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

Nicht jede digitale Datei, die eine Rechnung darstellt, ist auch wirklich eine elektronische Rechnung im rechtlichen und technischen Sinne. Ein eingescanntes Papier oder ein aus Word exportiertes PDF ist zwar für Menschen lesbar, für Software aber nur ein Bild ohne strukturierten Inhalt. Erst wenn die Rechnungsdaten in einem maschinenlesbaren Format vorliegen, kann eine Software wie der *SmartBillConverter* diese automatisch verarbeiten.

Dieses Kapitel erklärt die zwei für das Projekt wichtigen XML-basierten E-Rechnungsstandards - das österreichische *ebInterface 6.1* und das deutsche Hybridformat *ZUGFeRD 2.3* - sowie deren Bezug zur europäischen Norm EN 16931. Beide Standards lösen das gleiche Problem auf unterschiedliche Weise und haben dabei jeweils eigene Stärken und Schwächen.

2.1. ebInterface 6.1 (Österreich)

2.1.1. Zweck und Historie

Unter der Trägerschaft der AUSTRIAPRO, einer Serviceeinrichtung der Wirtschaftskammer Österreich (WKO), entstand ebInterface als nationale Antwort auf die wachsende Nachfrage nach einem einheitlichen, XML-basierten Rechnungsaustausch. Der Entwicklungsbeginn in den frühen 2000er-Jahren spiegelte eine strukturelle Lücke wider: Während Großkonzerne auf komplexe EDI-Systeme zurückgreifen konnten, fehlte kleinen und mittleren Unternehmen ein zugängliches, breit akzeptiertes Dateiformat. Über aufeinanderfolgende Versionen (4.0, 5.0, 6.0) wurde das Schema bis zur gegenwärtig verbindlichen Version 6.1 weiterentwickelt.

Einen entscheidenden Verbreitungsschub erlebte ebInterface im Jahr 2014, als die verpflichtende elektronische Rechnungslegung gegenüber österreichischen Behörden gesetzlich verankert wurde. Seitdem ist ebInterface die Pflichtgrundlage für alle Rechnungen, die über das Unternehmensserviceportal (USP) eingereicht werden, was dazu geführt hat, dass das Format auch außerhalb des öffentlichen Sektors immer weiter verbreitet ist.[1]

2.1.2. Kernelemente und Struktur

Im Gegensatz zu ZUGFeRD enthält eine ebInterface-Datei kein lesbares PDF-Layout - sie ist ausschließlich eine XML-Datei mit den Rechnungsdaten. Wer eine ebInterface-Rechnung mit eigenen Augen lesen will, braucht entweder einen speziellen Viewer oder muss sie zuerst mit XSLT in ein lesbares Format umwandeln.

Die Struktur einer ebInterface 6.1 Datei ist hierarchisch aufgebaut und besteht aus folgenden Hauptbereichen:

- **Root-Element**: `<Invoice>` definiert die Grundeigenschaften der Rechnung wie Währung, Sprache und Dokumententitel.
- **Header-Daten**: Enthält eine eindeutige Rechnungsnummer (`InvoiceNumber`), das Rechnungsdatum (`InvoiceDate`) und den Leistungszeitraum (`Delivery`).
- **Biller (Rechnungssteller)**: Beinhaltet detaillierte Informationen zum Rechnungssteller wie Anschrift, Kontaktdaten, Bankverbindung und die gesetzlich vorgeschriebenen UID-Nummer (VAT Identification Number).
- **InvoiceRecipient (Rechnungsempfänger)**: Analog zum Biller entählt dieser Bereich die Daten des Leistungsempfängers. Hier ist oft die Auftragsreferenz (Order Reference) besonders wichtig für die automatisierte Zuordnung der Rechnung.
- **Details (Positionen)**: Das Herzstück der Rechnung. Hier werden in einer Liste (`ItemList`) die einzelnen Positionen (`ListLineItem`) aufgeführt. Jede Position enthält Menge, Einheit, Beschreibung, Einzelpreis, Zeilensumme und Steuerreferenz.
- **Tax (Steuern)**: Eine Zusammenfassung der Steuerbeträge, sortiert nach Steuersätzen. Dies ist wichtig für die Prüfung des Vorsteuerabzugs.

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

- **PaymentConditions**: Beinhaltet Zahlungsziele, Fälligkeitsdaten und Skonto-Informationen.

2.1.3. Pflichtfelder und Validierung

Die Gültigkeit einer ebInterface-Rechnung wird durch ein XML Schema (XSD) definiert. Pflichtfelder sind die Angaben, die das Umsatzsteuergesetz (UStG) für eine ordnungsgemäße Rechnung vorschreibt. In Österreich gehören dazu unter anderem:

- Name und Anschrift des empfangenden und liefernden Unternehmers.
- Menge und gebräuchliche Bezeichnung der Waren oder erbrachten Leistungen.
- Kalendertag der Lieferung oder Leistung.
- Entgelt (Netto) und der darauf hinfällige Steuerbetrag.
- Der anzuwendende Steuersatz.
- Ausstellungsdatum und fortlaufende Rechnungsnummer.
- UID-Nummer des Rechnungsstellers (und ab 10.000 Euro Brutto auch des Empfängers).

Auffällig bei ebInterface 6.1 ist die strenge Typ-Prüfung: Datumsfelder akzeptieren nur das ISO-8601-Format, Betragsfelder müssen Dezimalzahlen sein. Das ist bei der automatischen Generierung durch KI sehr nützlich, weil Fehler wie ein falsch formatiertes Datum (z.B. TT.MM.JJJJ statt JJJJ-MM-TT) sofort als XSD-Validierungsfehler auffallen, statt unbemerkt falsche Werte in die Rechnung einzubauen.

2.1.4. Beispielhafte XML-Struktur

Um die Struktur zu veranschaulichen, zeigt das folgende Beispiel einen gekürzten Ausschnitt einer validen ebInterface 6.1 Rechnung. Man erkennt deutlich die hierarchische Gliederung und die aussagekräftigen Tag-Namen, die eine Umsetzung 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>
```

