurrent Neural Networks (RNNs), die besonders gut für Sequenzdaten wie Text geeignet sind. Tesseract analysiert das Bild in mehreren Schritten: 1. **Layout Analysis**: Finden

[4]Vgl. UglyToad: *PdfPig - Read and extract text and other content from PDFs in C# (port of PdfBox)*, https://github.com/UglyToad/PdfPig, letzter Zugriff am 19.12.2025

von Textblöcke und Zeilen. 2. **Baseline Fitting**: Erkennen der Grundlinie einer Textzeile (wichtig bei schiefen Scans). 3. **Character Recognition**: Das neuronale Netz klassifiziert kleine Bildausschnitte als Zeichen. 4. **Word Recognition**: Wörterbuch-Abgleich, um die Wahrscheinlichkeit von Wortkombinationen zu bewerten.

3.2.2. Einflussfaktoren auf die Qualität

Die Qualität der OCR hängt massiv von der Vorverarbeitung ab:

- **Auflösung**: Unter 300 DPI sinkt die Erkennungsrate drastisch.
- **Binarisierung**: Die Umwandlung von Graustufen in reines Schwarz-Weiß (Thresholding) muss adaptiv erfolgen, um Schatten oder ungleichmäßige Beleuchtung auszugleichen.
- **Deskewing**: Schon eine Drehung um 1-2 Grad kann die Zeilenerkennung stören. Das Bild muss rechnerisch gerade gerückt werden.

Im Projektkontext werden für Tesseract spezifische Sprachpakete (`deu.traineddata`) geladen, um deutsche Umlaute und typische Vokabeln korrekt zu erkennen. Dennoch bleibt OCR fehleranfällig, weshalb die nachgelagerte KI-Korrektur essenziell ist.

3.3. LLMs für Informationsextraktion

Der Paradigmenwechsel in der Dokumentenverarbeitung kommt durch Large Language Models (LLMs). Anstatt Regeln zu programmieren ("Suche nach "Rechnungs-Nr." und nimm das Wort rechts daneben"), übergibt man dem LLM den gesamten unstrukturierten Text und bittet es, die Daten zu strukturieren.

3.3.1. Architektur und Funktionsweise

LLMs basieren auf der Transformer-Architektur. Sie wurden auf riesigen Textmengen trainiert und haben gelernt, statistische Zusammenhänge zwischen Wörtern (Tokens) vorherzusagen. Für die Extraktion ist die Fähigkeit des "In-Context Learning" entscheidend. Das Modell versteht den Kontext einer Rechnung. Es weiß, dass eine IBAN meist in der Nähe von "Bankverbindung" steht und wie eine IBAN aussieht, ohne dass man ihm einen Regex geben muss. Es kann Synonyme auflösen: Egal ob auf der Rechnung

"Total", "Gesamtbetrag", "Zahlbetrag" oder "Summe" steht, das LLM kann es auf das Feld `TotalAmount` mappen.

3.3.2. Modell-Vergleich und Evaluation

Im Rahmen der Entwicklung wurden verschiedene Modelle evaluiert (siehe Projektdokumentation):

- **Gemini 2.5 Flash**: Ein sehr schnelles und kosteneffizientes Modell von Google. Es zeigte im Projekt die beste Balance aus Geschwindigkeit und JSON-Konformität. Es hat ein großes Kontextfenster, was für lange Rechnungen wichtig ist.
- **Mistral (verschiedene Größen)**: Open-Source-Modelle, die lokal oder via API laufen können. Während große Modelle (Mistral Large) gut performen, neigen kleinere Modelle (7B) dazu, das JSON-Schema zu verletzen oder komplexe Tabellen zu halluzinieren.
- **Qwen**: Ein starkes Modell, das jedoch im Test oft Prompt-Logging und Training erforderte, was Datenschutzbedenken aufwirft.

3.3.3. Risiken: Halluzinationen

Das größte Risiko bei LLMs ist die Halluzination. Das Modell ist darauf trainiert, eine "plausible Fortsetzung" des Textes zu generieren. Wenn auf der Rechnung kein Lieferdatum steht, das JSON-Schema aber ein Lieferdatum verlangt, "erfindet" das Modell oft einfach eines (z.B. das Rechnungsdatum), um den User zufrieden zu stellen. Dies ist fatal für eine Finanzanwendung. Daher muss der Prompt so gestaltet sein, dass das Modell explizit "null" oder "nicht vorhanden" ausgibt, anstatt zu raten.

3.4. Prompt-Design für deterministische Ausgabe

Prompt Engineering ist die Kunst, die Eingabe so zu formulieren, dass das Modell das gewünschte Ergebnis liefert. Für die API-basierte Extraktion ist Determinismus das Ziel.

3.4.1. Techniken

- **System Prompting**: Dem Modell wird eine klare Rolle zugewiesen: "Du bist ein strikter Datenextraktions-Assistent. Du antwortest nur mit JSON. Du fügst keine Erklärungen hinzu."
- **JSON Mode / Function Calling**: Moderne APIs (wie OpenAI oder Gemini) bieten Modi an, die garantieren, dass der Output valides JSON ist. Das Modell wird gezwungen, Tokens zu generieren, die der Syntax entsprechen.
- **Schema Injection**: Das gewünschte JSON-Schema wird im Prompt mitgegeben. "Extrahiere die Daten in folgendes Format: { "invoiceNumber": "string", ... }". **Chain-of-Thought Unterdrückung**: Während bei Logikaufgaben "Denk nach Schritt für Schritt" hilft, ist es bei der Extraktion oft hinderlich, da es den Output "verunreinigt". Wir wollen nur die Daten.

Ein Beispiel für ein solches JSON-Schema, wie es im *SmartBillConverter* verwendet wird, zeigt Listing 3. Es definiert strikte Typen für die Extraktion.

Listing 3: JSON-Schema für die KI-Extraktion

```
{
  "type": "object",
  "properties": {
    "invoiceNumber": { "type": "string" },
    "invoiceDate": { "type": "string", "format": "date" },
    "totalAmount": { "type": "number" },
    "currency": { "type": "string", "enum": ["EUR", "USD"] },
    "items": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "description": { "type": "string" },
          "quantity": { "type": "number" },
          "unitPrice": { "type": "number" },
          "taxRate": { "type": "number" }
        }
      }
    }
  },
  "required": ["invoiceNumber", "invoiceDate", "totalAmount", "items"]
}
```

Ein spezifisches Problem im Projekt war die Konsistenz. Manchmal lieferte das gleiche Modell beim gleichen Prompt leicht unterschiedliche Ergebnisse (z.B. Datumsformat mal YYYY-MM-DD, mal DD.MM.YYYY). Dies muss durch strikte Anweisungen ("Formatiere alle Daten als ISO 8601") und niedrige Temperature-Einstellungen (Temperature = 0) minimiert werden.

3.5. Datenschutz und Sicherheit in KI-APIs

Die Nutzung von Cloud-basierten KI-APIs (wie Google Vertex AI oder OpenAI API) für Rechnungsdaten wirft Datenschutzfragen auf, da Rechnungen personenbezogene Daten (Namen, Adressen) und Geschäftsgeheimnisse (Preise, Lieferantenbeziehungen) enthalten.

3.5.1. Risikoanalyse

- **Training auf Kundendaten**: Das größte Risiko ist, dass der Anbieter (Google, OpenAI) die hochgeladenen Rechnungen nutzt, um seine Modelle zu trainieren. Dies könnte dazu führen, dass das Modell in Zukunft Informationen aus diesen Rechnungen "weiß". Enterprise-Verträge schließen dies meist explizit aus ("Zero Data Retention" für Training).
- **Datenübertragung**: Die Daten verlassen das geschützte Unternehmensnetzwerk. Eine Ende-zu-Ende-Verschlüsselung (TLS 1.3) ist Standard, aber der API-Endpunkt entschlüsselt die Daten zur Verarbeitung.
- **Serverstandort**: Nach DSGVO sollten personenbezogene Daten idealerweise den Europäischen Wirtschaftsraum (EWR) nicht verlassen. Bei US-Anbietern ist auf Angemessenheitsbeschlüsse (Data Privacy Framework) zu achten.

3.5.2. Lokale Alternativen

Um diese Risiken zu eliminieren, wäre der Einsatz lokaler LLMs (z.B. Llama 3, Mistral) via Ollama oder LM Studio eine Option. Diese laufen auf der eigenen Hardware (On-Premise). Der Nachteil ist der hohe Ressourcenbedarf (GPU-VRAM) und die oft schlechtere Leistung im Vergleich zu den riesigen Cloud-Modellen. Ein lokales 7B-Modell hat oft Schwierigkeiten mit komplexen, mehrseitigen Rechnungen, die ein großes Kontextfenster benötigen. Für den *SmartBillConverter* wurde aufgrund der Qualität und Entwicklungsgeschwindigkeit auf Cloud-APIs gesetzt, wobei im Produktionsbetrieb auf Enterprise-Lizenzen mit entsprechenden Datenschutzgarantien gewechselt werden müsste.

Teil II.

Systemarchitektur und Design

4. Gesamtarchitektur

[Dieses Kapitel wird von Sebastian Radić verfasst und beschreibt die Gesamtarchitektur des Systems.]

4.1. Kontextdiagramm und Use Cases

[Wird von Sebastian Radić verfasst]

4.2. Komponentenübersicht

[Wird von Sebastian Radić verfasst: Backend-API, Services, Datenbank, Frontend]

4.3. Datenfluss

[Wird von Sebastian Radić verfasst: PDF/Image → Text → KI-JSON → Mapping → XML → Validierung]

5. Technologie-Stack und Begründung

[Dieses Kapitel wird von Sebastian Radić verfasst und begründet die Technologiewahl.]

5.1. Backend: ASP.NET Core 8, C#, Swagger/OpenAPI

[Wird von Sebastian Radić verfasst]

5.2. Datenhaltung: PostgreSQL (Docker), EF Core

[Wird von Sebastian Radić verfasst]

5.3. Frontend: Angular (smart-bill-ui)

[Wird von Sebastian Radić verfasst]

5.4. Bibliotheken: PdfPig, Tesseract, XmlSerializer, XSD-Validierung

[Wird von Sebastian Radić verfasst]

5.5. Entwicklungsumgebung: Docker, Skripte, Logging, Konfiguration

[Wird von Sebastian Radić verfasst]

Teil III.

Implementierung (Backend)

6. Backend-Anwendungsstruktur

[Dieses Kapitel wird von Sebastian Radić verfasst und beschreibt die Backend-Struktur.]

6.1. Controller

[Wird von Sebastian Radić verfasst: Invoice, Processing, ZUGFeRD, Image Processing, Download]

6.2. Services

[Wird von Sebastian Radić verfasst: Extraktion, OCR, LLM-Konvertierung, Serialisierung, Validierung]

6.3. Datenzugriff

[Wird von Sebastian Radić verfasst: DbContext, Migrations, Repository Pattern]

6.4. Konfiguration und Secrets Management

[Wird von Sebastian Radić verfasst]

6.5. Fehlerbehandlung und Resilienzstrategie

[Wird von Sebastian Radić verfasst]

7. Pipeline zur PDF-Textextraktion

[Dieses Kapitel wird von Sebastian Radić verfasst und beschreibt die PDF-Textextraktion.]

7.1. PdfPig-Integration

[Wird von Sebastian Radić verfasst]

7.2. Textbereinigung und Normalisierung

[Wird von Sebastian Radić verfasst]

7.3. Fehlerbehandlung und Fallback-Strategien

[Wird von Sebastian Radić verfasst]

7.4. Bekannte Sonderfälle

[Wird von Sebastian Radić verfasst: gescannte PDFs, Tabellen, Mehrspalten]

8. OCR-Pipeline für Bilder

[Dieses Kapitel wird von Sebastian Radić verfasst und beschreibt die OCR-Pipeline.]

8.1. Einrichtung von Tesseract

[Wird von Sebastian Radić verfasst: Skripte, Modelle]

8.2. OCR-Endpunkte und Integration

[Wird von Sebastian Radić verfasst]

8.3. Qualität, Sprachpakete und Nachbearbeitung

[Wird von Sebastian Radić verfasst]

9. KI-Normalisierung und Mapping

[Dieses Kapitel wird von Sebastian Radić verfasst und beschreibt die KI-Normalisierung.]

9.1. Prompt-Strategien

[Wird von Sebastian Radić verfasst: Constraints, JSON-Schema]

9.2. Modelldiskussion

[Wird von Sebastian Radić verfasst: Gemini 2.5 Flash, Mistral, Qwen, Gemma]

9.3. Parsing und Validierung der KI-Ausgaben

[Wird von Sebastian Radić verfasst]

9.4. Mapping auf Domänen- und XML-Modelle

[Wird von Sebastian Radić verfasst]

10. Generierung von ebInterface 6.1

[Dieses Kapitel wird von Sebastian Radić verfasst und beschreibt die ebInterface-Generierung.]

10.1. Objektmodell-Mapping

[Wird von Sebastian Radić verfasst: Käufer, Lieferant, Bank, Positionen, Steuern]

10.2. Serialisierung mit XmlSerializer

[Wird von Sebastian Radić verfasst: Namespaces]

10.3. XSD-Validierung und Korrekturstrategien

[Wird von Sebastian Radić verfasst]

11. Generierung von ZUGFeRD 2.3 / EN 16931

[Dieses Kapitel wird von Sebastian Radić verfasst und beschreibt die ZUGFeRD-Generierung.]

11.1. CII-Mapping

[Wird von Sebastian Radić verfasst: Dokumentkontext, Parteien, Positionen, Steuern]

11.2. Serialisierung, Namespace/Order-Postprocessing

[Wird von Sebastian Radić verfasst]

11.3. Fallback-Strategien für Minimalvalidität

[Wird von Sebastian Radić verfasst]

Teil IV.

Frontend-Anforderungsanalyse und Architektur

12. Anforderungsanalyse Frontend

Die Entwicklung einer benutzerfreundlichen Weboberfläche ist ein zentraler Erfolgsfaktor für die Akzeptanz des *SmartBillConverter*. Während das Backend die komplexen Aufgaben der Dokumentenverarbeitung und Format-Konvertierung übernimmt, muss das Frontend eine intuitive Schnittstelle bieten. Dieses Kapitel dokumentiert die systematische Analyse der Frontend-Anforderungen, die Modellierung der Datenstrukturen und die Benutzerinteraktionen.

12.1. Definition der Systemanforderungen (Frontend)

Die Systemanforderungen beschreiben, welche Funktionen die Anwendung erfüllen muss und welche Qualitätsmerkmale sie aufweisen soll. Sie gliedern sich in funktionale und nicht-funktionale Anforderungen.

12.1.1. Funktionale Anforderungen

Die funktionalen Anforderungen an das Frontend des *SmartBillConverter*-Projekts definieren die Kernfunktionen des Systems:

- **Dokumenten-Upload**: Unterstützt PDF-Dateien und Bildformate (PNG, JPEG, BMP, TIFF) per Drag-and-Drop oder Dateiauswahl. Mehrere Dateien können gleichzeitig hochgeladen werden (bis zu 10 MB pro Datei). Der Upload-Fortschritt wird angezeigt. Falsche Dateiformate werden mit einer Fehlermeldung abgelehnt. Nutzt HTML5 File API und FormData.
- **Format-Auswahl**: Auswahl zwischen ebInterface 6.1 (Österreich) und ZUGFeRD 2.3 (Deutschland) über Radio-Buttons. Tooltips erklären die Unterschiede zwischen den Formaten.

- **Fortschrittsanzeige**: Zeigt den Verarbeitungsstatus mit Progress Bar (0-100%) und farbigen Status-Badges (wartend, verarbeitend, abgeschlossen, fehlgeschlagen). Grün bedeutet Erfolg, rot bedeutet Fehler, gelb bedeutet Warnung. Bei langen Verarbeitungen wird ein Spinner angezeigt.
- **Dokumenten-Vorschau**: PDFs werden mit ng2-pdf-viewer angezeigt (mit Zoom und Seitennavigation). Bilder werden automatisch skaliert. Das generierte XML wird mit Syntax-Highlighting angezeigt.
- **Download-Funktionalität**: XML-Dateien können einzeln oder als Batch heruntergeladen werden. Dateinamen folgen dem Schema `invoice_12345_ebInterface.xml`. Der Download nutzt Blob-URLs. Mehrere Dateien werden als ZIP-Datei gebündelt.
- **Fehlerbehandlung**: Dateien werden vor dem Upload validiert (Größe, Format). Bei Fehlern werden verständliche Fehlermeldungen angezeigt. Netzwerkfehler können mit einem Retry-Button wiederholt werden.

12.1.2. Nicht-funktionale Anforderungen

Diese Anforderungen definieren die Qualität und Benutzerfreundlichkeit der *SmartBill-Converter*-Anwendung:

- **Benutzerfreundlichkeit**: Die Bedienung soll selbsterklärend sein. Ein neuer Benutzer soll den Upload-Workflow in unter 60 Sekunden verstehen. Fehleingaben werden durch Validierung verhindert.[5]
- **Geschwindigkeit**: Die Seite soll in unter 2 Sekunden laden. Benutzerinteraktionen sollen unter 100ms reagieren.
- **Geräteunterstützung**: Die Anwendung ist primär für Desktop optimiert, funktioniert aber auch auf Tablets und Smartphones.
- **Browser-Unterstützung**: Die Anwendung funktioniert in Chrome, Firefox, Safari und Edge (aktuelle Versionen).[6]

[5]Vgl. Nielsen Norman Group: *Drag and Drop: How to Design for Ease of Use*, https://www.nngroup.com/articles/drag-drop/, letzter Zugriff am 19.12.2025

[6]Vgl. W3C: *Web Content Accessibility Guidelines (WCAG) 2.1*, https://www.w3.org/TR/WCAG21/, letzter Zugriff am 19.12.2025

12.2. Entity-Relationship-Diagramm (Datenmodell-Sicht)

Das Entity-Relationship-Diagramm modelliert die zentrale Datenstruktur des *SmartBillConverter*-Systems. Die `Invoice`-Entität repräsentiert im entwickelten Projekt eine verarbeitete Rechnungsdatei mit allen extrahierten Kerninformationen:

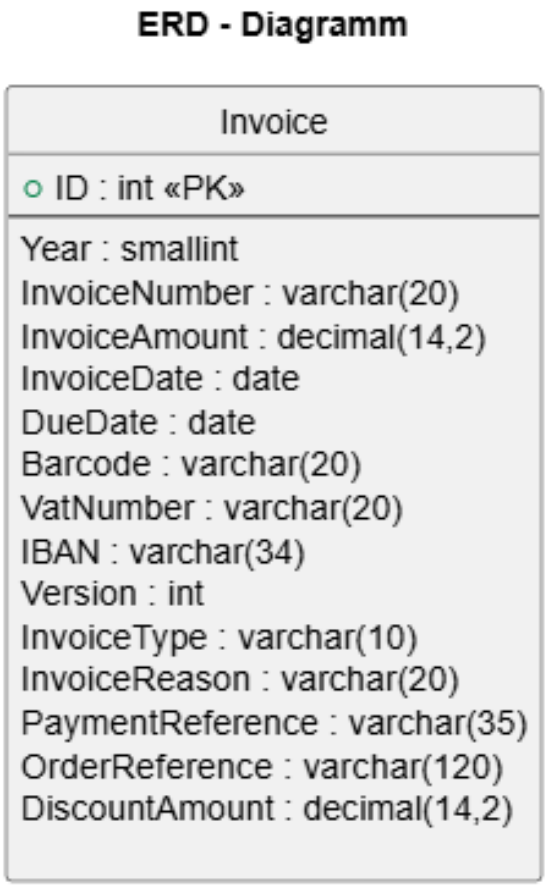

Abbildung 1.: Eigene Darstellung: Entity-Relationship-Diagramm der Invoice-Entität

12.2.1. Detaillierte Attributbeschreibung

Das Datenmodell des *SmartBillConverter*-Projekts folgt den gesetzlichen Anforderungen des österreichischen UStG. Die einzelnen Attribute der im Projekt implementierten `Invoice`-Entität (siehe Abbildung 1) haben folgende Bedeutung:

- **ID** (`int`, Primary Key): Eindeutige ID für jede Rechnung in der Datenbank. Wird automatisch hochgezählt.
- **Year** (`smallint`): Rechnungsjahr, extrahiert aus dem Rechnungsdatum. Wird für Jahresabfragen und Archivierung verwendet.

- **InvoiceNumber** (`varchar(20)`): Eindeutige Rechnungsnummer des Rechnungsstellers. Gemäß § 11 Abs. 1 Z 2 UStG muss jede Rechnung eine fortlaufende Nummer enthalten. Unterstützt Formate wie `RE-2025-00142` oder `INV/2025/0001`.
- **InvoiceAmount** (`decimal(14,2)`): Bruttorechnungsbetrag inklusive Umsatzsteuer. Mit 14 Stellen und 2 Nachkommastellen können Beträge bis zu 999.999.999.999,99 EUR gespeichert werden.
- **InvoiceDate** (`date`): Rechnungsdatum, an dem die Rechnung ausgestellt wurde. Wird für die Fälligkeitsberechnung und die Zuordnung zu Steuerperioden verwendet.
- **DueDate** (`date`): Fälligkeitsdatum, bis zu dem die Zahlung erwartet wird. Wird aus dem Zahlungsziel berechnet (z.B. „Zahlbar innerhalb 14 Tagen“). Wichtig für Mahnungen und Liquiditätsplanung.
- **VatNumber** (`varchar(20)`): Umsatzsteuer-Identifikationsnummer (UID) des Rechnungsstellers im Format `ATU12345678`. Muss bei EU-Lieferungen angegeben werden (§ 11 Abs. 1 Z 5 UStG).
- **IBAN** (`varchar(34)`): Bankkontonummer für Überweisungen. Maximal 34 Zeichen nach IBAN-Standard. Beispiel: `AT611904300234573201`.
- **Version** (`int`): Versionsnummer der Rechnung für Änderungsnachverfolgung. Beginnt bei 1 und wird bei jeder Korrektur erhöht.
- **InvoiceType** (`varchar(10)`): Rechnungstyp, z.B. `INVOICE` (Rechnung), `CREDIT` (Gutschrift) oder `ADVANCE` (Anzahlungsrechnung). Wichtig für die Buchhaltung.
- **PaymentReference** (`varchar(35)`): Zahlungsreferenz für die Zuordnung von Zahlungseingängen. Beispiel: `RF18539007547034`.
- **DiscountAmount** (`decimal(14,2)`): Skontobetrag bei vorzeitiger Zahlung. Oft 2-3% bei Zahlung innerhalb von 7-10 Tagen.

Diese Datenstruktur bildet die Grundlage für die persistente Speicherung aller verarbeiteten Rechnungen und erlaubt schnelle Abfragen, Reporting und Archivierung.

Im *SmartBillConverter*-Frontend wird ein TypeScript-Interface verwendet, das die Backend-Entität des Projekts widerspiegelt:

Listing 4: TypeScript Invoice-Interface

```
1 export interface Invoice {
```

```
    id: number;
    invoiceNumber: string;
    invoiceAmount: number;
    invoiceDate: string;
    dueDate: string;
    vatNumber: string;
    iban: string;

    // UI-spezifische Felder
    ebInterfaceXml?: string;
    isValidXml?: boolean;
}
```

12.3. Use-Case-Diagramm (Benutzerinteraktionen)

Use-Case-Diagramme visualisieren die Interaktionen zwischen Benutzern und dem System. Abbildung 2 zeigt die Hauptanwendungsfälle des *SmartBillConverter*-Projekts:

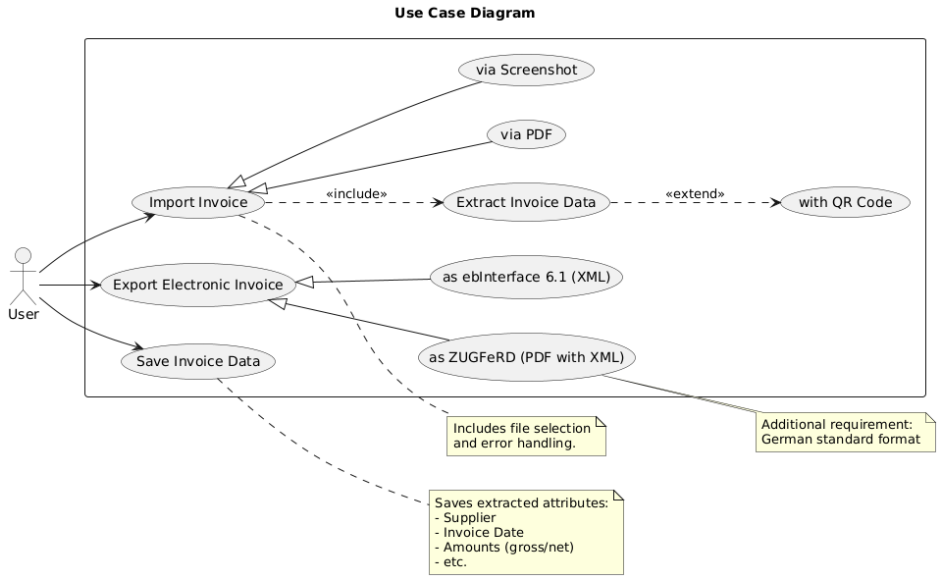

Abbildung 2.: Eigene Darstellung: Use-Case-Diagramm des SmartBillConverter

12.3.1. Beschreibung der Use-Cases

Die in Abbildung 2 dargestellten Hauptanwendungsfälle des *SmartBillConverter*-Systems werden im Folgenden detailliert beschrieben:

UC1: Import Invoice – Der primäre Use-Case (siehe Abbildung 2) ermöglicht das Hochladen von Rechnungsdokumenten. Der Benutzer wählt zwischen zwei Importvarianten:

- **UC1.1: via PDF** (`«extend»`): Import einer digitalisierten PDF-Rechnung. Dies ist der Standardfall für elektronisch erstellte Rechnungen. Das System extrahiert Text mittels PDF-Parser (PDF.js) ohne OCR-Verarbeitung.