```
9       <VATIdentificationNumber>ATU12345678</VATIdentificationNumber>
10      <Address>
11        <Name>Musterfirma GmbH</Name>
12        <Street>Hauptstrasse 1</Street>
13        <Town>Wien</Town>
14        <ZIP>1010</ZIP>
15        <Country>Austria</Country>
16      </Address>
17    </Biller>
18    <Details>
19      <ItemList>
20        <ListLineItem>
21          <Description>Software Entwicklung</Description>
22          <Quantity Unit='h'>10.00</Quantity>
23          <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>
```

Was an diesem Beispiel auffällt: Die Einzelpositionen im `Details`-Block und die Steuerberechnung im `Tax`-Block müssen rechnerisch übereinstimmen. Genau das ist bei der KI-basierten Generierung das häufigste Problem, weil LLMs solche Berechnungen nicht zuverlässig selbst durchführen.

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

2.2.1. Das hybride Konzept

ZUGFeRD, entwickelt vom *Forum elektronische Rechnung Deutschland (FeRD)*, funktioniert ganz anders als reine XML-Formate wie ebInterface. Statt einer eigenständigen XML-Datei wird eine ganz normale PDF-Rechnung erstellt, in die die XML-Daten unsichtbar eingebettet werden. Technisch gesehen ist das eine PDF/A-3-Datei, die eine XML-Datei (meistens `factur-x.xml` oder `zugferd-invoice.xml`) als versteckten Anhang enthält.

Durch diese Dualität entsteht ein Format mit zwei unabhängig nutzbaren Informationsschichten:

- **Visuelle Ebene**: Das PDF-Rendering entspricht dem gewohnten Rechnungslayout und ist ohne Spezialsoftware lesbar.

- **Strukturierte Datenschicht**: Das eingebettete XML liefert dieselben Informationen in maschinenverarbeitbarer Form.

Diese Architekturentscheidung löst ein in der Praxis häufig auftretendes Akzeptanzproblem: Empfänger ohne automatisierte Verarbeitungsinfrastruktur behandeln ZUGFeRD-Dokumente schlicht als gewöhnliche PDFs, während systemseitig die XML-Nutzlast selektiv extrahiert und weiterverarbeitet werden kann.

2.2.2. Profile und Konformität

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

1. **MINIMUM**: Enthält nur grundlegende Daten, um eine Buchungshilfe zu bieten. Es erfüllt nicht die Anforderungen einer steuerrechtlich gültigen Rechnung.
2. **BASIC WL (Without Lines)**: Enthält Kopfdaten und Summen, jedoch keine einzelnen Positionsdaten. Nützlich für einfache Verbuchung, aber eingeschränkt in der Prüfung.
3. **BASIC**: Erfüllt die Anforderungen des deutschen UStG für Rechnungen unter bestimmten Grenzen, stellt aber eine nur einen kleinen Teil der EN 16931 dar.
4. **EN 16931 (COMFORT)**: Das Standardprofil, welches die europäische Norm vollständig abbildet und für den grenzüberschreitenden Verkehr als auch für B2G (Business to Government) geeignet ist. Dies ist das Zielformat für den *SmartBillConverter*.
5. **EXTENDED**: Ergänzt erweiterte branchenspezifische Erweiterungen, die über die Norm hinausgehen.

2.2.3. CII-Struktur (Cross Industry Invoice)

Für die XML-Struktur verwendet ZUGFeRD das *Cross Industry Invoice* (CII) Schema der UN/CEFACT. Dieser Standard wurde eigentlich für komplexe globale Lieferketten entwickelt und ist daher für eine einfache Rechnung deutlich umfangreicher als nötig. Ein gutes Beispiel: Ein einfacher Einzelpreis ist in CII kein einfacher Zahlenwert, sondern

ein ganzes Objekt mit Betrag, Währung, Basismenge und Einheitencode als separate Felder.

Beispielhafte Verschachtelung in CII: `SupplyChainTradeTransaction` → `IncludedSupplyChainTradeLineItem` → `SpecifiedLineTradeAgreement` → `NetPriceProductTradePrice` → `ChargeAmount`. Diese hohe Komplexität macht die Implementierung eines Mappers anspruchsvoll, da hunderte von Pfaden korrekt ausgefüllt werden müssen, um Validierungsfehler zu vermeiden.[2] Das folgende Beispiel stellt ein Ausschnitt einer ZUGFeRD-XML dar, der die Komplexität im Vergleich zu ebInterface verdeutlicht. Es sollten die tiefen Verschachtelungen für einfache Informationen wie den Steuerbetrag beachtet werden.

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

Interessant ist, dass die Norm EN 16931 zwei völlig unterschiedliche XML-Formate als gleichwertig anerkennt: UBL (ISO/IEC 19845) und UN/CEFACT CII. Das liegt daran, dass verschiedene Länder und Netzwerke unterschiedliche Präferenzen haben. Peppol, das europaweite E-Procurement-Netzwerk, nutzt hauptsächlich UBL, während ZUGFeRD auf CII basiert. Das führt dazu, dass es in Europa kein einheitliches Format gibt.

[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

2.3.1. Strukturelle Unterschiede

UBL hat für jeden Dokumenttyp ein eigenes Schema: eines für Bestellungen (`Order`), eines für Rechnungen (`Invoice`), eines für Lieferscheine (`DespatchAdvice`) usw. Das macht die Struktur übersichtlich und relativ einfach zu lesen. **CII** hingegen versucht alle möglichen Geschäftsprozesse in einem einzigen Schema abzubilden, wobei eine Rechnung dann nur ein Sonderfall von Lieferkettenoperationen ist. Das macht CII mächtiger, aber auch deutlich komplizierter.

Ein konkretes Beispiel ist die Handhabung von Steuern:

- In **UBL** werden Steuern häufig direkt auf Zeilenebene referenziert (`Classified-TaxCategory`).
- In **CII** gibt es komplexe `ApplicableTradeTax`-Strukturen, die sowohl auf Dokumentenebene (Summen) als auch auf Zeilenebene (Referenzen) übereinstimmend geführt werden müssen.

2.3.2. Herausforderungen für den Konverter

Für den *SmartBillConverter* bedeutet das eine wichtige Anforderung: Das interne Datenmodell darf nicht zu eng an ebInterface angelehnt sein, weil sonst die Umwandlung nach CII sehr schwierig wird. ebInterface fasst zum Beispiel alle Rechnungsstellerdaten unter einem einzigen *Biller*-Element zusammen, während CII die gleichen Daten auf viele Container wie *SupplyChainTradeTransaction* verteilt. Ein direktes Mapping von ebInterface auf CII würde zu sehr unübersichtlichem Code führen.

Die Lösung war, ein eigenes internes C#-Objekt als Zwischenschritt zu verwenden. Die KI befüllt dieses Objekt mit den extrahierten Daten, und danach wird es von separaten Klassen in das jeweilige Zielformat (ebInterface oder ZUGFeRD) umgewandelt. Tabelle 1 fasst die bedeutenden Unterschiede zusammen, die bei der Umsetzung beachtet wurden.

Merkmal	**ebInterface 6.1**	**ZUGFeRD 2.3 (CII)**
Basis-Standard	National (AUSTRIAPRO)	International (UN/CEFACT)
Dateiformat	Reines XML	PDF/A-3 mit eingebettetem XML
Struktur-Tiefe	Flach bis Mittel	Sehr tief verschachtelt
Steuer-Logik	Zentraler Tax-Block	Verteilt (Line & Header)
Pflichtfelder	Fokus auf UStG (AT)	Fokus auf EN 16931 (EU)
Visualisierung	Benötigt Stylesheet	PDF ist menschenlesbar

Tabelle 1.: Vergleich zwischen ebInterface und ZUGFeRD

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

Ein XML-Dokument kann technisch korrekt aufgebaut sein und trotzdem falsche Inhalte haben. Gerade bei der automatischen Generierung durch KI ist das ein häufiges Problem. Im *SmartBillConverter* werden deshalb beide Arten von Fehlern getrennt geprüft.

2.4.1. Syntaktische Validierung (XSD)

Das XSD (XML Schema Definition) des jeweiligen Standards beschreibt genau, wie das XML aufgebaut sein muss. Es legt fest, welche Felder vorhanden sein müssen, in welcher Reihenfolge sie kommen dürfen und welchen Datentyp sie haben sollen. Typische Prüfpunkte sind:

- Vollständigkeit der obligatorischen Felder?
- Korrektheit des Datumsformats (strikt ISO 8601: YYYY-MM-DD)?
- Numerische Typen ohne unerlaubte String-Repräsentationen?
- Einhaltung der Elementreihenfolge - insbesondere CII toleriert keinerlei Abweichungen.

Ein Dokument, das diesen syntaktischen Vertrag verletzt, wird von konformen Empfangssystemen in aller Regel bereits beim Einlesen verworfen, ohne dass eine inhaltliche Prüfung auch nur beginnt.

2.4.2. Semantische Validierung (Business Rules)

Nur weil das XML technisch korrekt aufgebaut ist, heißt das noch nicht, dass die Inhalte auch stimmen. Deshalb legt die Norm EN 16931 zusätzlich eine Liste von Geschäftsregeln fest, die sicherstellen sollen, dass die Beträge und Steuern rechnerisch korrekt und widerspruchsfrei sind. Diese Regeln werden meistens mit *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 ist, dass LLMs keine Taschenrechner sind. Sie schätzen auf Basis von Wahrscheinlichkeiten den nächsten Token, anstatt wirklich zu rechnen. In der Praxis heißt das: Ein LLM kann eine plausibel aussehende, aber falsche Steuersumme ausgeben, weil es den Wert errät statt ihn zu berechnen:

- 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. Deshalb wurde im *SmartBillConverter* eine klare Aufgabenteilung eingebaut: Die KI liefert nur die Einzelwerte (Einzelbeträge, Steuersätze, Mengen), während alle Summen, Rundungen und Steuerberechnungen fest im C#-Backend berechnet werden. Nur so kann sichergestellt werden, dass die Regel BR-S-08 immer korrekt eingehalten wird.[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 automatische Verarbeitung von Rechnungen ist ein altbekanntes Problem. Frühere Systeme haben meistens mit Templates gearbeitet: Für jeden Lieferanten wurde manuell festgelegt, an welcher Stelle im Dokument zum Beispiel die Rechnungsnummer steht. Sobald sich das Layout auch nur minimal ändert, hört so ein System auf zu funktionieren. Der *SmartBillConverter* geht deshalb einen anderen Weg und verwendet Large Language Models (LLMs), die das Layout eines Dokuments selbstständig verstehen können. Die folgenden Abschnitte erklären die dafür nötigen technischen Grundlagen.

3.1. PDFs und Textextraktion

3.1.1. Die Natur des PDF-Formats

PDFs sind keine Textdokumente mit Layoutinformationen - sie sind Renderinganweisungen, die zufällig auch Text enthalten können. Das Format speichert primär geometrische Befehle vom Typ „Zeichne Glyphe A an Koordinate $(100, 200)$ in Schrift Helvetica 12 pt“. Ob dieses A zum Wort „Rechnungsnummer“ gehört, ob dieses Wort eine Tabelüberschrift ist oder ob die nachfolgende Zahl einen Netto- oder Bruttobetrag darstellt - all das ist im PDF-Stream nicht kodiert und muss von Extraktionssoftware heuristisch erschlossen werden.

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

Um dieses Problem zu lösen, wird die .NET-Bibliothek *PdfPig* verwendet. Sie analysiert die genauen Koordinaten jedes einzelnen Buchstabens im PDF und versucht daraus die richtige Lesereihenfolge zu rekonstruieren. Buchstaben, die dicht nebeneinander auf gleicher Höhe liegen, werden zu Wörtern zusammengefügt, und Wörter auf der gleichen Linie bilden dann eine Textzeile. Besonders schwierig sind dabei Tabellen: Die Linien einer Tabelle im PDF sind nur gewöhnliche Vektorgrafiken ohne Verbindung zum Text. Deshalb kann die Software oft nicht erkennen, welche Zahl zu welcher Spalte gehört. Trotzdem ist diese direkte Textextraktion aus dem PDF deutlich besser als OCR, weil dabei keine Buchstabenverwechslungen wie bei „8“ und „B“ passieren können.[4]

3.2. OCR mit Tesseract

Liegt eine Rechnung nicht als nativ-digitales PDF vor - sei es als Scan einer physischen Vorlage oder als fotografische Aufnahme - liefert jede koordinatenbasierte Textextraktion ein leeres Ergebnis. Der Inhalt existiert in diesen Fällen ausschließlich als Pixelmuster, das erst durch Optical Character Recognition (OCR) in maschinenlesbaren Text überführt werden muss.

[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

3.2.1. Funktionsweise von Tesseract

Als OCR-Engine wird *Tesseract* verwendet, ein Open-Source-Projekt das ursprünglich von HP entwickelt wurde und heute von Google weiterentwickelt wird. Ab Version 4.0 basiert Tesseract auf LSTM-Netzen (Long Short-Term Memory), einer Art neuronalem Netz das gut mit sequenziellen Daten wie Text umgehen kann. Dadurch erkennt es Buchstaben deutlich zuverlässiger als ältere regelbasierte Methoden. Die Verarbeitung läuft grob in vier Schritten ab:

1. **Layout-Analyse**: Identifikation von Textregionen, Spalten und Zeilen im Bild.
2. **Baseline-Fitting**: Erkennen der Grundlinie jeder Textzeile - wichtig damit Buchstaben richtig segmentiert werden können, auch wenn der Scan leicht schief ist.
3. **Zeichenerkennung**: Das LSTM-Netz analysiert kleine Bildausschnitte und ordnet ihnen Buchstaben mit einem Konfidenzwert zu.
4. **Worterkennung**: Wörterbücher helfen dabei, dass die erkannten Zeichenfolgen auch sinnvolle Wörter ergeben.

3.2.2. Einflussfaktoren auf die Qualität

Die erzielte Erkennungsqualität hängt dabei in erheblichem Maß von der Eingabequalität und vorgeschalteter Bildaufbereitung ab:

- **Auflösung**: Unter 300 DPI wird die Erkennungsrate deutlich schlechter, weil die feinen Linien einzelner Buchstaben zu wenige Pixel haben um sicher erkannt zu werden.
- **Binarisierung**: Das Umwandeln in Schwarz-Weiß muss bei ungleichmäßiger Beleuchtung oder Schatten adaptiv erfolgen, also für verschiedene Bildbereiche getrennt berechnet werden, statt einen einzigen fixen Schwellwert für das gesamte Bild zu verwenden.
- **Geraderichten (Deskewing)**: Selbst eine Drehung um nur zwei Grad reicht aus, damit Tesseract die Textzeilen nicht mehr richtig findet. Scans müssen deshalb immer rechnerisch gerade gerückt werden.

Im Projekt werden die deutschsprachigen Trainingsdaten (`deu.traineddata`) geladen, da österreichische Rechnungen spezifische Vokabeln und Umlaute enthalten, deren fehler-

hafte Erkennung nachgelagerte Extraktionsschritte unverhältnismäßig stark beeinträchtigt. Die verbleibende Fehlerquote wird durch die anschließende LLM-Korrekturschicht aufgefangen.

3.3. LLMs für Informationsextraktion

Der größte Unterschied zu älteren Ansätzen ist, dass kein Entwickler mehr für jedes mögliche Rechnungslayout eigene Erkennungsregeln schreiben muss. Stattdessen wird einfach der gesamte Text der Rechnung an das LLM übergeben, und das Modell strukturiert die Daten selbstständig in das gewünschte Format.

3.3.1. Architektur und Funktionsweise

Moderne LLMs basieren auf der Transformer-Architektur und verwenden Attention-Mechanismen, um Zusammenhänge zwischen verschiedenen Teilen eines Textes zu erkennen. Wichtig für die Extraktion ist das sogenannte *In-Context Learning*: Das Modell wurde zwar nie speziell auf Rechnungen trainiert, hat aber durch sein allgemeines Training gelernt, dass eine IBAN meistens bei Wörtern wie „Bankverbindung“ oder „Kontonummer“ steht. Außerdem kann das Modell Synonyme automatisch auflösen: Ob auf der Rechnung „Total“, „Zahlbetrag“, „Rechnungssumme“ oder „zu zahlender Betrag“ steht - das Modell erkennt in allen Fällen, dass dasselbe Feld `TotalAmount` gemeint ist.

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 Problem bei LLMs ist die Halluzination. LLMs sind so trainiert, dass sie immer eine möglichst sinnvolle Antwort generieren. Wenn auf einer Rechnung zum Beispiel kein Lieferdatum steht, das JSON-Schema aber eines erwartet, dann erfindet das Modell einfach ein Datum - meistens das Rechnungsdatum. Für eine Finanzanwendung ist das ein ernstes Problem, weil ein erfundenes Lieferdatum in einer archivierten Rechnung steuerrechtliche Folgen haben kann. Der Prompt muss deshalb das Modell klar anweisen, in solchen Fällen `null` zurückzugeben statt zu raten.

3.4. Prompt-Design für deterministische Ausgabe

Prompt Engineering bedeutet, die Eingabe an das Modell so zu formulieren, dass man die gewünschten Ergebnisse bekommt. Bei der Datenextraktion geht es dabei nicht um Kreativität, sondern darum, dass das Modell bei gleicher Eingabe immer die gleiche Ausgabe liefert.

3.4.1. Techniken

- **System Prompting**: Am Anfang des Prompts wird dem Modell eine klare Rolle gegeben, z.B.: „Du bist ein Datenextraktions-Assistent. Antworte ausschließlich mit einem gültigen JSON-Objekt, ohne zusätzlichen Text.“
- **JSON Mode / Structured Output**: Moderne APIs wie Google Gemini oder OpenAI bieten einen speziellen Modus an, bei dem die Antwort des Modells immer gültiges JSON ist. Das ist zuverlässiger als einfach im Prompt darum zu bitten.
- **Schema Injection**: Das Ziel-JSON-Schema wird als Teil des Prompts übergeben, sodass das Modell die erwarteten Feldnamen, Typen und Pflichtfelder als Kontext erhält.
- **Chain-of-Thought Unterdrückung**: Bei manchen Aufgaben hilft es, das Modell Schritt für Schritt denken zu lassen. Bei der Datenextraktion ist das aber hinderlich,

weil das Modell dann Erklärungen in die Antwort mischt. Mit der Anweisung „Keine Erklärungen, nur das JSON-Objekt“ wird das verhindert.

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 konkretes Problem war, dass dasselbe Modell beim gleichen Prompt manchmal unterschiedliche Ergebnisse geliefert hat - zum Beispiel Daten mal im Format YYYY-MM-DD und mal als DD.MM.YYYY. Das lässt sich durch genaue Formatierungsanweisungen im Prompt („Alle Daten im Format YYYY-MM-DD“) und die Einstellung Temperature = 0 beheben, weil das Modell dann immer den wahrscheinlichsten Wert auswählt und die Ausgabe stabiler wird.

3.5. Datenschutz und Sicherheit in KI-APIs

Rechnungen enthalten viele sensible Informationen: personenbezogene Daten wie Namen, Adressen und UID-Nummern, aber auch Geschäftsinterna wie Preise, Lieferanten und

Projektbezeichnungen. Wenn diese Daten an eine Cloud-KI-API geschickt werden, verlassen sie die eigene Infrastruktur - was aus Datenschutzsicht problematisch ist.

3.5.1. Risikoanalyse

- **Modelltraining auf Kundendaten**: Ohne einen speziellen Enterprise-Vertrag können Anbieter wie Google oder OpenAI die eingesendeten Anfragen für das Training ihrer Modelle verwenden. Im normalen Tarif ist das erlaubt, im Enterprise-Tarif wird es vertraglich ausgeschlossen.
- **Daten im Klartext beim Anbieter**: Auch wenn die Übertragung per TLS verschlüsselt ist, muss der Anbieter die Daten für die Verarbeitung entschlüsseln. Kurzzeitig liegen die Rechnungsdaten also im Klartext auf den Servern des Anbieters. Bei einem lokal betriebenen Modell fällt dieses Risiko weg.
- **Rechtliche Unsicherheit**: US-Anbieter fallen unter den CLOUD Act, der amerikanischen Behörden bestimmte Zugriffmöglichkeiten gibt, selbst wenn die Daten auf europäischen Servern gespeichert sind. Das EU-US Data Privacy Framework, das solche Datentransfers regelt, ist außerdem politisch unsicher und wurde schon mehrfach vor Gericht angefochten.

3.5.2. Lokale Alternativen

Eine datenschutzfreundliche Alternative wäre, ein Open-Source-Modell wie Llama 3 oder Mistral lokal auf eigener Hardware zu betreiben, zum Beispiel mit dem Tool Ollama. Dann verlassen die Rechnungsdaten nie die eigene Infrastruktur. Der Nachteil ist, dass lokale Modelle deutlich mehr Hardware-Ressourcen brauchen. Für mehrseitige Rechnungen mit viel Text werden große Modelle mit 70B+ Parametern benötigt, die ohne eine leistungsstarke GPU sehr langsam sind. Kleinere Modelle mit 7B oder 13B Parametern liefern bei komplexen Rechnungen oft zu schlechte Ergebnisse. Für den *SmartBillConverter* wurde deshalb die Cloud-API von Gemini gewählt, weil die Qualität besser ist und die Entwicklung damit schneller geht. Für einen richtigen Produktivbetrieb müsste man aber auf einen Enterprise-Tarif mit entsprechenden Datenschutzgarantien umsteigen.

Teil II.

Systemarchitektur und Design

4. Gesamtarchitektur

Das Kapitel beschreibt, wie die einzelnen Teile von SmartBillConverter zusammenspielen. Statt nur eine Auflistung von Komponenten zu liefern, geht es hier darum zu verstehen, warum das System so aufgebaut wurde und welchen Weg eine Rechnung durch das System nimmt, bevor am Ende ein fertiges XML rauskommt.

4.1. Kontextdiagramm und Use Cases

SmartBillConverter ist eine Webanwendung, bei der der Benutzer über ein Angular-Frontend eine Rechnung hochlädt. Das kann entweder ein PDF oder ein Bild sein. Das Backend übernimmt dann den Rest komplett alleine: Text extrahieren, mit KI normalisieren, in ein Rechnungsformat übersetzen und als XML ausliefern.

Die wichtigsten Akteure im System sind:

- **Benutzer** – lädt eine Rechnung hoch und bekommt am Ende das fertige XML zurück
- **Backend-API** – nimmt die Datei entgegen und koordiniert alle Verarbeitungsschritte