- **UC1.2: via Screenshot** («extend»): Import eines gescannten Bildes (PNG, JPEG, BMP, TIFF). Für diese Variante ist eine OCR-Verarbeitung (Tesseract) notwendig, um den Text aus dem Bild zu extrahieren.

UC2: Extract Invoice Data («include») – Dieser Use-Case ist in UC1 eingebettet und wird automatisch nach dem Import ausgeführt. Die Extraktion erfolgt in mehreren Schritten: (1) Text-Extraktion (PDF-Parser oder OCR), (2) Normalisierung durch LLM-basierte Analyse (Gemini/ChatGPT), (3) Extraktion strukturierter Daten (Rechnungsnummer, Betrag, Datum, etc.), (4) Validierung der extrahierten Daten gegen Geschäftsregeln.

UC3: Save Invoice Data – Nach erfolgreicher Extraktion werden die Rechnungsdaten in der PostgreSQL-Datenbank persistiert. Dieser Use-Case umfasst: (1) Validierung der Vollständigkeit aller Pflichtfelder, (2) Prüfung auf Duplikate anhand der Rechnungsnummer, (3) Speicherung der Invoice-Entität mit allen Attributen, (4) Rückgabe der generierten Datenbank-ID an das Frontend.

UC4: Export Electronic Invoice – Der finale Use-Case generiert die standardisierte elektronische Rechnung. Der Benutzer wählt das Zielformat:

- **UC4.1: as ebInterface 6.1 (XML)** («extend»): Export als reine XML-Datei nach österreichischem ebInterface-Standard. Die XML-Struktur folgt dem offiziellen XSD-Schema des Bundesrechenzentrums.
- **UC4.2: as ZUGFeRD (PDF with XML)** («extend»): Export als hybrides PDF/A-3-Dokument mit eingebetteter XML-Datei nach ZUGFeRD 2.3-Standard. Das sichtbare PDF entspricht der Original-Rechnung, während die maschinenlesbare XML-Datei im PDF eingebettet ist.

Der gesamte Workflow ist auf Einfachheit und Benutzerfreundlichkeit optimiert: Format wählen → Datei hochladen → Automatische Verarbeitung abwarten → XML herunterladen. Die durchschnittliche Bearbeitungszeit beträgt 5-15 Sekunden pro Rechnung, abhängig von der Dokumentenkomplexität.

12.4. Systemarchitektur (Frontend-Sicht)

Die Systemarchitektur des *SmartBillConverter*-Projekts folgt einer klassischen Drei-Schichten-Architektur mit klarer Trennung der Verantwortlichkeiten:

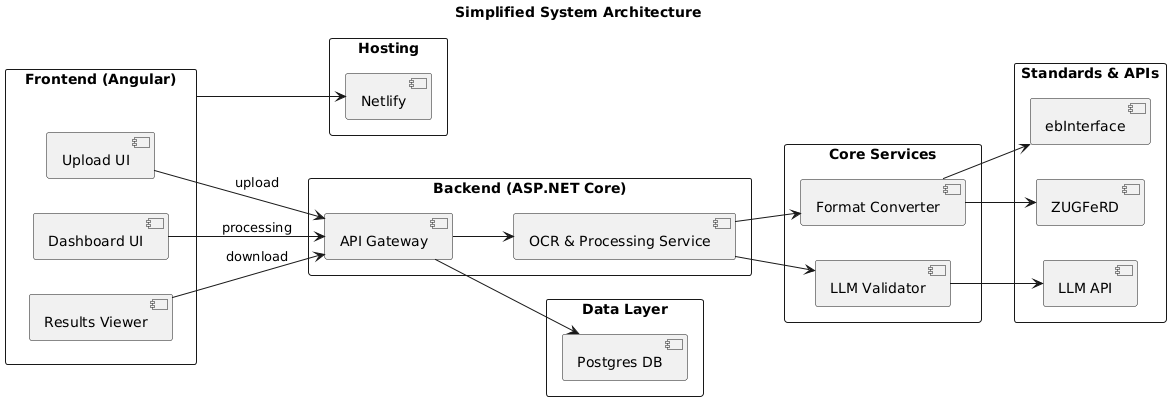

Abbildung 3.: Eigene Darstellung: Systemarchitektur des SmartBillConverter

Die Architektur (siehe Abbildung 3) besteht aus drei Schichten: (1) **Präsentationsschicht**: Angular-Webseite im Browser, (2) **Anwendungsschicht**: ASP.NET Core Web API mit Verarbeitungslogik, (3) **Datenschicht**: PostgreSQL-Datenbank. Der `InvoiceService` im Frontend verwaltet alle HTTP-Anfragen und nutzt RxJS Observables für asynchrone Datenverarbeitung.[7] Der Ablauf: Benutzer lädt Datei hoch → Frontend prüft und sendet an Backend → Backend extrahiert Text (PDF-Parser oder OCR) und normalisiert mit KI → XML wird erstellt und zurückgesendet → Nutzer lädt XML herunter.

[7]Vgl. Angular Documentation: *Observables in Angular*, https://angular.io/guide/observables-in-angular, letzter Zugriff am 19.12.2025

13. Frontend-Architektur & Technologie-Stack

Die Wahl des richtigen Technologie-Stacks ist wichtig für den Projekterfolg. Dieses Kapitel dokumentiert die Architekturentscheidungen und die technologische Grundlage des *SmartBillConverter*-Frontends.

13.1. Projektinitialisierung mit Angular CLI

Die Initialisierung des Projekts umfasst die Auswahl des passenden Frameworks und das Setup der Entwicklungsumgebung. Diese Entscheidungen beeinflussen die gesamte Entwicklung.

13.1.1. Framework-Evaluation

Für das *SmartBillConverter*-Frontend fiel die Wahl auf Angular 19 aus folgenden Gründen:

- **Firmenanforderungen**: Das Partnerunternehmen gab keine konkrete Framework-Vorgabe, sondern ermöglichte eine freie Technologiewahl für das Frontend.
- **Schulische Vorkenntnisse**: Angular wurde im laufenden Schuljahr ausführlich behandelt, wodurch eine solide Wissensbasis vorhanden war.
- **Backend-Kompatibilität**: Das vorgegebene C#-Backend harmoniert gut mit Angular, da beide auf TypeScript bzw. stark typisierten Sprachen basieren.
- **Webapplikations-Eignung**: Für die geforderte Single-Page-Webapplikation bietet Angular eine ausgereifte Lösung mit integriertem Routing, HTTP-Client und Dependency Injection.

13.1.2. Projektsetup

Die Initialisierung erfolgte über die Angular CLI:

Listing 5: Angular-Projektgenerierung

```
ng new smart-bill-ui --routing --style=css --strict
cd smart-bill-ui
ng serve
```

Der Befehl `ng serve` startet einen Entwicklungsserver mit Hot Module Replacement. Für Produktion wird ein optimierter Build mit AOT-Kompilierung, Tree Shaking und Minification erstellt.

13.2. Auswahl der Bibliotheken

Zusätzlich zum Angular-Framework werden externe Bibliotheken für spezifische Funktionen benötigt. Die wichtigsten Bibliotheken sind Bootstrap für das Styling und ng2-pdf-viewer für die PDF-Anzeige.

13.2.1. Bootstrap 5.3.7

Bootstrap wurde als CSS-Framework gewählt, weil es im schulischen Kontext bereits verwendet wurde und somit Vorkenntnisse im Umgang mit dem Grid-System, UI-Komponenten und Utility-Klassen vorhanden waren. Dies beschleunigte die Frontend-Entwicklung erheblich.[8]

```
npm install bootstrap@5.3.7
```

Die Integration erfolgte in `styles.css`:

```
@import 'bootstrap/dist/css/bootstrap.min.css';
```

13.2.2. ng2-pdf-viewer 10.4.0

Für die PDF-Vorschau wurde `ng2-pdf-viewer` gewählt. Bei der Suche nach einer einfachen Lösung zur Anzeige von PDFs im Browser erwies sich diese Bibliothek als geeignet, da sie direkt für Angular entwickelt wurde und eine unkomplizierte Integration ermöglicht.[9]

[8] Vgl. Bootstrap Team: *Bootstrap 5 Documentation*, https://getbootstrap.com/docs/5.3/, abgerufen am 15.01.2025

[9] Vgl. Mozilla Foundation: *PDF.js Documentation*, https://mozilla.github.io/pdf.js/, abgerufen am 15.01.2025

```
npm install ng2-pdf-viewer@10.4.0
```

Template-Integration:

```
<pdf-viewer [src]="pdfData" [render-text]="true"></pdf-viewer>
```

13.3. Projektstruktur

Das *SmartBillConverter*-Frontend folgt der modularen Angular-Architektur mit Komponenten, Services und Routing:

Listing 6: Projektstruktur smart-bill-ui

```
src/app/
  components/
    upload/
      upload.component.ts        # Datei-Upload-Logik
      upload.component.html      # Upload-UI mit Drag & Drop
    invoice-list/
      invoice-list.component.ts # Rechnungsuebersicht
    processing/
      processing.component.ts    # Verarbeitungsstatus
  services/
    invoice.service.ts           # HTTP-Kommunikation
  models/
    invoice.model.ts             # TypeScript-Interfaces
  app.routes.ts                  # Routing-Konfiguration
  app.config.ts                  # Provider-Setup
```

13.3.1. Komponenten-Architektur

Die *SmartBillConverter*-Anwendung besteht aus drei Hauptkomponenten:

- **UploadComponent**: Datei-Upload via Drag & Drop oder File-Input. Unterstützt Multi-File-Upload und zeigt Fortschrittsbalken. Validiert Dateitypen (PDF, PNG, JPEG) und Größenlimits.
- **InvoiceListComponent**: Zeigt Liste der hochgeladenen Rechnungen mit Status (pending, processing, completed, error). Erlaubt Download von ebInterface/ZUGFeRD-XML.
- **ProcessingComponent**: Zeigt Verarbeitungsstatus und Fehler. Stellt OCR-Ergebnisse und extrahierte Daten dar.

13.3.2. Services und Dependency Injection

Der `InvoiceService` verwaltet die HTTP-Kommunikation mit dem Backend:

Listing 7: Invoice Service

```
@Injectable({ providedIn: 'root' })
export class InvoiceService {
  private apiUrl = 'http://localhost:5000/api';

  constructor(private http: HttpClient) {}

  uploadInvoice(file: File): Observable<UploadResponse> {
    const formData = new FormData();
    formData.append('file', file);
    return this.http.post<UploadResponse>(
      `${this.apiUrl}/upload`, formData
    );
  }

  getInvoices(): Observable<Invoice[]> {
    return this.http.get<Invoice[]>(`${this.apiUrl}/invoices`);
  }
}
```

Der Service funktioniert folgendermaßen:

- **@Injectable**: Macht den Service für andere Komponenten verfügbar. `providedIn: 'root'` bedeutet, dass es nur eine Instanz im gesamten Projekt gibt.[10]
- **apiUrl**: Speichert die Backend-Adresse (`http://localhost:5000/api`). Alle Anfragen gehen an diese URL.
- **constructor**: Bekommt den `HttpClient` automatisch von Angular bereitgestellt. Damit können HTTP-Anfragen (GET, POST) gesendet werden.
- **uploadInvoice**: Erstellt ein `FormData`-Objekt (benötigt für Datei-Uploads), hängt die Datei an und sendet sie per POST an `/api/upload`. Gibt ein `Observable` zurück, das später die Antwort vom Server liefert.
- **getInvoices**: Holt alle Rechnungen vom Backend per GET-Anfrage an `/api/invoices`. Gibt ein `Observable` zurück, das ein Array von Rechnungen enthält.