- **Gemini API (Google)** – externe KI, die den rohen Rechnungstext in strukturierte Daten umwandelt
- **PostgreSQL-Datenbank** – speichert verarbeitete Rechnungen persistent

Die wichtigsten Use Cases, die das System abdeckt, sind:

1. PDF-Rechnung hochladen und als ebInterface 6.1 XML herunterladen
2. Bild einer Rechnung per OCR verarbeiten und als ZUGFeRD 2.3 XML bekommen
3. Bereits verarbeitete Rechnungen in der Datenbank ansehen und erneut herunterladen

Der gesamte Verarbeitungsprozess läuft serverseitig, also muss der Browser nichts selbst rechnen. Das macht das Frontend zur reinen Anzeige- und Uploadschicht.

4.2. Komponentenübersicht

Das Backend besteht aus mehreren klar voneinander getrennten Schichten. Jede Schicht hat genau eine Aufgabe, was das Testen und Warten deutlich einfacher macht.

- **Controller-Schicht** – nimmt HTTP-Anfragen entgegen und leitet sie weiter. Es gibt je einen Controller für Bildverarbeitung, Invoices, ZUGFeRD, Downloads und allgemeines Processing.
- **Service-Schicht** – enthält die gesamte Geschäftslogik. Wichtige Services sind `PdfExtractionService`, `OcrService`, `LlmConversionService`, `EbInterfaceService` und `ZugferdService`.
- **Repository-Schicht** – kapselt den Datenbankzugriff. Der `InvoiceRepository` ist über eine `IInvoiceRepository`-Schnittstelle angebunden, damit ein späterer Wechsel des Persistence-Backends kein Problem wäre.
- **Datenbank** – PostgreSQL läuft als Docker-Container. Entity Framework Core mit Code-First-Migrationen verwaltet das Schema automatisch.
- **Frontend (smart-bill-ui)** – Angular-SPA. Kommuniziert ausschließlich über HTTP mit der Backend-API. Rendert Formulare, Tabellen und Download-Buttons, kümmert sich aber um keine Logik.

Die Komponenten sind also in einem klassischen dreischichtigen Modell aufgebaut: Präsentation (Angular), Logik (Services) und Datenhaltung (Repository + PostgreSQL).

4.3. Datenfluss

Wenn der Benutzer eine Datei hochlädt, durchläuft sie mehrere Stufen, bis das XML fertig ist. Der genaue Ablauf hängt davon ab, ob eine PDF oder ein Bild übergeben wird, aber die mittleren Schritte sind identisch.

1. **Upload** – Der Browser schickt die Datei per HTTP-POST an die Backend-API. Je nach Dateityp landet die Anfrage entweder beim `ProcessingController` (PDF) oder beim `ImageProcessingController` (Bild).
2. **Textextraktion** –
   - PDF: `PdfExtractionService` nutzt PdfPig, um den Text direkt aus dem PDF zu lesen.
   - Bild: `OcrService` gibt das Bild an Tesseract weiter und bekommt den erkannten Text zurück.
3. **KI-Normalisierung** – Der rohe Text wird an den `LlmConversionService` übergeben. Dieser baut einen Prompt zusammen und schickt ihn an die Gemini API. Die Antwort ist ein JSON-Objekt, das einer definierten `InvoiceData`-Struktur entspricht.
4. **Deserialisierung und Validierung** – Das JSON wird in ein C#-Objekt deserialisiert. Falls Pflichtfelder fehlen, greift ein Fallback-Mechanismus und füllt die fehlenden Werte mit sinnvollen Standardwerten.
5. **XML-Generierung** –
   - `EbInterfaceService` erzeugt ein ebInterface 6.1 XML über `XmlSerializer`.
   - `ZugferdService` erzeugt ein ZUGFeRD 2.3 CII XML analog dazu.
6. **XSD-Validierung** – Das fertige XML wird gegen das offizielle Schema geprüft. Fehler werden geloggt, und der Service versucht bekannte Probleme automatisch zu korrigieren.
7. **Speicherung und Rückgabe** – Die verarbeitete Rechnung wird in der Datenbank gespeichert. Das XML wird als HTTP-Response an den Browser zurückgeschickt und steht zum Download bereit.

Der gesamte Weg einer Rechnung durch das System lässt sich also zusammenfassen als: *Datei → Text → KI-JSON → C#-Objekt → XML → Validierung → Datenbank → Download.*

5. Technologie-Stack und Begründung

Für ein Projekt dieser Art gibt es viele mögliche Technologiekombinationen. Dieses Kapitel erklärt, welche Tools und Frameworks konkret eingesetzt wurden und warum diese Wahl sinnvoll war – sowohl aus technischer Sicht als auch hinsichtlich des vorhandenen Vorwissens im Team.

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

Das Backend wurde mit ASP.NET Core und C# umgesetzt. Der Hauptgrund dafür war, dass wir ASP.NET aus dem Unterricht bereits kannten und wussten, wie man damit eine REST-API aufbaut. Außerdem ist das Framework sehr gut dokumentiert und bietet mit dem eingebauten Dependency-Injection-System eine saubere Möglichkeit, Services zu trennen und testbar zu halten.

Swagger (via Swashbuckle) ist automatisch aktiv, wenn das Backend im Development-Modus läuft. Das macht es einfach, die API-Endpunkte direkt im Browser auszuprobieren, ohne extra ein Tool wie Postman zu brauchen. Die `.http`-Datei (`BillCoverterService.http`) im Projekt erlaubt das schnelle Testen direkt aus Visual Studio heraus.

Controllers wie `InvoiceController` und `ZugferdController` sind als `ApiController` mit Routing-Attributen annotiert und liefern standardmäßig JSON-Antworten zurück. Für Fehler wird `ProblemDetails` nach RFC 9457 genutzt.

5.2. Datenhaltung: PostgreSQL (Docker), EF Core

Als Datenbank wurde PostgreSQL gewählt, weil es kostenlos, stabil und weit verbreitet ist. Im Projekt läuft PostgreSQL als Docker-Container, was bedeutet, dass kein manuelles Installieren nötig ist – ein `docker-compose up` reicht.

Der Datenbankzugriff erfolgt über Entity Framework Core mit dem Code-First-Ansatz. Das bedeutet, die Datenbankstruktur wird aus den C#-Modellen abgeleitet. Sobald sich ein Modell ändert, erstellt man eine neue Migration mit `dotnet ef migrations add` und wendet sie mit `dotnet ef database update` an.

Die Migrationshistorie liegt im Ordner `Migrations/` und ist versioniert, sodass jede Änderung am Schema nachvollziehbar bleibt. Der `ApplicationDbContext` definiert die Tabellen und wird per Dependency Injection in die Repositories injiziert.

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

Das Frontend ist eine Angular-SPA (*Single Page Application*) im Ordner `smart-bill-ui/`. Es kommuniziert ausschließlich über HTTP mit der Backend-API und hat keine eigene Logik zur Rechnungsverarbeitung.

Angular wurde gewählt, weil der Frontend-Entwickler Luis Schörgendorfer damit gearbeitet hat und das Framework sich gut für formularbasierte Webanwendungen eignet. TypeScript sorgt für Typsicherheit, was Fehler durch falsch formatierte API-Anfragen reduziert.

Das Frontend wird ebenfalls als Docker-Container gebaut und über einen Nginx-Webserver ausgeliefert. Die `nginx.conf` ist so konfiguriert, dass alle Routen ans Angular-Routing weitergeleitet werden.

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

Neben dem Framework selbst kommen mehrere spezialisierte Bibliotheken zum Einsatz:

- **PdfPig** – open-source Bibliothek zum Lesen von PDF-Dateien in C#. Sie extrahiert den Textinhalt direkt aus dem PDF-Strukturbaum, ohne einen externen Prozess starten zu müssen. Das war der Hauptvorteil gegenüber iTextSharp, das lizenzrechtliche Einschränkungen hat.
- **Tesseract OCR** – Engine zur Texterkennung in Bildern, basiert auf einem LSTM-Netzwerk. Wird über den NuGet-Wrapper `Tesseract` angebunden. Sprachmodelle (Deutsch, Englisch) müssen separat heruntergeladen und im `tessdata/`-Ordner abgelegt werden. Dafür gibt es die PowerShell-Skripte `download-tessdata.ps1` und `download-tesseract-models.ps1`.
- **XmlSerializer** – Teil von `System.Xml.Serialization` aus .NET. Wird verwendet, um die C#-Modelle direkt in XML zu serialisieren. Über Attribute wie `[XmlElement]`, `[XmlAttribute]` und `[XmlNamespaceDeclarations]` lässt sich die Ausgabe exakt steuern.
- **XSD-Validierung** – Die offiziellen Schemadateien für ebInterface 6.1 und ZUGFeRD 2.3 CII liegen im Ordner `Doc/`. Das generierte XML wird mit `XmlSchemaSet` und `XmlReaderSettings` gegen das jeweilige Schema geprüft, bevor es zurückgegeben wird.

5.5. Entwicklungsumgebung: Docker, Skripte, Logging, Konfiguration

Das gesamte System lässt sich mit einer einzigen `docker-compose.yml` starten, die im Root-Ordner `SmartBillConverter/` liegt. Sie definiert drei Services: das Backend, das Frontend und PostgreSQL. Für die lokale Entwicklung ist ein separates `appsettings.Development.json` vorhanden, das die Datenbankverbindung und API-Keys überschreibt.

Sensible Werte wie der Gemini-API-Key werden über Umgebungsvariablen übergeben und nicht in der Versionskontrolle gespeichert. Im `appsettings.json` sind nur Platzhalter-Werte hinterlegt.

Für das Einrichten von Tesseract auf einem frischen System gibt es mehrere PowerShell-Skripte im Backend-Ordner, die das Herunterladen der Modelle und das Kopieren in den richtigen Pfad automatisieren.

Das Logging nutzt die standardmäßige ASP.NET-Core-Logging-Infrastruktur mit `ILogger<T>`. Jeder Service, der kritische Schritte ausführt (z. B. LLM-Aufruf, XML-Validierung), schreibt bei Fehlern eine geloggerte Warnung oder einen Error-Eintrag. So lassen sich Probleme im Betrieb leichter nachvollziehen.

Teil III.

Implementierung (Backend)

6. Backend-Anwendungsstruktur

Das Backend des *SmartBillConverter* ist eine RESTful Web API auf Basis von ASP.NET Core 9.0. Der Aufbau folgt einem klassischen Schichtenmodell: Controller nehmen HTTP-Anfragen entgegen, Services enthalten die eigentliche Logik, und Repositories kümmern sich um den Datenbankzugriff. Dieses Kapitel zeigt, wie die einzelnen Teile technisch umgesetzt wurden.

6.1. Controller

Die Controller sind das Erste, was eine eingehende HTTP-Anfrage sieht. Sie prüfen die Anfrage, leiten die eigentliche Arbeit an die zuständigen Services weiter und schicken das Ergebnis zurück ans Frontend. Im Projekt gibt es fünf Controller, jeder für einen eigenen Aufgabenbereich.

6.1.1. ProcessingController: Upload und Verarbeitungs-Orchestrierung

Der `ProcessingController` ist der wichtigste Controller im Projekt. Er steuert den gesamten Ablauf von der hochgeladenen PDF-Datei bis zur fertigen XML.

Struktur und Dependency Injection

Der Controller nutzt Constructor Injection für alle Abhängigkeiten:

Listing 4: ProcessingController Constructor