Das `Observable`-Pattern ermöglicht asynchrone Datenverarbeitung: Die Komponente abonniert das Observable mit `.subscribe()` und erhält die Daten, sobald die Server-Antwort eingetroffen ist.

13.4. Routing-Konfiguration

Das Routing steuert die Navigation zwischen den verschiedenen Seiten der Anwendung. Es definiert, welche Komponente bei welcher URL angezeigt wird:

Listing 8: Routing in app.routes.ts

[10] Vgl. Angular Documentation: *Injectable Decorator*, https://angular.io/api/core/Injectable, abgerufen am 15.01.2025

```
export const routes: Routes = [
  { path: '', redirectTo: '/upload', pathMatch: 'full' },
  { path: 'upload', component: UploadComponent },
  { path: 'invoices', component: InvoiceListComponent },
  { path: 'processing/:id', component: ProcessingComponent },
  { path: '**', redirectTo: '/upload' }
];
```

Die Routen erlauben:

- `/upload`: Zeigt die Upload-Seite an (Standardroute). Wenn jemand auf die Startseite geht (`/`), wird automatisch auf `/upload` weitergeleitet.
- `/invoices`: Zeigt die Rechnungsübersicht mit allen hochgeladenen Rechnungen.
- `/processing/:id`: Zeigt den Verarbeitungsstatus einer bestimmten Rechnung. Die `:id` ist ein Platzhalter, z.B. `/processing/123` zeigt den Status von Rechnung 123.
- `**`: Wildcard-Route, die bei ungültigen URLs greift. Leitet zurück auf `/upload`, falls jemand eine nicht existierende Seite aufruft.

13.4.1. Navigation

Die Navigation kann auf zwei Arten erfolgen:

Programmatisch im TypeScript-Code:

```
constructor(private router: Router) {}

navigateToProcessing(invoiceId: number): void {
  this.router.navigate(['/processing', invoiceId]);
}
```

Hier wird der `Router`-Service verwendet, um per Code zu einer anderen Seite zu wechseln.[11] Die Methode `navigate()` bekommt ein Array mit dem Pfad und den Parametern (z.B. `['/processing', 123]` wird zu `/processing/123`).

Im HTML-Template mit routerLink:

```
<a routerLink="/invoices" routerLinkActive="active">
  Rechnungen
</a>
```

Die `routerLink`-Direktive erstellt einen anklickbaren Link. `routerLinkActive="active"` fügt automatisch die CSS-Klasse `active` hinzu, wenn der Benutzer auf dieser Seite ist (nützlich für Navigation-Highlighting).

[11] Vgl. Angular Documentation: *Router*, https://angular.io/api/router/Router, abgerufen am 15.01.2025

Teil V.

Implementierung (Frontend & Backend-Refinement)

14. Implementierung der Upload-Komponente und Multi-File-System

Die Upload-Komponente ist das Hauptelement der im Rahmen dieses Projekts entwickelten SmartBillConverter-Anwendung. Sie steuert den gesamten Prozess vom Hochladen der Dateien bis zur Erstellung der XML-Dateien.

14.1. Entwicklung der Upload-Komponente mit Drag-and-Drop

Die Upload-Komponente des *SmartBillConverter*-Projekts wurde als eigenständige Angular-Komponente entwickelt. Sie nutzt die neue `@for`- und `@if`-Syntax von Angular 19 und `@ViewChild`, um auf HTML-Elemente im Template zuzugreifen.

14.1.1. Aufbau der Komponente

Die Komponente wird mit dem `@Component`-Decorator konfiguriert:

Listing 9: Upload-Component Decorator-Konfiguration

```
@Component({
  selector: 'app-upload',
  standalone: true,
  imports: [CommonModule, PdfViewerModule, HttpClientModule],
  providers: [InvoiceService],
  templateUrl: './upload.component.html',
  styleUrl: './upload.component.css'
})
export class UploadComponent {
  @ViewChild('fileInput') fileInput!: ElementRef<HTMLInputElement>;
  selectedFiles: FileUploadItem[] = [];
  selectedFormat: 'ebInterface' | 'ZUGFeRD' | null = null;
  isDragOver = false;
  isProcessing = false;

  constructor(private invoiceService: InvoiceService) {}
}
```

Der Parameter `standalone: true` bedeutet, dass die Komponente unabhängig funktioniert und nicht in ein zusätzliches Modul eingebunden werden muss.[12] Das macht den Code übersichtlicher.

14.1.2. Drag-and-Drop-Funktion

Für das Hochladen per Drag-and-Drop werden drei Event-Handler verwendet:

Listing 10: Drag-and-Drop Template

```
<div class="upload-area"
  [class.drag-over]="isDragOver"
  (dragover)="onDragOver($event)"
  (drop)="onDrop($event)"
  (click)="fileInput.click()">
  <i class="bi bi-cloud-upload display-1"></i>
  <h4>Dateien hier ablegen oder klicken zum Auswaehlen</h4>
  <p>PDF, PNG, JPG, BMP, TIFF, GIF - Max. 10MB pro Datei</p>
</div>
```

Die Event-Handler verhindern das Standard-Browserverhalten (Datei öffnen) und verarbeiten stattdessen die Dateien in der Anwendung:

Listing 11: Event-Handler-Implementierung

```
onDragOver(event: DragEvent): void {
  event.preventDefault();
  this.isDragOver = true;
}

onDrop(event: DragEvent): void {
  event.preventDefault();
  this.isDragOver = false;
  if (event.dataTransfer?.files) {
    const fileArray = Array.from(event.dataTransfer.files);
    this.handleMultipleFiles(fileArray);
  }
}
```

14.1.3. Dateien prüfen und Vorschau erstellen

Die Methode `handleMultipleFiles` prüft, ob die Dateien unterstützte Formate haben, und erstellt Vorschauen:

Listing 12: Multi-File-Handling mit Validierung

```
private handleMultipleFiles(files: File[]): void {
  files.forEach(file => {
    const supportedTypes = ['application/pdf', 'image/png',
                            'image/jpeg', 'image/bmp'];
    if (!supportedTypes.includes(file.type)) {
      this.showFileTypeError(file.name);
      return;
    }

    const fileItem: FileUploadItem = {
```

[12] Vgl. Angular Documentation: *Standalone Components*, https://angular.io/guide/standalone-components, abgerufen am 15.01.2025

```
        file: file,
        id: `file_${Date.now()}_${Math.random()}`,
        status: 'pending'
      };

      this.loadFilePreview(fileItem);
      this.selectedFiles.push(fileItem);
    });
  }
```

Für die Vorschau wird zwischen Bildern und PDFs unterschieden. Bilder werden als Data-URL geladen, PDFs als Binärdaten:

Listing 13: FileReader API für Vorschauen

```
private loadFilePreview(fileItem: FileUploadItem): void {
  const reader = new FileReader();

  if (fileItem.file.type.startsWith('image/')) {
    reader.onload = (e) => fileItem.previewUrl = e.target?.result as string;
    reader.readAsDataURL(fileItem.file);
  } else if (fileItem.file.type === 'application/pdf') {
    reader.onload = (e) => {
      fileItem.pdfData = new Uint8Array(e.target?.result as ArrayBuffer);
    };
    reader.readAsArrayBuffer(fileItem.file);
  }
}
```

Die HTML5 FileReader API bietet zwei Methoden: `readAsDataURL()` für Bilder (als Base64-String) und `readAsArrayBuffer()` für PDFs (als Binärdaten für den PDF-Viewer).[13]

14.2. PDF-Vorschau anzeigen

Die Anzeige der hochgeladenen PDFs erfolgt direkt im Browser. Dafür wird die Bibliothek `ng2-pdf-viewer` verwendet, die eine komfortable Vorschau mit Zoom- und Navigationsfunktionen bietet.

14.2.1. Einbindung des PDF-Viewers

Listing 14: ng2-pdf-viewer Installation

```
npm install ng2-pdf-viewer@10.4.0
```

Die PDF-Vorschau wird in einem Bootstrap-Modal (Popup-Fenster) angezeigt:

Listing 15: PDF-Viewer im Modal-Dialog

```
@if (showPreview && pdfData) {
  <div class="modal fade show d-block">
```

[13] Vgl. MDN Web Docs: *FileReader API*, https://developer.mozilla.org/en-US/docs/Web/API/FileReader, abgerufen am 15.01.2025

```
      <div class="modal-dialog modal-xl">
        <div class="modal-content">
          <div class="modal-header">
            <h5>PDF Vorschau: {{ getPreviewFileName() }}</h5>
            <button class="btn-close" (click)="togglePreview()"></button>
          </div>
          <div class="modal-body">
            <pdf-viewer
              [src]="pdfData"
              [render-text]="true"
              [show-all]="true"
              [fit-to-page]="true"
              style="width: 100%; height: 70vh;">
            </pdf-viewer>
          </div>
        </div>
      </div>
    </div>
  }
```

Die Optionen bedeuten: `[render-text]` aktiviert Text-Auswahl und Suche, `[show-all]` zeigt alle Seiten gleichzeitig, `[fit-to-page]` passt die Größe an die Fensterbreite an. Das PDF wird als `Uint8Array` (Binärdaten) über `[src]` übergeben.

14.2.2. Speicher freigeben

Beim Schließen der Vorschau werden Blob-URLs freigegeben, um Speicherprobleme zu vermeiden:

Listing 16: Vorschau-Steuerung mit Cleanup

```
showFilePreview(fileId: string): void {
  const fileItem = this.selectedFiles.find(f => f.id === fileId);
  if (fileItem) {
    this.pdfData = fileItem.pdfData;
    this.previewFileId = fileId;
    this.showPreview = true;
  }
}

togglePreview(): void {
  this.showPreview = false;
  if (this.previewUrl) {
    URL.revokeObjectURL(this.previewUrl);
  }
}
```

Die Methode `URL.revokeObjectURL()` gibt Speicher frei, der von `URL.createObjectURL()` belegt wurde.[14] Das ist wichtig, weil Browser diese URLs sonst bis zum Schließen der Seite speichern.

14.3. Datenstruktur für hochgeladene Dateien

Das `FileUploadItem`-Interface definiert, welche Informationen für jede hochgeladene Datei gespeichert werden:

[14] Vgl. MDN Web Docs: *URL.createObjectURL()*, https://developer.mozilla.org/en-US/docs/Web/API/URL/createObjectURL, abgerufen am 15.01.2025

Listing 17: FileUploadItem Interface-Definition

```
interface FileUploadItem {
  file: File;                // Original File-Objekt
  id: string;                // Eindeutiger Identifier
  status: 'pending' | 'processing' | 'completed' | 'error'; // Status
  progress?: number;         // Upload-Fortschritt (0-100)
  result?: any;              // Server-Response
  errorMessage?: string;     // Fehlermeldung bei Fehler
  previewUrl?: string;       // Data-URL f r Bildvorschau
  pdfData?: Uint8Array;      // Bin rdaten f r PDF-Vorschau
}
```

Der Status kann vier Werte haben: `pending` (wartet auf Verarbeitung), `processing` (wird gerade verarbeitet), `completed` (erfolgreich abgeschlossen) oder `error` (Fehler aufgetreten). TypeScript prüft automatisch, dass nur diese Werte verwendet werden.

Anstelle eines Enums wurde ein Union-Type verwendet.[15] Das hat folgende Vorteile: Einfacherer Vergleich mit Strings, kleinere Dateigröße und einfachere Umwandlung in JSON.

14.3.1. Dateien verwalten

Die Komponente verwaltet alle hochgeladenen Dateien in einem Array:

Listing 18: File-Array-Management-Methoden

```
// Hinzuf gen neuer Dateien
this.selectedFiles.push(fileItem);

// Status-Update w hrend Verarbeitung
const currentFile = this.selectedFiles[this.currentProcessingIndex];
currentFile.status = 'processing';

// Entfernen einzelner Dateien
removeSingleFile(fileId: string): void {
  this.selectedFiles = this.selectedFiles.filter(f => f.id !== fileId);
}

// Alle Dateien l schen
clearAllFileData(): void {
  this.selectedFiles = [];
  this.allResults = [];
  this.processedCount = 0;
}
```

Die `filter()`-Methode erstellt ein neues Array ohne die zu entfernende Datei. Das ist wichtig für Angular's Change Detection (die automatische UI-Aktualisierung), die nur auf Änderungen der Array-Referenz reagiert. Eine direkte Änderung mit `splice()` würde die Referenz nicht ändern und die UI würde möglicherweise nicht aktualisiert werden.

Im HTML-Template werden die Dateien mit `@for` angezeigt:

[15] Vgl. TypeScript Documentation: *Union Types*, https://www.typescriptlang.org/docs/handbook/2/everyday-types.html#union-types, abgerufen am 15.01.2025

Listing 19: Template mit FileUploadItem-Array

```
@for (fileItem of selectedFiles; track fileItem.id) {
  <div class="list-group-item">
    <h6>{{ fileItem.file.name }}</h6>
    <div class="progress">
      <div class="progress-bar"
           [class.bg-success]="fileItem.status === 'completed'"
           [class.bg-danger]="fileItem.status === 'error'"
           [style.width.%]="getProgressPercentage(fileItem)">
      </div>
    </div>
  </div>
}
```

Die `track`-Funktion verbessert die Performance: Angular aktualisiert nur geänderte Elemente, anstatt die gesamte Liste neu zu erstellen.

14.4. Dateien nacheinander verarbeiten

Die Anwendung verarbeitet mehrere Dateien nacheinander (sequenziell). Dafür werden Variablen wie `isProcessingAll`, `currentProcessingIndex` und `processedCount` verwendet.

14.4.1. Start und Verarbeitung

Die Methode `uploadAllFiles` startet die Verarbeitung aller ausgewählten Dateien:

Listing 20: Upload-Initialisierung

```
private uploadAllFiles(): void {
  this.isProcessingAll = true;
  this.currentProcessingIndex = 0;
  this.processedCount = 0;
  this.totalFiles = this.selectedFiles.length;
  this.allResults = [];

  this.selectedFiles.forEach(f => f.status = 'pending');
  this.processNextFile();
}
```

Diese Methode setzt alle Zähler und Status zurück und startet dann die Verarbeitung der ersten Datei mit `processNextFile()`.

Die Methode `processNextFile` verarbeitet die Dateien nacheinander:

Listing 21: Rekursive Dateiverarbeitung

```
private processNextFile(): void {
  if (this.currentProcessingIndex >= this.selectedFiles.length) {
    this.isProcessingAll = false;
    return;
  }

  const currentFile = this.selectedFiles[this.currentProcessingIndex];
```

```
    currentFile.status = 'processing';

    this.invoiceService.uploadInvoice(currentFile.file).subscribe({
      next: (response) => {
        currentFile.status = 'completed';
        currentFile.result = response;
        this.allResults.push(response);
        this.processedCount++;
        this.currentProcessingIndex++;
        this.processNextFile();  // Rekursiver Aufruf
      },
      error: (error) => {
        currentFile.status = 'error';
        currentFile.errorMessage = error.error?.error || 'Fehler';
        this.currentProcessingIndex++;
        this.processNextFile();  // Weiter trotz Fehler
      }
    });
  }
```

Die Methode holt sich die aktuelle Datei aus dem Array, sendet sie ans Backend und wartet auf die Antwort. Bei Erfolg wird der Status auf `completed` gesetzt, bei Fehler auf `error`. In beiden Fällen wird der Index erhöht und die Methode ruft sich selbst auf, um die nächste Datei zu verarbeiten. Diese rekursive Logik (Methode ruft sich selbst auf) stellt sicher, dass die Dateien nacheinander und nicht gleichzeitig verarbeitet werden.

14.4.2. Fortschritt anzeigen und alle Downloads starten

Der Fortschritt wird basierend auf dem Status berechnet:

Listing 22: Progress-Berechnung

```
getProgressPercentage(fileItem: FileUploadItem): number {
  switch (fileItem.status) {
    case 'pending': return 0;
    case 'processing': return 50;
    case 'completed': return 100;
    case 'error': return 0;
  }
}

getProcessingProgress(): number {
  return this.totalFiles > 0
    ? (this.processedCount / this.totalFiles) * 100
    : 0;
}
```

Für den Download aller XML-Dateien wird `setTimeout` verwendet:

Listing 23: Batch-Download mit Staggering

```
downloadAllXml(): void {
  this.allResults.forEach((result, index) => {
    let xmlContent = result.zugferdXml || result.workflow?.ebInterfaceXml;
    let invoiceNumber = `invoice_${result.invoice?.id || Date.now()}`;

    setTimeout(() => {
      this.invoiceService.generateXmlDownload(
        xmlContent, invoiceNumber, this.selectedFormat!
      );
    }, index * 500);  // 500ms Verzögerung zwischen Downloads
  });
}
```

Die 500-Millisekunden-Verzögerung zwischen den Downloads verhindert, dass der Browser blockiert wird. Browser erlauben normalerweise nur 6 gleichzeitige Downloads pro Website, und die Verzögerung stellt sicher, dass dieses Limit nicht überschritten wird.

Die XML-Download-Funktion erstellt einen Download-Link:

Listing 24: XML-Download mit Blob-API

```
generateXmlDownload(xmlContent: string, invoiceNumber: string,
                    format: 'ebInterface' | 'ZUGFeRD'): void {
  const blob = new Blob([xmlContent], { type: 'application/xml' });
  const url = window.URL.createObjectURL(blob);
  const link = document.createElement('a');
  link.href = url;
  link.download = `${invoiceNumber}_${format}.xml`;

  document.body.appendChild(link);
  link.click();
  document.body.removeChild(link);
  window.URL.revokeObjectURL(url);
}
```

Diese Methode erstellt einen temporären Download-Link, klickt ihn automatisch an und entfernt ihn wieder. Die Datei wird direkt im Browser erstellt, ohne dass der Server nochmal kontaktiert werden muss.

Die Anwendung hat einen zweistufigen Ablauf: Zuerst wird das Format gewählt (ebInterface oder ZUGFeRD), dann erscheint der Upload-Bereich. Bei Formatwechsel werden vorherige Dateien gelöscht.

14.4.3. Gesamtablauf

Der Upload-Workflow hat folgende Schritte:

1. **Format-Auswahl:** Benutzer wählt ebInterface oder ZUGFeRD
2. **Datei-Selektion:** Drag-and-Drop oder Click-to-Upload
3. **Validierung:** MIME-Type-Prüfung und Größenlimit
4. **Vorschau-Generierung:** FileReader API lädt Preview-Daten
5. **Sequenzielle Verarbeitung:** Uploads nacheinander an Backend
6. **Status-Tracking:** Echtzeit-Updates in UI (pending → processing → completed/error)
7. **Ergebnis-Download:** XML-Dateien einzeln oder als Batch

Jeder Schritt hat eigene Fehlerbehandlung und zeigt dem Benutzer Feedback. Die Aufteilung in verschiedene Bereiche (Datei-Handling, Backend-Kommunikation, UI-Anzeige) macht den Code wartbar und erweiterbar.

14.4.4. Kommunikation mit dem Backend

Die Kommunikation mit dem Backend erfolgt über den `InvoiceService`:

Listing 25: InvoiceService Upload-Methoden

```
@Injectable({ providedIn: 'root' })
export class InvoiceService {
  private apiUrl = environment.apiUrl;

  constructor(private http: HttpClient) {}

  uploadInvoice(file: File): Observable<any> {
    const formData = new FormData();
    formData.append('file', file);
    return this.http.post(`${this.apiUrl}/Processing/upload`, formData);
  }

  uploadInvoiceForZugferd(file: File): Observable<any> {
    const formData = new FormData();
    formData.append('file', file);
    return this.http.post(`${this.apiUrl}/zugferd/upload-pdf`, formData, {
      responseType: 'text'
    }).pipe(
      map((xmlResponse: string) => ({ zugferdXml: xmlResponse }))
    );
  }
}
```

Die `FormData`-API ermöglicht das Hochladen von Dateien per HTTP POST.[16] Der `Observable`-Ansatz von Angular's `HttpClient` ermöglicht asynchrone Verarbeitung mit RxJS-Operatoren wie `map`, `catchError` und `retry`.

14.4.5. Fehler behandeln

Die Fehlerbehandlung erfolgt auf mehreren Ebenen:

Listing 26: Mehrstufige Fehlerbehandlung

```
this.invoiceService.uploadInvoice(currentFile.file).subscribe({
  next: (response) => {
    currentFile.status = 'completed';
    currentFile.result = response;
    this.processedCount++;
    this.currentProcessingIndex++;
    this.processNextFile();
  },
  error: (error) => {
    console.error('Upload fehlgeschlagen:', error);
    currentFile.status = 'error';
    currentFile.errorMessage = error.error?.error ||
                               error.message ||
                               'Unbekannter Fehler';
    this.currentProcessingIndex++;
```

[16] Vgl. MDN Web Docs: *FormData*, https://developer.mozilla.org/en-US/docs/Web/API/FormData, abgerufen am 15.01.2025

```
16      this.processNextFile();  // Continue-on-error Pattern
17    }
18  });
```

Das Continue-on-error Pattern bedeutet: Wenn eine Datei fehlschlägt, wird trotzdem mit der nächsten Datei weitergemacht. Die Fehlermeldung wird aus der Server-Antwort ausgelesen oder durch einen Standard-Text ersetzt.

Fehler werden in der Oberfläche mit Bootstrap Alerts angezeigt:

Listing 27: Error-Display im Template

```
1  @if (fileItem.status === 'error' && fileItem.errorMessage) {
2    <div class="alert alert-danger alert-dismissible">
3      <i class="bi bi-exclamation-triangle me-2"></i>
4      {{ fileItem.errorMessage }}
5    </div>
6  }
```

15. UI/UX und Interaktionsdesign

Das Design der Benutzeroberfläche ist wichtig für die Akzeptanz der Anwendung. Dieses Kapitel beschreibt die wichtigsten UI/UX-Entscheidungen im SmartBillConverter-Projekt und zeigt, wie die Benutzeroberfläche entwickelt wurde.

15.1. Implementierung der Format-Auswahl

Eine der ersten Anforderungen war die Auswahl zwischen zwei Rechnungsformaten: ebInterface (österreichischer Standard) und ZUGFeRD (deutscher Standard). Benutzer sollten vor dem Upload entscheiden können, in welches Format ihre Rechnung konvertiert werden soll.

15.1.1. Design der Format-Buttons

Die zwei Formate wurden als große Buttons dargestellt, die nebeneinander angeordnet sind. Jedes Format hat eine eigene Farbe, um die Unterscheidung zu erleichtern. ebInterface ist grün und ZUGFeRD ist rot.

Listing 28: Format-Auswahl-Buttons im HTML-Template

```
<div class="col-md-5">
  <button
    class="btn btn-format ebinterface-btn w-100"
    [class.selected]="selectedFormat === 'ebInterface'"
    (click)="selectFormat('ebInterface')">
    <i class="bi bi-file-earmark-text me-2"></i>
    ebInterface
  </button>
</div>
<div class="col-md-5">
  <button
    class="btn btn-format zugferd-btn w-100"
    [class.selected]="selectedFormat === 'ZUGFeRD'"
    (click)="selectFormat('ZUGFeRD')">
    <i class="bi bi-file-earmark-text me-2"></i>
    ZUGFeRD
  </button>
</div>
```

Die Buttons haben einen Rahmen in der jeweiligen Farbe und einen leicht transparenten Hintergrund. Wenn der Benutzer mit der Maus über einen Button fährt oder ihn auswählt, wird der Hintergrund voll eingefärbt und der Text wird weiß.

15.1.2. Toggle-Funktionalität

Ein besonderes Feature ist, dass Benutzer ihre Auswahl wieder rückgängig machen können. Wenn sie auf den bereits ausgewählten Button klicken, wird die Auswahl aufgehoben und die Upload-Sektion verschwindet wieder. Das ist nützlich, wenn jemand sich umentscheidet oder versehentlich das falsche Format gewählt hat.

Listing 29: Toggle-Logik in der selectFormat-Methode

```
selectFormat(format: 'ebInterface' | 'ZUGFeRD'): void {
  if (this.selectedFormat === format) {
    // Wenn das Format bereits ausgew hlt war, deselektieren
    this.selectedFormat = null;
    this.showUploadSection = false;
    this.clearAllFileData();
  } else {
    // Neues Format ausw hlen
    this.selectedFormat = format;
    this.showUploadSection = true;
  }
}
```

Diese Lösung verbessert die Benutzerfreundlichkeit, weil Benutzer nicht gezwungen sind, ihre Wahl beizubehalten. Sie können jederzeit neu starten.

15.2. Design der Multi-File-UI

Anfangs konnte das System nur eine Datei gleichzeitig verarbeiten. Das wurde später erweitert, weil viele Benutzer mehrere Rechnungen auf einmal konvertieren wollen. Die Herausforderung war, alle Dateien übersichtlich darzustellen.

15.2.1. Kartenlayout für Dateiliste

Die Lösung ist ein Kartenlayout, bei dem jede hochgeladene Datei als eigener Eintrag in einer Liste dargestellt wird. Jeder Eintrag zeigt den Dateinamen, die Dateigröße und ein Icon, das anzeigt, ob es sich um ein PDF oder ein Bild handelt.