```
1  [ApiController]
2  [Route('api/[controller]')]
3  public class ProcessingController : ControllerBase
4  {
5      private readonly IInvoiceService _invoiceService;
6      private readonly ILlmConversionService _llmConversionService;
7      private readonly ILogger<ProcessingController> _logger;
```

```
8
9       public ProcessingController(
10          IInvoiceService invoiceService,
11          ILlmConversionService llmConversionService,
12          ILogger<ProcessingController> logger)
13      {
14          _invoiceService = invoiceService;
15          _llmConversionService = llmConversionService;
16          _logger = logger;
17      }
18  }
```

Die `[ApiController]`-Attribute aktiviert automatische Model-Validierung, Fehlerbehandlung und API-spezifische Routing-Konventionen. `ILogger` wird vom ASP.NET Core Framework bereitgestellt und ermöglicht strukturiertes Logging.[5]

Upload-Endpoint mit Validierung

Der Haupt-Endpoint `/api/Processing/upload` akzeptiert PDF-Dateien via Multipart-Form-Data:

Listing 5: Upload-Endpoint mit Validierung

```
1   [HttpPost('upload')]
2   [Consumes('multipart/form-data')]
3   [ProducesResponseType(StatusCodes.Status200OK)]
4   [ProducesResponseType(StatusCodes.Status400BadRequest)]
5   public async Task<IActionResult> UploadPdf(IFormFile file)
6   {
7       if (file == null || file.Length == 0)
8           return BadRequest(new { error = 'Keine Datei hochgeladen' });
9
10      if (!file.ContentType.Equals('application/pdf',
11          StringComparison.OrdinalIgnoreCase))
12          return BadRequest(new { error = 'Nur PDF-Dateien erlaubt' });
13
14      if (file.Length > 10 * 1024 * 1024) // 10MB Limit
15          return BadRequest(new { error = 'Datei zu gross (max. 10MB)' });
16
17      using var stream = file.OpenReadStream();
18      var invoice = await _invoiceService.ProcessInvoiceAsync(
19          stream, file.FileName);
20
21      return Ok(new { success = true, invoice });
22  }
```

Die Prüfung läuft in drei Schritten: Erst wird geschaut ob eine Datei vorhanden ist, dann ob es wirklich ein PDF ist, und zuletzt ob die Datei nicht größer als 10 MB ist. Das Limit ist wichtig damit der Server bei sehr großen Dateien nicht zu viel Speicher braucht. `IFormFile` ist die ASP.NET-Klasse für hochgeladene Dateien.

Die eigentliche Arbeit macht `ProcessInvoiceAsync` im `IInvoiceService`: PDF-Text extrahieren, KI aufrufen, Daten speichern. Der Controller selbst macht nur das Nötigste - die HTTP-Anfrage prüfen und das Ergebnis zurückschicken.

[5] Vgl. Microsoft: *Dependency injection in ASP.NET Core*, https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection, letzter Zugriff am 06.01.2026

6.1.2. InvoiceController: CRUD-Operationen

Der `InvoiceController` bietet RESTful CRUD-Endpoints für gespeicherte Rechnungen:

Listing 6: InvoiceController GET-Endpoints

```
[ApiController]
[Route('api/[controller]')]
public class InvoiceController : ControllerBase
{
    private readonly IInvoiceRepository _repository;

    [HttpGet]
    public async Task<IActionResult> GetAll()
    {
        var invoices = await _repository.GetAllAsync();
        return Ok(invoices);
    }

    [HttpGet('{id}')]
    public async Task<IActionResult> GetById(int id)
    {
        var invoice = await _repository.GetByIdAsync(id);
        if (invoice == null)
            return NotFound(new { error = $'Rechnung {id} nicht gefunden' });
        return Ok(invoice);
    }

    [HttpGet('search')]
    public async Task<IActionResult> Search(
        [FromQuery] string? invoiceNumber,
        [FromQuery] int? year)
    {
        var results = await _repository.SearchAsync(invoiceNumber, year);
        return Ok(results);
    }
}
```

Das `[FromQuery]`-Attribut liest URL-Parameter automatisch aus. Beispiel: `GET /api/Invoice/search?year=2024` wird zu `Search(null, 2024)`. Das Repository-Pattern verbirgt die Datenbankzugriffe hinter einer Schnittstelle, was das Testen mit Mock-Objekten einfach macht.[6]

6.1.3. ZugferdController: Hybrides PDF/XML-Format

Der `ZugferdController` verarbeitet PDFs speziell für das ZUGFeRD-Format:

Listing 7: ZUGFeRD Upload-Endpoint

```
[ApiController]
[Route('api/[controller]')]
public class ZugferdController : ControllerBase
{
    private readonly IZugferdService _zugferdService;

    [HttpPost('upload-pdf')]
    [Consumes('multipart/form-data')]
    public async Task<IActionResult> UploadPdfForZugferd(IFormFile file)
    {
```

[6]Vgl. Fowler, Martin: *Patterns of Enterprise Application Architecture - Repository*, https://martinfowler.com/eaaCatalog/repository.html, letzter Zugriff am 06.01.2026

```
11          if (file == null || file.Length == 0)
12              return BadRequest('Keine Datei');
13
14          using var stream = file.OpenReadStream();
15          var zugferdXml = await _zugferdService.ConvertToZugferdAsync(
16              stream, file.FileName);
17
18          return Ok(new {
19              success = true,
20              zugferdXml,
21              format = 'ZUGFeRD 2.3 EN16931'
22          });
23      }
24  }
```

Der `ZugferdService` liest den Text aus dem PDF, schickt ihn an die KI und bekommt ZUGFeRD-konformes CII-XML zurück. Das XML wird direkt als String in der Antwort mitgeschickt, damit das Frontend es als Download anbieten kann.

6.1.4. ImageProcessingController: OCR für Scans

Für gescannte Rechnungen (PNG, JPEG, TIFF) bietet der `ImageProcessingController` OCR-Verarbeitung:

Listing 8: Image-OCR-Endpoint

```
1   [HttpPost('ocr')]
2   [Consumes('multipart/form-data')]
3   public async Task<IActionResult> ProcessImage(IFormFile imageFile)
4   {
5       var supportedFormats = new[] { '.png', '.jpg', '.jpeg', '.bmp', '.tiff' };
6       var extension = Path.GetExtension(imageFile.FileName).ToLower();
7
8       if (!supportedFormats.Contains(extension))
9           return BadRequest($'Format {extension} nicht unterstuetzt');
10
11      var extractedText = await _ocrService.ExtractTextFromImageAsync(imageFile);
12      var llmResponse = await _llmService.ConvertToXmlAsync(extractedText);
13
14      return Ok(new {
15          extractedText,
16          ebInterfaceXml = llmResponse.EbInterfaceXml
17      });
18  }
```

Die Prüfung läuft über die Dateiendung. Tesseract kann alle gängigen Bildformate verarbeiten, aber die Erkennungsqualität hängt stark von der Auflösung ab (optimal sind 300 DPI). Der zweistufige Ablauf OCR → KI hilft dabei, OCR-Fehler im zweiten Schritt noch zu korrigieren.

6.1.5. DownloadController: XML-Datei-Export

Der `DownloadController` liefert generierte XML-Dateien mit korrekten HTTP-Headers:

Listing 9: XML-Download mit Content-Disposition