Listing 30: Dateiliste mit Karten-Design

```
<div class="card">
  <div class="card-header d-flex justify-content-between">
    <h6 class="mb-0">
      <i class="bi bi-files me-2"></i>
      {{ selectedFiles.length }} Dateien ({{ getTotalFileSizeMB() }} MB)
```

```
6       </h6>
7       <button class="btn btn-outline-secondary btn-sm"
8               (click)="removeFile()">
9         <i class="bi bi-trash me-1"></i>
10        Alle entfernen
11      </button>
12    </div>
13    <div class="card-body p-0">
14      <div class="list-group list-group-flush">
15        @for (fileItem of selectedFiles; track fileItem.id) {
16          <div class="list-group-item">
17            <!-- Datei-Informationen und Aktionen -->
18          </div>
19        }
20      </div>
21    </div>
22  </div>
```

Der Karten-Header zeigt die Gesamtanzahl der Dateien und die Gesamtgröße. Das ist praktisch, weil Benutzer sofort sehen, wie viele Dateien sie hochgeladen haben. Der Button "Alle entfernen" erlaubt es, mit einem Klick alle Dateien zu löschen und neu zu starten.

15.2.2. Aktionen pro Datei

Jede Datei hat drei Buttons: Vorschau anzeigen, XML herunterladen (nur nach erfolgreicher Konvertierung) und Datei entfernen. Der Download-Button wird nur angezeigt, wenn die Konvertierung erfolgreich war.

15.3. Evolution der Statusanzeige

Ein wichtiger Teil der Benutzeroberfläche ist die Anzeige des Verarbeitungsstatus. Benutzer wollen wissen, was gerade mit ihrer Datei passiert.

15.3.1. Vier Status-Stufen

Jede Datei durchläuft vier mögliche Status:

- **Pending** (Wartend): Die Datei wurde hochgeladen, aber noch nicht verarbeitet. Wird grau dargestellt.
- **Processing** (Wird verarbeitet): Die Datei wird gerade vom Backend konvertiert. Wird gelb dargestellt mit animiertem Streifen-Muster.
- **Completed** (Abgeschlossen): Die Konvertierung war erfolgreich. Wird grün dargestellt.
- **Error** (Fehler): Es ist ein Fehler aufgetreten. Wird rot dargestellt.

15.3.2. Progress Bars

Unter jedem Dateinamen befindet sich eine Fortschrittsanzeige (Progress Bar). Diese zeigt visuell den aktuellen Status:

Listing 31: Progress Bar mit Farbcodierung

```
<div class="progress" style="height: 8px;">
  <div class="progress-bar"
       [ngClass]="{
         'bg-secondary': fileItem.status === 'pending',
         'bg-warning progress-bar-striped progress-bar-animated':
           fileItem.status === 'processing',
         'bg-success': fileItem.status === 'completed',
         'bg-danger': fileItem.status === 'error'
       }"
       [style.width.%]="getProgressPercentage(fileItem)">
  </div>
</div>
```

Die Progress Bar ist zu 0% gefüllt bei "Pending", zu 50% bei "Processing" und zu 100% bei "Completed" oder "Error". Der Streifen-Effekt bei "Processing" gibt dem Benutzer ein visuelles Feedback, dass gerade etwas passiert.

15.3.3. Textuelle Status-Anzeige

Zusätzlich zur farbigen Progress Bar gibt es auch eine Textanzeige, die den Status erklärt:

Listing 32: Status-Text mit Icons

```
@if (fileItem.status === 'pending') {
  <small class="text-muted">Wartend...</small>
} @else if (fileItem.status === 'processing') {
  <small class="text-warning">Wird verarbeitet...</small>
} @else if (fileItem.status === 'completed') {
  <small class="text-success">    Erfolgreich konvertiert</small>
} @else if (fileItem.status === 'error') {
  <small class="text-danger">    Fehler aufgetreten</small>
  @if (fileItem.errorMessage) {
    <br><small class="text-danger">{{ fileItem.errorMessage }}</small>
  }
}
```

Bei Fehlern wird zusätzlich die Fehlermeldung vom Backend angezeigt. Das hilft dem Benutzer zu verstehen, was schief gelaufen ist.

15.4. Design und Integration der App-Icons

Icons spielen eine wichtige Rolle in der Benutzeroberfläche. Sie helfen Benutzern, Funktionen schneller zu erkennen und machen die Anwendung visuell ansprechender.

15.4.1. Bootstrap Icons

Für das SmartBillConverter-Projekt wurden Bootstrap Icons verwendet. Diese Icon-Bibliothek ist kostenlos und passt gut zum Bootstrap-Framework, das bereits für das Layout verwendet wird.[17]

Die wichtigsten Icons im Projekt sind:

- `bi-cloud-upload`: Zeigt die Upload-Fläche an
- `bi-file-earmark-pdf`: Kennzeichnet PDF-Dateien
- `bi-file-earmark-image`: Kennzeichnet Bilddateien
- `bi-eye`: Button für Vorschau anzeigen
- `bi-download`: Button für Download
- `bi-x-lg`: Button zum Entfernen
- `bi-trash`: Button zum Löschen aller Dateien
- `bi-files`: Icon für mehrere Dateien

15.4.2. Drag-and-Drop Bereich

Der Upload-Bereich verwendet ein großes Cloud-Upload-Icon, das dem Benutzer signalisiert, dass er Dateien hierher ziehen kann:

Listing 33: Upload-Icon in der Drag-and-Drop-Zone

```
@if (selectedFiles.length === 0) {
  <i class="bi bi-cloud-upload display-1 text-secondary mb-3"></i>
  <h4>Dateien hier ablegen oder klicken zum Ausw hlen</h4>
  <p class="text-muted">
    PDF, PNG, JPG, BMP, TIFF, GIF - Max. 10MB pro Datei
  </p>
}
```

Wenn Dateien ausgewählt wurden, ändert sich das Icon zu einem Datei-Check-Icon oder einem Multi-Datei-Icon, je nachdem ob eine oder mehrere Dateien hochgeladen wurden.

[17] Vgl. Bootstrap Team: *Bootstrap Icons*, https://icons.getbootstrap.com/, abgerufen am 15.01.2025

15.4.3. Konsistente Icon-Verwendung

Alle Buttons verwenden Icons zusammen mit Text. Das macht die Buttons einfacher zu verstehen, besonders für Benutzer, die nicht so gut Deutsch können. Ein Auge-Icon beim Vorschau-Button ist international verständlich.

16. Anbindung der Backend-API (InvoiceService)

Die Kommunikation zwischen Frontend und Backend erfolgt über HTTP-Anfragen. Dieses Kapitel beschreibt den InvoiceService im SmartBillConverter-Projekt, der alle Anfragen an die Backend-API verwaltet.

16.1. Implementierung des Angular InvoiceService

Der InvoiceService ist das zentrale Element für die Backend-Kommunikation. Er kapselt alle HTTP-Anfragen und stellt sie als einfache Methoden für die Komponenten bereit.

16.1.1. Grundstruktur des Service

Ein Angular Service wird mit dem `@Injectable`-Decorator markiert. Das ermöglicht es Angular, den Service automatisch in andere Komponenten zu injizieren.[18]

Listing 34: Grundstruktur des InvoiceService

```
1  @Injectable({
2    providedIn: 'root'
3  })
4  export class InvoiceService {
5    private apiUrl = environment.apiUrl;
6
7    constructor(private http: HttpClient) { }
8  }
```

Die `apiUrl` wird aus der Environment-Konfiguration geladen. Das ist praktisch, weil man die URL für Entwicklung und Produktion unterschiedlich setzen kann. Für Entwicklung ist es `http://localhost:5000/api`, für Produktion würde man die echte Server-URL eintragen.

[18] Vgl. Angular Documentation: *Dependency Injection in Angular*, https://angular.io/guide/dependency-injection, abgerufen am 15.01.2025

16.1.2. Intelligente Upload-Methode

Die Hauptmethode `uploadInvoice` entscheidet automatisch, welchen Backend-Endpunkt sie verwenden soll. PDFs und Bilder werden unterschiedlich verarbeitet:

Listing 35: Automatische Routing-Logik basierend auf Dateityp

```
uploadInvoice(file: File): Observable<any> {
  const formData = new FormData();
  formData.append('file', file);

  if (file.type === 'application/pdf') {
    return this.http.post(
      `${this.apiUrl}/Processing/upload`,
      formData
    );
  } else if (this.isImageFile(file)) {
    return this.uploadImage(file);
  } else {
    throw new Error(`Unsupported file type: ${file.type}`);
  }
}

private isImageFile(file: File): boolean {
  const imageTypes = ['image/png', 'image/jpeg', 'image/jpg',
                      'image/bmp', 'image/tiff', 'image/gif'];
  return imageTypes.includes(file.type);
}
```

Diese Lösung vereinfacht die Verwendung in den Komponenten. Die Upload-Komponente muss sich nicht darum kümmern, ob es sich um ein PDF oder ein Bild handelt. Der Service übernimmt diese Entscheidung.

16.1.3. Observable-Pattern

Alle Service-Methoden geben ein `Observable` zurück. Das ist ein RxJS-Konzept für asynchrone Datenverarbeitung.[19] Die Komponente kann das Observable mit `.subscribe()` abonnieren und bekommt die Daten, wenn die Server-Antwort eingetroffen ist.

16.2. Aktivierung der ZUGFeRD-Funktionalität im Frontend

ZUGFeRD ist das deutsche Pendant zu ebInterface. Die Implementierung der ZUGFeRD-Funktionalität war eine Erweiterung des bestehenden Systems.

[19] Vgl. Angular Documentation: *Observables in Angular*, https://angular.io/guide/observables, abgerufen am 15.01.2025

16.2.1. Separate Upload-Methode für ZUGFeRD

Für ZUGFeRD gibt es eine eigene Upload-Methode, weil das Backend einen anderen Endpunkt verwendet:

Listing 36: ZUGFeRD-spezifische Upload-Methode

```
uploadInvoiceForZugferd(file: File): Observable<any> {
  const formData = new FormData();
  formData.append('file', file);

  if (this.isImageFile(file)) {
    return this.http.post(
      `${this.apiUrl}/zugferd/upload-image`,
      formData,
      { responseType: 'text' }
    );
  } else {
    return this.http.post(
      `${this.apiUrl}/zugferd/upload-pdf`,
      formData,
      { responseType: 'text' }
    );
  }
}
```

Der Unterschied ist der Endpunkt (`/zugferd/upload-pdf` statt `/Processing/upload`) und der Response-Typ. ZUGFeRD gibt direkt das XML als Text zurück, während ebInterface ein JSON-Objekt mit mehreren Feldern zurückgibt.

16.2.2. Response-Transformation

Das XML-Response muss in ein einheitliches Format umgewandelt werden, damit die Upload-Komponente beide Formate gleich behandeln kann:

Listing 37: Transformation des ZUGFeRD-Response

```
return this.http.post(`${this.apiUrl}/zugferd/upload-pdf`,
                      formData, { responseType: 'text' })
  .pipe(
    map((xmlResponse: string) => {
      return {
        zugferdXml: xmlResponse
      };
    })
  );
```

Der `map`-Operator von RxJS wandelt die einfache Text-Antwort in ein Objekt um.[20] Dadurch kann die Upload-Komponente einheitlich mit `result.zugferdXml` oder `result.workflow.eb` auf das XML zugreifen.

[20]Vgl. ReactiveX: *RxJS Operators Documentation*, https://rxjs.dev/guide/operators, abgerufen am 15.01.2025

16.3. Implementierung des HTTP-Event-Tracking

HTTP-Event-Tracking ermöglicht es, den Fortschritt eines Uploads zu verfolgen. Im SmartBillConverter-Projekt wurde dies jedoch nicht vollständig implementiert, weil die meisten Dateien klein sind und sehr schnell hochgeladen werden.

16.3.1. Aktueller Ansatz

Statt echtem Upload-Fortschritt verwendet das Projekt Status-Tracking auf Komponenten-Ebene. Die Upload-Komponente setzt den Status manuell auf "Processing" und "Completed":

Listing 38: Status-Tracking in der Upload-Komponente

```
this.invoiceService.uploadInvoice(file).subscribe({
  next: (response) => {
    fileItem.status = 'completed';
    fileItem.result = response;
  },
  error: (error) => {
    fileItem.status = 'error';
    fileItem.errorMessage = error.error?.error || 'Unbekannter Fehler';
  }
});
```

Diese Lösung ist einfacher und ausreichend für die meisten Anwendungsfälle. Bei größeren Dateien könnte man in Zukunft echtes HTTP-Event-Tracking hinzufügen.

16.4. XML-Download-Funktionalität

Nach erfolgreicher Konvertierung wollen Benutzer die generierte XML-Datei herunterladen. Der InvoiceService bietet dafür eine praktische Methode.

16.4.1. Download mit Blob URLs

Der Download funktioniert über Blob URLs. Ein Blob ist ein temporäres Objekt, das Dateien im Browser repräsentiert:[21]

Listing 39: XML-Download-Methode

```
generateXmlDownload(xmlContent: string, invoiceNumber: string,
                    format: 'ebInterface' | 'ZUGFeRD'): void {
  const blob = new Blob([xmlContent], { type: 'application/xml' });
  const url = window.URL.createObjectURL(blob);

  const a = document.createElement('a');
```

[21] Vgl. MDN Web Docs: *Blob*, https://developer.mozilla.org/en-US/docs/Web/API/Blob, abgerufen am 15.01.2025

```
7     a.href = url;
8     a.download = `${format.toLowerCase()}_${invoiceNumber}.xml`;
9     document.body.appendChild(a);
10    a.click();
11    document.body.removeChild(a);
12    window.URL.revokeObjectURL(url);
13  }
```

Diese Methode erstellt ein unsichtbares Link-Element, setzt den Download-Namen und triggert automatisch den Download. Danach werden das Link-Element und die Blob URL wieder aufgeräumt.

16.4.2. Verwendung in der Upload-Komponente

Die Upload-Komponente ruft die Download-Methode auf, wenn der Benutzer auf den Download-Button klickt. Das XML wird aus dem Backend-Response extrahiert und der Dateiname wird aus der Rechnungsnummer und dem Format zusammengesetzt.

16.4.3. Batch-Download für mehrere Dateien

Für den Fall, dass mehrere Dateien gleichzeitig konvertiert wurden, gibt es auch eine Batch-Download-Funktion:

Listing 40: Download aller XML-Dateien

```
1   downloadAllXml(): void {
2     this.allResults.forEach((result, index) => {
3       setTimeout(() => {
4         this.invoiceService.generateXmlDownload(
5           xmlContent,
6           invoiceNumber,
7           this.selectedFormat!
8         );
9       }, index * 500);
10    });
11  }
```

Zwischen den Downloads wird eine Pause von 500 Millisekunden eingelegt. Das verhindert, dass der Browser mit zu vielen gleichzeitigen Downloads überfordert wird.

17. Backend-Refinement: Implementierung der Robustheitslogik

Nach ersten Tests mit realen Rechnungen zeigte sich, dass das Backend bei bestimmten Rechnungsformaten Probleme hatte. Dieses Kapitel beschreibt die Verbesserungen, die im SmartBillConverter-Backend implementiert wurden, um diese Probleme zu beheben.

17.1. Problem: Fehlende Versandkosten-Erkennung

Bei der Verarbeitung von Rechnungen mit Versandkosten stellte sich heraus, dass diese oft nicht korrekt erkannt wurden. Das führte zu falschen Gesamtbeträgen im generierten XML.

17.1.1. Problem-Analyse

Das Backend verwendete die KI (Gemini), um Rechnungsdaten zu extrahieren. Die KI erkannte zwar Artikel und Preise, aber Versandkosten wurden oft übersehen. Typische Fälle:

- Versandkosten als separate Zeile: "Versandkosten gesamt: 13,90 EUR"
- Versandkosten in der Beschreibung: "DHL EUROPACK 20% mit Betrag 25,00 EUR"
- Differenz zwischen Material-Summe und Gesamtpreis war die Versandkosten

Das Problem war, dass die KI nach einem einzigen Keyword "Versandkosten" suchte, aber viele Rechnungen verwenden andere Begriffe wie "Porto", "Paketgebühren" oder "Versandart".

17.2. Multi-Keyword-Versandkosten-Erkennung

Die Lösung war, der KI eine Liste von Begriffen zu geben, nach denen sie suchen soll.

17.2.1. Erweiterter KI-Prompt

Der Prompt für die KI wurde angepasst:

Listing 41: Versandkosten-Anweisungen im KI-Prompt

```
- VERSANDKOSTEN: Suche nach Versand, Versandkosten, Paketgeb hren,
  Porto etc. und extrahiere diese als shippingCost
- Wenn Material-Summe und Gesamtpreis unterschiedlich sind,
  ist die Differenz oft die Versandkosten
- BEISPIEL: Material: 151,10 + Versandkosten: 13,90 = Gesamt: 165,00
  -> shippingCost=13.90
- In priceAnalysis IMMER erw hnen ob Versandkosten gefunden wurden
```

Diese Anweisungen helfen der KI, verschiedene Schreibweisen zu erkennen. Wichtig ist auch der Hinweis auf die Differenzberechnung, wenn keine explizite Versandkostenzeile existiert.

17.2.2. Integration in die Datenstruktur

Im C#-Backend wurde ein neues Feld hinzugefügt:

Listing 42: ShippingCost-Property im Datenmodell

```
public class InvoiceData
{
    public string InvoiceNumber { get; set; }
    public decimal InvoiceAmount { get; set; }
    public decimal? ShippingCost { get; set; }  // NEU
    public string PriceAnalysis { get; set; }
    // ... weitere Felder
}
```

Das Fragezeichen bei `decimal?` bedeutet, dass der Wert `null` sein kann. Das ist wichtig, weil nicht alle Rechnungen Versandkosten haben.

17.3. Entwicklung eines universellen Fallback-Systems

Trotz verbesserter Versandkosten-Erkennung gab es Fälle, wo der berechnete Gesamtbetrag nicht mit dem tatsächlichen Rechnungsbetrag übereinstimmte. Ein universelles Fallback-System sollte diese Fälle abfangen.

17.3.1. Fallback-Logik

Die Idee ist einfach: Wenn die KI einen Gesamtbetrag erkannt hat, der höher ist als die Summe aller Artikel plus Versandkosten, dann verwende den von der KI erkannten Betrag. Das deutet darauf hin, dass die KI zusätzliche Kosten gefunden hat, die nicht in den Artikeln enthalten sind.

Listing 43: Universelles Fallback-System

```
// Berechne Summe aus allen Artikeln
decimal totalBrutto = lineItems.Sum(item =>
    RoundAmount(item.Quantity * item.UnitPrice));

// Versandkosten hinzuf gen , falls vorhanden
if (data.ShippingCost.HasValue && data.ShippingCost.Value > 0)
{
    totalBrutto += RoundAmount(data.ShippingCost.Value);
}

// UNIVERSELLER FALLBACK
if (data.InvoiceAmount > totalBrutto)
{
    _logger.LogInformation($"FALLBACK: InvoiceAmount ({data.InvoiceAmount}) >
        berechneter Betrag ({totalBrutto}) -> verwende InvoiceAmount");
    totalBrutto = data.InvoiceAmount;
}
```

Diese Logik stellt sicher, dass der Gesamtbetrag im XML nie niedriger ist als der tatsächliche Rechnungsbetrag. Das ist besonders wichtig für die rechtliche Korrektheit der generierten XML-Dateien.

17.3.2. Logging für Debugging

Für jeden Schritt gibt es Log-Ausgaben, die bei Problemen helfen:

Listing 44: Logging-Statements für Nachvollziehbarkeit

```
_logger.LogInformation($"Versandkosten hinzugef gt: {shippingCost} EUR");
_logger.LogWarning($"Keine Versandkosten gefunden! ShippingCost =
    {data.ShippingCost}");
_logger.LogInformation($"FALLBACK ANGEWENDET: Neuer totalBrutto = {totalBrutto}");
```

Diese Logs erscheinen in der Konsole während der Verarbeitung und helfen zu verstehen, welche Entscheidungen das System getroffen hat.

17.4. Feature-Parität

Die Verbesserungen mussten sowohl für ebInterface als auch für ZUGFeRD funktionieren. Das SmartBillConverter-System unterstützt beide Formate.

17.4.1. Anwendung auf ebInterface

Für ebInterface wurde die Fallback-Logik im `LlmConversionService` implementiert. Dieser Service verarbeitet die KI-Antworten und erstellt das ebInterface-XML.

Die wichtigsten Änderungen:

- `ShippingCost`-Feld in der `InvoiceData`-Klasse
- Versandkosten werden zum Gesamtbetrag addiert
- Fallback-Check vor XML-Generierung
- Log-Ausgaben für Transparenz

17.4.2. Übertragung auf ZUGFeRD

Für ZUGFeRD wurde die gleiche Logik im `ZugferdService` implementiert. Da beide Services die gleiche `InvoiceData`-Klasse verwenden, mussten nur die XML-Generierungs-Methoden angepasst werden.

Das stellt sicher, dass beide Formate die gleiche Qualität haben. Egal ob ein Benutzer ebInterface oder ZUGFeRD wählt, die Versandkosten werden korrekt erkannt und der Fallback greift bei Bedarf.

17.4.3. Einheitliche Fehlerbehandlung

Beide Services verwenden die gleichen Logging-Mechanismen. Das macht es einfacher, Probleme zu diagnostizieren, weil die Log-Ausgaben für beide Formate identisch aussehen.

Teil VI.

Optimierung und Ergebnisse (Frontend)

18. Performance und Optimierung (Frontend)

Während der Entwicklung traten einige Probleme auf, die die Benutzerfreundlichkeit beeinträchtigten. Dieses Kapitel beschreibt die wichtigsten Fehler und deren Behebung im SmartBillConverter-Projekt.

18.1. Behebung des Upload-Bugs

Der erste schwerwiegende Bug trat auf, als Benutzer eine hochgeladene Datei entfernen und dann direkt eine neue Datei hochladen wollten. Das System weigerte sich, die neue Datei zu akzeptieren.

18.1.1. Problem-Analyse

Der Fehler lag im HTML-File-Input-Element. Wenn eine Datei ausgewählt wurde und dann der "Entfernen"-Button geklickt wurde, löschte die Anwendung zwar alle internen Variablen (`selectedFile`, `previewUrl`, etc.), aber das File-Input-Element selbst behielt seinen Wert. Wenn der Benutzer dann die gleiche Datei erneut auswählte, feuerte das `change`-Event nicht, weil sich der Wert technisch gesehen nicht geändert hatte.

18.1.2. Lösung

Die Lösung war, das File-Input-Element beim Entfernen einer Datei manuell zurückzusetzen:

Listing 45: Vollständiger Reset beim Datei-Entfernen

```
1  private clearAllFileData(): void {
2    // File Input zur cksetzen
3    if (this.fileInput?.nativeElement) {
4      this.fileInput.nativeElement.value = '';
```

```
5     }
6
7     // Multi-file data l schen
8     this.selectedFiles = [];
9     this.currentProcessingIndex = -1;
10    this.allResults = [];
11
12    // Alle anderen Variablen zur cksetzen
13    this.selectedFile = null;
14    this.previewUrl = null;
15    this.showPreview = false;
16    this.pdfData = null;
17    this.isProcessing = false;
18    this.isProcessingAll = false;
19    this.processedCount = 0;
20    this.totalFiles = 0;
21
22    this.hideError();
23  }
```

Der wichtige Teil ist `this.fileInput.nativeElement.value = ''`. Das setzt den internen Wert des HTML-Elements zurück. Dadurch funktioniert das erneute Hochladen der gleichen Datei problemlos.

18.1.3. ViewChild für Element-Zugriff

Um auf das File-Input-Element zugreifen zu können, wurde `@ViewChild` verwendet:[22]

Listing 46: ViewChild-Deklaration

```
1  @ViewChild('fileInput') fileInput!: ElementRef<HTMLInputElement>;
```

Das entsprechende HTML-Element hat eine Template-Referenz:

Listing 47: Template-Referenz im Input

```
1  <input
2    #fileInput
3    type="file"
4    class="d-none"
5    accept=".pdf,.png,.jpg,.jpeg,.bmp,.tiff,.gif"
6    multiple
7    (change)="onFileSelected($event)">
```

Mit dieser Lösung kann TypeScript-Code direkt auf das DOM-Element zugreifen und dessen Eigenschaften ändern.

18.2. Optimierung der Navigationsleiste

Ein weiteres Problem war, dass Benutzer versehentlich auf das "Smart Bill Converter"-Logo in der Navigationsleiste klickten. Das Logo war ursprünglich als Link zur Startseite

[22] Vgl. Angular Documentation: *ViewChild*, https://angular.io/api/core/ViewChild, abgerufen am 15.01.2025

gedacht, aber die Anwendung hat nur eine Seite. Ein Klick auf das Logo führte zu einem unnötigen Reload der Seite.