```
[HttpGet('xml/{id}')]
public async Task<IActionResult> DownloadXml(int id)
{
    var invoice = await _repository.GetByIdAsync(id);
    if (invoice?.EbInterfaceXml == null)
        return NotFound();

    var xmlBytes = Encoding.UTF8.GetBytes(invoice.EbInterfaceXml);
    var fileName = $'invoice_{invoice.InvoiceNumber}_ebInterface.xml';

    return File(xmlBytes, 'application/xml', fileName);
}
```

Die `File()`-Methode setzt automatisch den `Content-Disposition: attachment`-Header, wodurch der Browser den Download-Dialog anzeigt. Der Dateiname folgt einem sprechenden Schema für einfache Archivierung.

6.2. Services

Services enthalten die eigentliche Programmlogik. Jeder Service wird als Scoped registriert, das heißt pro HTTP-Request entsteht eine eigene Instanz und wird danach wieder aufgeräumt.

6.2.1. Interface-basierte Abstraktion

Alle Services sind über Interfaces definiert, was Dependency Inversion ermöglicht:

Listing 10: Service-Interfaces

```
public interface IInvoiceService
{
    Task<Invoice> ProcessInvoiceAsync(Stream pdfStream, string fileName);
    Task<Invoice> SaveInvoiceAsync(Invoice invoice);
}

public interface IPdfExtractionService
{
    Task<string> ExtractTextFromPdfAsync(Stream pdfStream);
}

public interface IOcrExtractionService
{
    Task<string> ExtractTextFromImageAsync(IFormFile imageFile);
    Task<string> ExtractTextFromImageAsync(Stream stream, string fileName);
}

public interface ILlmConversionService
{
    Task<LlmResponse> ConvertToXmlAsync(string extractedText);
}
```

Weil alle Services über Interfaces angesprochen werden, kann man die konkrete Implementierung jederzeit austauschen - zum Beispiel von Gemini auf GPT-4 wechseln - ohne

den Controller ändern zu müssen. Das ist auch für Tests nützlich, weil man Interfaces durch einfache Mock-Objekte ersetzen kann.

6.2.2. Service-Registrierung in Program.cs

Die Services werden im Dependency Injection Container registriert:

Listing 11: Service-Registrierung

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<IInvoiceRepository, InvoiceRepository>();
builder.Services.AddScoped<IPdfExtractionService, PdfExtractionService>();
builder.Services.AddScoped<IOcrExtractionService, OcrExtractionService>();
builder.Services.AddScoped<ILlmConversionService, LlmConversionService>();
builder.Services.AddScoped<IInvoiceService, InvoiceService>();

builder.Services.AddHttpClient<ILlmConversionService, LlmConversionService>()
    .SetHandlerLifetime(TimeSpan.FromMinutes(5));
```

`AddScoped` heißt: Pro HTTP-Request wird eine neue Instanz erstellt und am Ende wieder gelöscht. `AddHttpClient` meldet den `HttpClient` für den `LlmConversionService` an, was automatisch Connection-Pooling und Timeout-Verwaltung mitbringt.[7]

6.2.3. InvoiceService: Orchestrierung der Pipeline

Der `InvoiceService` koordiniert alle Verarbeitungsschritte:

Listing 12: InvoiceService Pipeline

```
public class InvoiceService : IInvoiceService
{
    public async Task<Invoice> ProcessInvoiceAsync(
        Stream pdfStream, string fileName)
    {
        var extractedText = await _pdfService.ExtractTextFromPdfAsync(pdfStream);
        _logger.LogInformation('Text extrahiert: {Length} Zeichen',
            extractedText.Length);

        var llmResponse = await _llmService.ConvertToXmlAsync(extractedText);

        var invoice = MapToInvoice(llmResponse.ParsedData);
        invoice.ExtractedText = extractedText;
        invoice.EbInterfaceXml = llmResponse.FinalEbInterfaceXml;

        await _repository.AddAsync(invoice);
        await _repository.SaveChangesAsync();
        return invoice;
    }
}
```

Der Ablauf ist gradlinig: PDF rein, Text extrahieren, KI aufrufen, in der Datenbank speichern. Nach jedem Schritt wird geloggt, damit man beim Debuggen nachvollziehen kann, wo etwas schiefgelaufen ist.

[7] Vgl. Microsoft: *Make HTTP requests using IHttpClientFactory*, https://learn.microsoft.com/en-us/dotnet/core/extensions/httpclient-factory, letzter Zugriff am 06.01.2026

6.3. Datenzugriff

Das Repository-Pattern abstrahiert die Datenbankzugriffe und bietet eine Collection-ähnliche Schnittstelle.

6.3.1. IInvoiceRepository Interface

Listing 13: IInvoiceRepository Definition

```
1  public interface IInvoiceRepository
2  {
3      Task<Invoice?> GetByIdAsync(int id);
4      Task<List<Invoice>> GetAllAsync();
5      Task<List<Invoice>> SearchAsync(string? invoiceNumber, int? year);
6      Task AddAsync(Invoice invoice);
7      Task UpdateAsync(Invoice invoice);
8      Task DeleteAsync(int id);
9      Task<int> SaveChangesAsync();
10 }
```

6.3.2. Repository-Implementierung mit Entity Framework

Listing 14: InvoiceRepository Implementierung

```
1  public class InvoiceRepository : IInvoiceRepository
2  {
3      private readonly ApplicationDbContext _context;
4
5      public async Task<List<Invoice>> SearchAsync(
6          string? invoiceNumber, int? year)
7      {
8          var query = _context.Invoices.AsQueryable();
9
10         if (!string.IsNullOrEmpty(invoiceNumber))
11             query = query.Where(i => i.InvoiceNumber.Contains(invoiceNumber));
12
13         if (year.HasValue)
14             query = query.Where(i => i.Year == year.Value);
15
16         return await query.ToListAsync();
17     }
18 }
```

`AsQueryable()` baut die SQL-Abfrage erst dann auf, wenn tatsächlich auf die Datenbank zugegriffen wird. `Contains()` wird zu SQL `LIKE` übersetzt, sodass Teilbegriffe gefunden werden.

6.3.3. ApplicationDbContext: Entity Framework Configuration

Listing 15: ApplicationDbContext

```
1  public class ApplicationDbContext : DbContext
2  {
3      public DbSet<Invoice> Invoices { get; set; }
4
```

```
5       protected override void OnModelCreating(ModelBuilder modelBuilder)
6       {
7           modelBuilder.Entity<Invoice>(entity =>
8           {
9               entity.HasKey(e => e.Id);
10              entity.Property(e => e.InvoiceAmount).HasPrecision(14, 2);
11              entity.Property(e => e.DiscountAmount).HasPrecision(14, 2);
12              entity.HasIndex(e => e.InvoiceNumber);
13              entity.HasIndex(e => e.Year);
14          });
15      }
16  }
```

`HasPrecision(14, 2)` legt fest, dass Geldbeträge mit 14 Stellen und 2 Nachkommastellen gespeichert werden. Die Indizes auf `InvoiceNumber` und `Year` machen spätere Suchanfragen deutlich schneller.[8]

6.3.4. Invoice Entity-Modell

Listing 16: Invoice Entity

```
1   public class Invoice
2   {
3       [Key]
4       public int Id { get; set; }
5       public short Year { get; set; }
6       [MaxLength(20)]
7       public string? InvoiceNumber { get; set; }
8       [Column(TypeName = 'decimal(14,2)')]
9       public decimal InvoiceAmount { get; set; }
10      public DateTime InvoiceDate { get; set; }
11      public DateTime DueDate { get; set; }
12      [MaxLength(20)]
13      public string? VatNumber { get; set; }
14      [MaxLength(34)]
15      public string? IBAN { get; set; }
16      [NotMapped]
17      public string? ExtractedText { get; set; }
18      [NotMapped]
19      public string? EbInterfaceXml { get; set; }
20  }
```

Properties mit `[NotMapped]` werden nicht in der Datenbank angelegt - sie sind nur im C#-Objekt vorhanden, damit sie in der API-Antwort mitgeschickt werden können.

6.4. Konfiguration und Secrets Management

6.4.1. appsettings.json

Listing 17: appsettings.json

```
1   {
2     'ConnectionStrings': {
3       'DefaultConnection': 'Host=localhost;Port=5432;Database=smartbill;...'
```

[8] Vgl. Microsoft: *Entity Framework Core Documentation*, https://learn.microsoft.com/en-us/ef/core/, letzter Zugriff am 06.01.2026

```
  },
  'Tesseract': { 'DataPath': './tessdata' },
  'Logging': {
    'LogLevel': { 'Default': 'Information',
                  'Microsoft.EntityFrameworkCore': 'Warning' }
  }
}
```

6.4.2. Environment Variables und .env-Datei

Sensible Daten (API-Keys, Passwörter) kommen aus Environment Variables, nicht aus der Konfigurationsdatei:

Listing 18: .env-Datei laden

```
if (builder.Environment.IsDevelopment())
{
    var envPath = Path.Combine(Directory.GetCurrentDirectory(), '.env');
    if (File.Exists(envPath))
    {
        foreach (var line in File.ReadAllLines(envPath))
        {
            if (string.IsNullOrWhiteSpace(line) || line.StartsWith('#')) continue;
            var parts = line.Split('=', 2);
            if (parts.Length == 2)
                Environment.SetEnvironmentVariable(parts[0].Trim(), parts[1].Trim()
                    );
        }
    }
}
```

Die `.env`-Datei ist in `.gitignore` und wird nicht ins Repository eingecheckt. Beispieleintrag: `GEMINI_API_KEY=AIzaSy...`. In Produktion werden echte Umgebungsvariablen des Servers verwendet.[9]

6.4.3. Zugriff in Services

Listing 19: API Key laden

```
_apiKey = Environment.GetEnvironmentVariable('GEMINI_API_KEY')
    ?? throw new InvalidOperationException('GEMINI_API_KEY fehlt');
```

Der `?? throw`-Ausdruck lässt die Anwendung sofort abstürzen wenn der Key fehlt, bevor sie überhaupt eine Anfrage bearbeitet. So merkt man den Fehler bei der Inbetriebnahme und nicht erst zur Laufzeit.

[9]Vgl. The Twelve-Factor App: *III. Config*, https://12factor.net/config, letzter Zugriff am 06.01.2026

6.5. Fehlerbehandlung und Resilienzstrategie

6.5.1. Try-Catch in Controllern

Listing 20: Controller-Fehlerbehandlung

```
[HttpPost('upload')]
public async Task<IActionResult> UploadPdf(IFormFile file)
{
    try
    {
        if (file == null || file.Length == 0)
            return BadRequest(new { error = 'Keine Datei' });

        using var stream = file.OpenReadStream();
        var invoice = await _invoiceService.ProcessInvoiceAsync(stream, file.
            FileName);
        return Ok(new { success = true, invoice });
    }
    catch (InvalidOperationException ex)
    {
        _logger.LogError(ex, 'Verarbeitungsfehler bei {FileName}', file?.FileName);
        return StatusCode(500, new { error = 'Verarbeitung fehlgeschlagen',
                                     details = ex.Message });
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, 'Unerwarteter Fehler');
        return StatusCode(500, new { error = 'Interner Serverfehler' });
    }
}
```

Bekannte Fehler wie `InvalidOperationException` werden zuerst abgefangen und dem Benutzer mit einer verständlichen Nachricht angezeigt. Der zweite `catch`-Block fängt alle anderen unerwarteten Fehler. HTTP-Status 500 bedeutet, dass der Server selbst einen Fehler hatte.

6.5.2. Strukturiertes Logging mit ILogger

Die geschweiften Klammern im Log-Aufruf sind keine normalen Strings, sondern Platzhalter die beim Log-Aggregieren durchsuchbar bleiben:

Listing 21: Strukturiertes Logging

```
_logger.LogInformation('PDF geoeffnet: {PageCount} Seiten', document.NumberOfPages
    );
_logger.LogInformation('Extraktion: {CharCount} Zeichen', extractedText.Length);
_logger.LogError(ex, 'PDF-Extraktion fehlgeschlagen');
```

`LogError` schreibt automatisch auch den Stack-Trace mit, was bei der Fehlersuche sehr nützlich ist.[10]

[10] Vgl. Microsoft: *Logging in .NET Core and ASP.NET Core*, https://learn.microsoft.com/en-us/aspnet/core/fundamentals/logging/, letzter Zugriff am 06.01.2026

6.5.3. HTTP-Statuscodes

- **200 OK**: Verarbeitung erfolgreich, Daten in der Antwort
- **400 Bad Request**: Eingabefehler auf Client-Seite (fehlende Datei, falsches Format)
- **404 Not Found**: Angefragte Ressource existiert nicht
- **500 Internal Server Error**: Fehler auf Serverseite (z. B. PDF-Extraktion gescheitert)

6.5.4. CORS-Konfiguration

Angular läuft auf Port 4200, die API auf Port 5000. Damit das Frontend Anfragen ans Backend schicken darf, muss CORS konfiguriert werden:

Listing 22: CORS-Policy

```
builder.Services.AddCors(options =>
{
    options.AddPolicy('AllowFrontend', policy =>
        policy.AllowAnyOrigin().AllowAnyMethod().AllowAnyHeader());
});
app.UseCors('AllowFrontend');
```

`AllowAnyOrigin()` ist für die Entwicklung in Ordnung. Im echten Betrieb sollte die Origin auf die konkrete Frontend-URL eingeschränkt werden.[11]

[11] Vgl. Mozilla: *Cross-Origin Resource Sharing (CORS)*, https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS, letzter Zugriff am 06.01.2026

7. Pipeline zur PDF-Textextraktion

Bevor eine Rechnung verarbeitet werden kann, muss ihr Text ausgelesen werden. Bei digital erstellten PDFs ist das relativ einfach - der Text ist schon als Daten drin. Schwieriger wird es bei eingescannten Dokumenten, wo nur Bilder vorliegen. Dieses Kapitel beschreibt, wie digitale PDFs im *SmartBillConverter* verarbeitet werden.

7.1. PdfPig-Integration

Für die Textextraktion aus digitalen PDFs wird die Open-Source-Bibliothek *PdfPig* eingesetzt. Sie ist in C# geschrieben und bietet direkten Zugriff auf die interne Struktur von PDF-Dateien.

7.1.1. PdfExtractionService: Architektur und Interface

Das Interface des Services ist bewusst einfach gehalten:

Listing 23: IPdfExtractionService Interface

```
1  public interface IPdfExtractionService
2  {
3      Task<string> ExtractTextFromPdfAsync(Stream pdfStream);
4  }
```

Als Parameter nimmt das Interface einen `Stream` statt einem Dateipfad. Das ist praktischer, weil der Stream sowohl von einer hochgeladenen Datei als auch aus einem Speicherpuffer kommen kann. Async ist die Methode, weil das Lesen von Streams eine I/O-Operation ist, die den Server nicht blockieren soll.

7.1.2. Stream-Handling und Document-Opening

Listing 24: PDF Stream-Verarbeitung

```
public async Task<string> ExtractTextFromPdfAsync(Stream pdfStream)
{
    try
    {
        if (pdfStream.CanSeek) pdfStream.Position = 0;

        var extractedText = new StringBuilder();

        using (var document = PdfDocument.Open(pdfStream))
        {
            _logger.LogInformation('PDF geoeffnet - {PageCount} Seiten',
                document.NumberOfPages);
            // Seiten-Iteration folgt...
        }
        return await Task.FromResult(extractedText.ToString());
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, 'Kritischer Fehler bei der PDF-Extraktion');
        return await Task.FromResult(GenerateFallbackDemoText());
    }
}
```

Das Zurücksetzen der Stream-Position ist wichtig: Hat der Controller den Stream vorher schon für eine Prüfung gelesen, steht der Lesezeiger am Ende. Ohne Reset sieht PdfPig eine leere Datei. `CanSeek` prüft vorher ob das überhaupt möglich ist.

Mit `using` wird das `PdfDocument` nach der Verarbeitung automatisch aus dem Speicher entfernt. Da PdfPig das ganze Dokument lädt, ist das 10 MB-Limit im Controller wichtig.[12]

7.1.3. Seitenweise Extraktion mit Fehler-Isolation

PDFs können einzelne kaputte Seiten enthalten. Die Implementierung verarbeitet jede Seite in einem eigenen Try-Catch-Block:

Listing 25: Robuste Seiten-Iteration

```
for (int pageNumber = 1; pageNumber <= document.NumberOfPages; pageNumber++)
{
    extractedText.AppendLine($'--- Seite {pageNumber} ---');
    try
    {
        var page = document.GetPage(pageNumber);
        var pageText = ExtractTextFromPage(page);

        if (!string.IsNullOrWhiteSpace(pageText))
            extractedText.AppendLine(pageText);
        else
            extractedText.AppendLine('[Keine lesbaren Textinhalte auf dieser Seite]
                ');
    }
    catch (Exception pageEx)
    {
        extractedText.AppendLine($'[Fehler Seite {pageNumber}: {pageEx.Message}]');
    }
    extractedText.AppendLine();
}
```

[12] Vgl. UglyToad: *PdfPig Documentation - Opening Documents*, https://github.com/UglyToad/PdfPig/wiki/Opening-documents, letzter Zugriff am 06.01.2026

Wenn Seite 2 defekt ist, werden Seite 1 und 3 trotzdem extrahiert. Die Fehlermeldung wird direkt in den Text geschrieben, sodass das KI-Modell im nächsten Schritt sieht, dass Seite 2 fehlt.