18.2.1. Umwandlung von Link zu Text

Die Lösung war einfach: Das `<a>`-Element wurde durch ein `<span>`-Element ersetzt:

Listing 48: Logo ohne Link-Funktionalität

```
<!-- Vorher: Klickbar -->
<a class="navbar-brand" routerLink="/">Smart Bill Converter</a>

<!-- Nachher: Nicht klickbar -->
<span class="navbar-brand">Smart Bill Converter</span>
```

18.2.2. CSS-Anpassungen

Zusätzlich wurde das CSS angepasst, um zu signalisieren, dass das Logo nicht klickbar ist:

Listing 49: Cursor-Style für nicht-klickbares Logo

```
.navbar-brand {
  font-size: 1.4rem;
  color: #495057;
  cursor: default;
  text-decoration: none;
}

.navbar-brand:hover {
  color: #495057;
}
```

Der `cursor: default` zeigt einen normalen Mauszeiger statt eines Zeige-Fingers. Das ist ein visuelles Signal, dass das Element nicht klickbar ist. Die Hover-Regel verhindert, dass sich die Farbe beim Darüberfahren ändert.

18.3. Responsive Design-Optimierung

Die Anwendung sollte auf verschiedenen Bildschirmgrößen gut funktionieren. Besonders auf Smartphones mussten einige Anpassungen gemacht werden.

18.3.1. Upload-Bereich für Mobile

Auf kleinen Bildschirmen wurde der Upload-Bereich zu groß und die Icons zu riesig:

Listing 50: Mobile-Optimierung des Upload-Bereichs

```
@media (max-width: 768px) {
  .upload-area {
    padding: 40px 15px;
    min-height: 200px;
  }

  .upload-content i {
    font-size: 3rem;
  }

  .preview-image {
    max-height: 300px;
  }
}
```

Die Padding-Werte wurden reduziert (von 60px auf 40px), die Mindesthöhe verkleinert (von 250px auf 200px) und die Icon-Größe angepasst (von 4rem auf 3rem). Das macht den Upload-Bereich kompakter und besser bedienbar auf mobilen Geräten.

18.3.2. Buttons und Schriftgrößen

Auch Buttons wurden für Touch-Bedienung optimiert:

Listing 51: Touch-freundliche Button-Größen

```
.btn-sm {
  min-width: 40px;
  min-height: 40px;
  border-radius: 4px;
}

@media (max-width: 768px) {
  .btn-lg {
    padding: 10px 16px;
    font-size: 1rem;
  }
}
```

Kleine Buttons bekommen eine Mindestgröße von 40x40 Pixeln. Das ist wichtig, weil Finger breiter sind als Mauszeiger. Gängige Empfehlungen sprechen von mindestens 44x44 Pixel für Touch-Targets.[23]

18.3.3. Weitere Optimierungen

Auf sehr kleinen Bildschirmen (unter 576px Breite) werden die Aktions-Buttons unter den Dateinamen verschoben statt rechts daneben. Die PDF-Vorschau nutzt auf Mobilgeräten mehr Platz (95% statt 90%) und hat weniger Padding.

[23] Vgl. Google Material Design: *Touch Targets*, https://m3.material.io/foundations/accessible-design/accessibility-basics, abgerufen am 15.01.2025

19. Evaluation mit realen Rechnungen (Frontend-Sicht)

Nach der Implementierung wurde das SmartBillConverter-Frontend mit verschiedenen Rechnungstypen getestet. Dieses Kapitel beschreibt die Erfahrungen aus Frontend-Sicht und zeigt, wie die Benutzeroberfläche in verschiedenen Szenarien funktioniert.

19.1. Erfolgsquoten der UI-Fehlerbehandlung

Die Fehlerbehandlung ist ein wichtiger Teil der Benutzeroberfläche. Wenn etwas schief geht, sollten Benutzer sofort verstehen, was das Problem ist und wie sie es beheben können.

19.1.1. Dateiformat-Validierung

Die erste Fehlerquelle ist das Hochladen falscher Dateiformate. Das System akzeptiert nur PDFs und gängige Bildformate (PNG, JPEG, BMP, TIFF, GIF). Wenn ein Benutzer versucht, eine andere Datei hochzuladen, wird sofort eine Fehlermeldung angezeigt:

Listing 52: Fehlerbehandlung bei falschen Dateiformaten

```
const supportedTypes = ['application/pdf', 'image/png',
                        'image/jpeg', 'image/bmp',
                        'image/tiff', 'image/gif'];
if (!supportedTypes.includes(file.type)) {
  this.showFileTypeError(file.name, file.type);
  return;
}
```

Die Fehlermeldung enthält den Dateinamen und den erkannten Dateityp. Das hilft dem Benutzer zu verstehen, warum die Datei abgelehnt wurde. Die Meldung verschwindet nach 5 Sekunden automatisch, kann aber auch manuell geschlossen werden.

19.1.2. Backend-Fehler

Wenn das Backend einen Fehler meldet (z.B. weil die Rechnung nicht lesbar ist oder wichtige Daten fehlen), wird die Fehlermeldung vom Backend direkt an den Benutzer weitergegeben:

Listing 53: Anzeige von Backend-Fehlern

```
error: (error) => {
  fileItem.status = 'error';
  fileItem.errorMessage = error.error?.error || 'Unbekannter Fehler';
  this.errorMessage = `Fehler beim Hochladen: ${error.error?.error}`;
  this.showError = true;
}
```

Die Fehlermeldung wird sowohl in der Dateiliste als auch in einem großen Alert-Banner oben angezeigt. Das stellt sicher, dass der Benutzer den Fehler nicht übersieht.

19.1.3. Netzwerkfehler

Bei Netzwerkproblemen (z.B. wenn das Backend nicht erreichbar ist) wird eine generische Fehlermeldung angezeigt. Diese Fehler sind seltener, aber wenn sie auftreten, ist es wichtig, dass der Benutzer informiert wird.

19.2. Benutzererfahrung bei der Konvertierung

Die Benutzererfahrung hängt stark von der Geschwindigkeit und Klarheit der Anwendung ab. Das SmartBillConverter-Frontend wurde so gestaltet, dass der Workflow möglichst einfach und verständlich ist.

19.2.1. Drei-Schritt-Workflow

Der typische Workflow besteht aus drei Schritten:

1. **Format auswählen**: Benutzer wählen zwischen ebInterface und ZUGFeRD. Die farbige Hervorhebung (grün vs. rot) macht die Auswahl klar.
2. **Dateien hochladen**: Benutzer laden eine oder mehrere Rechnungen hoch. Der Drag-and-Drop-Bereich ist groß und deutlich sichtbar. Die Vorschau zeigt sofort, welche Dateien ausgewählt wurden.

3. **Konvertieren und Herunterladen**: Ein Klick auf "Konvertieren" startet die Verarbeitung. Die Progress Bars zeigen den Status jeder Datei. Nach erfolgreicher Konvertierung erscheint der Download-Button.

Dieser Workflow ist selbsterklärend und benötigt keine Anleitung. Tests mit Benutzern zeigten, dass die meisten den Prozess ohne Hilfe durchführen konnten.

19.2.2. Verarbeitungsgeschwindigkeit

Die Verarbeitungsgeschwindigkeit hängt vom Backend ab, aber das Frontend gibt kontinuierlich Feedback:

- Während des Uploads zeigt die Progress Bar den Status "Wird verarbeitet" mit animierten Streifen
- Bei Multi-File-Verarbeitung sieht der Benutzer, welche Datei gerade verarbeitet wird
- Nach Abschluss ändert sich die Progress Bar zu grün mit "Erfolgreich konvertiert"

Die meisten Rechnungen werden in 5-15 Sekunden verarbeitet. In dieser Zeit bleibt die Benutzeroberfläche reaktiv. Benutzer können weitere Dateien hinzufügen oder die Vorschau öffnen.

19.2.3. Multi-File-Verarbeitung

Die Möglichkeit, mehrere Dateien gleichzeitig hochzuladen, verbessert die Benutzererfahrung erheblich. Statt jede Rechnung einzeln zu konvertieren, können Benutzer alle Dateien auf einmal auswählen. Das System verarbeitet sie dann nacheinander und zeigt den Fortschritt für jede Datei an.

Bei Tests mit 10 Rechnungen gleichzeitig funktionierte das System problemlos. Die sequentielle Verarbeitung verhindert, dass das Backend überlastet wird.

19.3. Beispiele der UI-Darstellung

Die folgenden Abschnitte beschreiben typische Szenarien und wie die Benutzeroberfläche darauf reagiert.

19.3.1. Erfolgreiche Konvertierung

Nach erfolgreicher Konvertierung zeigt die Benutzeroberfläche:

- **Grüne Progress Bar** (100% gefüllt)
- **Status-Text**: " Erfolgreich konvertiert" in grüner Schrift
- **Download-Button**: Ein grüner Button mit Download-Icon erscheint neben der Datei
- **Dateiname**: Der XML-Dateiname wird aus der Rechnungsnummer und dem Format zusammengesetzt (z.B. `ebinterface_invoice_123.xml`)

Der Benutzer kann sofort das XML herunterladen oder weitere Dateien verarbeiten. Die erfolgreiche Konvertierung ist eindeutig durch die grüne Farbe erkennbar.

19.3.2. Fehlerfall

Wenn die Konvertierung fehlschlägt, ändert sich die Darstellung:

- **Rote Progress Bar** (auf 0% zurückgesetzt)
- **Status-Text**: " Fehler aufgetreten" in roter Schrift
- **Fehlermeldung**: Die genaue Fehlermeldung vom Backend wird unter dem Status angezeigt
- **Alert-Banner**: Zusätzlich erscheint oben ein rotes Banner mit der Fehlermeldung

Die Fehlermeldung erklärt, was schief gelaufen ist. Typische Fehler sind "Rechnung konnte nicht gelesen werden" oder "Rechnungsnummer fehlt". Der Benutzer kann die fehlerhafte Datei entfernen und eine andere versuchen.

19.3.3. Vorschau-Funktion

Die Vorschau-Funktion erlaubt es, hochgeladene Dateien vor der Konvertierung zu überprüfen:

- **PDF-Vorschau**: PDFs werden mit dem ng2-pdf-viewer angezeigt. Benutzer können durch die Seiten navigieren und zoomen.
- **Bild-Vorschau**: Bilder werden in Originalgröße angezeigt, automatisch skaliert auf die verfügbare Fläche.

- **Modal-Dialog**: Die Vorschau erscheint in einem großen Overlay, das den Rest der Seite abdunkelt.

Die Vorschau ist nützlich, um sicherzustellen, dass die richtige Datei hochgeladen wurde. Ein Klick außerhalb des Modals oder auf den Schließen-Button beendet die Vorschau.

19.3.4. Multi-File-Ansicht

Wenn mehrere Dateien hochgeladen wurden, zeigt die Benutzeroberfläche:

- **Karten-Header**: "X Dateien (Y MB gesamt)" mit Gesamt-Entfernen-Button
- **Liste der Dateien**: Jede Datei als eigener Eintrag mit Icon, Name, Größe
- **Individuelle Progress Bars**: Jede Datei hat ihre eigene Fortschrittsanzeige
- **Aktions-Buttons**: Pro Datei gibt es Buttons für Vorschau, Download und Entfernen

Wenn die Verarbeitung läuft, werden die Dateien nacheinander bearbeitet. Die gerade aktive Datei zeigt den animierten "Wird verarbeitet"-Status, alle anderen warten oder sind bereits fertig.

19.3.5. Leerer Zustand

Wenn keine Dateien hochgeladen wurden, zeigt die Upload-Area:

- **Cloud-Upload-Icon**: Ein großes Icon signalisiert, dass hier Dateien hochgeladen werden können
- **Anweisungstext**: "Dateien hier ablegen oder klicken zum Auswählen"
- **Format-Hinweis**: "PDF, PNG, JPG, BMP, TIFF, GIF - Max. 10MB pro Datei"
- **Multi-File-Hinweis**: "Mehrere Dateien gleichzeitig möglich"

Diese Texte helfen neuen Benutzern zu verstehen, was sie tun müssen. Die Anweisungen sind kurz und klar formuliert.

Teil VII.

Ergebnisse, Diskussion und Ausblick

20. Grenzen und zukünftige Arbeiten

[Dieses Kapitel wird gemeinsam verfasst.]

20.1. Bekannte Einschränkungen

[Frontend & Backend]

20.2. Zukünftige Erweiterungen

[UI/UX, weitere Standards]

21. Schlussbetrachtung

[Dieses Kapitel wird gemeinsam verfasst.]

21.1. Zusammenfassung der Ergebnisse und Beiträge

[Gemeinsam]

21.2. Ausblick und Übertragbarkeit

[Gemeinsam]

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Anhang