7.1.4. Geometrische Wortsortierung

PdfPig gibt einzelne Wörter mit ihren Koordinaten zurück - keine Zeilen. Die richtige Lesereihenfolge muss erst rekonstruiert werden:

Listing 26: Geometrische Wort-Sortierung

```
private string ExtractTextFromPage(Page page)
{
    var words = page.GetWords();
    if (words == null || !words.Any()) return '';

    var sortedWords = words
        .OrderBy(w => Math.Round(w.BoundingBox.Bottom, 1))
        .ThenBy(w => Math.Round(w.BoundingBox.Left, 1))
        .ToList();

    var extractedText = new StringBuilder();
    var currentLine = new StringBuilder();
    double? currentLineY = null;
    const double lineHeightTolerance = 5.0;

    foreach (var word in sortedWords)
    {
        var wordY = Math.Round(word.BoundingBox.Bottom, 1);
        if (currentLineY.HasValue &&
            Math.Abs(wordY - currentLineY.Value) > lineHeightTolerance)
        {
            if (currentLine.Length > 0)
            {
                extractedText.AppendLine(currentLine.ToString().Trim());
                currentLine.Clear();
            }
        }
        if (currentLine.Length > 0) currentLine.Append(' ');
        currentLine.Append(word.Text);
        currentLineY = wordY;
    }
    if (currentLine.Length > 0)
        extractedText.AppendLine(currentLine.ToString().Trim());

    return CleanExtractedText(extractedText.ToString().Trim());
}
```

Sortiert wird zuerst nach Y-Koordinate (Zeile von oben nach unten), dann nach X (Wort von links nach rechts). `BoundingBox.Bottom` ist im PDF-Koordinatensystem die untere Wortkante. Die Toleranz von 5 Punkten (rund 1,8 mm) verträgt kleine vertikale Verschiebungen innerhalb einer Zeile.[13]

[13] Vgl. PDF Reference 1.7: *Coordinate Systems*, Adobe Systems, 2006, S. 117-120

7.2. Textbereinigung und Normalisierung

Der Rohtext direkt aus dem PDF enthält oft störende Artefakte: mehrfache Leerzeichen, zu viele Leerzeilen, Whitespace am Anfang jeder Zeile.

Listing 27: Text-Bereinigung mit Regex

```
private string CleanExtractedText(string text)
{
    if (string.IsNullOrEmpty(text)) return text;
    try
    {
        text = Regex.Replace(text, @' +', ' ');
        text = Regex.Replace(text, @'\n{3,}', '\n\n');
        var lines = text.Split('\n')
            .Select(line => line.Trim())
            .Where(line => !string.IsNullOrEmpty(line));
        return string.Join('\n', lines);
    }
    catch (Exception ex)
    {
        _logger.LogWarning(ex, 'Fehler beim Bereinigen');
        return text;
    }
}
```

`@" +"` matched ein oder mehr Leerzeichen und ersetzt sie durch ein einzelnes. `@"\n{3,}"` kürzt drei oder mehr Leerzeilen auf maximal zwei. Die `Trim()`-Kette entfernt führende und nachfolgende Leerzeichen pro Zeile.

7.3. Fehlerbehandlung und Fallback-Strategien

7.3.1. Fallback bei leerem Text

Manche PDFs enthalten trotz vorhandenem Dateiformat keinen lesbaren Text (z. B. eingescannte PDFs ohne Text-Overlay):

Listing 28: Leertext-Erkennung

```
var result = extractedText.ToString();

if (string.IsNullOrWhiteSpace(result) ||
    result.Contains('[Keine lesbaren Textinhalte') ||
    result.Replace('---', '').Replace('Seite', '').Trim().Length < 50)
{
    _logger.LogWarning('Wenig Text extrahiert, Fallback aktiv');
    result += '\n\n' + GenerateFallbackDemoText();
}
```

Die Prüfung schaut auf drei Dinge: komplett leer, nur Fehlermeldungen, oder nach dem Herausrechnen der Seitentrennzeilen weniger als 50 Zeichen. Trifft eines zu, wird ein Demo-Text angehängt damit das KI-Modell trotzdem etwas verarbeiten kann.

7.3.2. Entscheidungslogik: PDF vs. OCR

Im `InvoiceService` wird anhand der Dateiendung entschieden welche Methode zum Einsatz kommt:

Listing 29: PDF-oder-OCR-Entscheidung

```
if (fileName.EndsWith('.pdf', StringComparison.OrdinalIgnoreCase))
{
    extractedText = await _pdfService.ExtractTextFromPdfAsync(pdfStream);

    if (string.IsNullOrWhiteSpace(extractedText) || extractedText.Length < 100)
        _logger.LogWarning('PDF wahrscheinlich gescannt, wenig Text gefunden');
}
else
{
    extractedText = await _ocrService.ExtractTextFromImageAsync(pdfStream, fileName
        );
}
```

Ein häufiges Problem ist das *gescannte PDF*: eine Datei mit `.pdf`-Endung, die aber nur ein Scan-Bild enthält ohne echten Text. Die Heuristik `Length < 100` erkennt diesen Fall und könnte einen OCR-Fallback auslösen.

7.4. Bekannte Sonderfälle

Die PdfPig-Integration funktioniert gut für typische einspaltige Rechnungen. Es gibt aber Situationen, wo Probleme entstehen.

7.4.1. Gescannte PDFs

Wenn ein PDF kein Text-Overlay hat, gibt PdfPig eine leere Liste zurück. Der Benutzer bekommt keine direkte Fehlermeldung, sondern das Fallback-System greift ein. Im echten Produktionseinsatz wäre hier ein automatischer OCR-Fallback sinnvoll:

Listing 30: OCR-Fallback fuer gescannte PDFs (Konzept)

```
if (extractedText.Length < 100)
{
    pdfStream.Position = 0;
    var images = await ConvertPdfToImages(pdfStream);
    var ocrTexts = new List<string>();
    foreach (var image in images)
        ocrTexts.Add(await _ocrService.ExtractTextFromImageAsync(
            image.Stream, $'page{image.PageNumber}.png'));
    extractedText = string.Join('\n--- Naechste Seite ---\n', ocrTexts);
}
```

`ConvertPdfToImages` würde jede PDF-Seite in ein Bild rendern (z. B. mit SkiaSharp). Das ist in der aktuellen Version noch nicht eingebaut.

7.4.2. Mehrspaltige Layouts

Das geometrische Sortierverfahren geht davon aus, dass der Text in einer einzigen Spalte von oben nach unten fließt. Bei zweispaltigen Rechnungen kommt es zu Problemen: Wörter der linken und rechten Spalte auf derselben Y-Höhe werden in der falschen Reihenfolge zusammengeführt.

Beispiel: Steht links „Rechnungsnummer:“ und rechts „Kundennummer:“ auf gleicher Höhe, entstehen unter Umständen Mischzeilen wie „Rechnungsnummer: Kundennummer:“. Das KI-Modell kann damit meistens noch umgehen, aber es erhöht das Risiko von Extraktionsfehlern.

Eine Lösung wäre, Spalten anhand von Lücken im X-Koordinatenraum zu erkennen und separat zu verarbeiten. Das ist im aktuellen Stand nicht umgesetzt.

7.4.3. Tabellarische Positionen

Rechnungspositionen werden in PDFs meist als Tabelle dargestellt. Die geometrische Sortierung rekonstruiert Tabellenzeilen korrekt, solange die Zeilen klar voneinander getrennt sind. Problematisch wird es bei:

- Sehr dicht beieinanderliegenden Zeilen (Zeilenabstand unter 5 Punkte)
- Beschreibungstexten die über mehrere Zeilen umbrechen
- Spalten ohne klare Trennlinie

In solchen Fällen kann der extrahierte Text unübersichtlich werden. Das KI-Modell im nächsten Schritt ist aber darauf trainiert, trotzdem die richtigen Zahlen herauszulesen - sofern alle Werte im Text vorhanden sind.[14]

[14] Vgl. UglyToad: *PdfPig - Page and Word Extraction*, https://github.com/UglyToad/PdfPig/wiki/Words, letzter Zugriff am 06.01.2026

8. OCR-Pipeline für Bilder

Wenn eine Rechnung als Scan oder Foto vorliegt, gibt es keinen digitalen Text den man einfach auslesen könnte. Hier muss OCR (Optical Character Recognition) den Text aus dem Bild erkennen. Im Projekt wird dafür *Tesseract* verwendet, die bekannteste Open-Source-OCR-Engine.

8.1. Einrichtung von Tesseract

8.1.1. OcrExtractionService: Setup und Abhängigkeiten

Tesseract braucht sogenannte *Traineddata*-Dateien - das sind die Sprachmodelle mit denen das Programm Buchstaben erkennt. Sie sind nicht im NuGet-Package enthalten und müssen separat heruntergeladen werden:

Listing 31: OcrExtractionService Constructor

```
public class OcrExtractionService : IOcrExtractionService
{
    private readonly string _tessDataPath;
    private static readonly string[] SupportedFormats =
        { '.png', '.jpg', '.jpeg', '.bmp', '.tiff', '.gif' };

    public OcrExtractionService(ILogger<OcrExtractionService> logger,
                                IConfiguration configuration)
    {
        _logger = logger;
        _tessDataPath = configuration.GetValue<string>('Tesseract:DataPath')
            ?? Path.Combine(Directory.GetCurrentDirectory(), 'tessdata');

        if (!Directory.Exists(_tessDataPath))
            _logger.LogWarning('Tesseract data path nicht gefunden: {Path}',
                _tessDataPath);
    }
}
```

`tessDataPath` zeigt auf den Ordner mit den Sprachdateien (z. B. `deu.traineddata`). Das Projekt enthält ein PowerShell-Script das sie automatisch herunterlädt:

Listing 32: download-tessdata.ps1 (Auszug)

```
$tessDataPath = '.\tessdata'
$baseUrl = 'https://github.com/tesseract-ocr/tessdata/raw/main'
```

```
3
4  if (-not (Test-Path $tessDataPath)) { New-Item -ItemType Directory -Path
       $tessDataPath }
5
6  Invoke-WebRequest -Uri '$baseUrl/deu.traineddata' -OutFile '$tessDataPath\deu.
       traineddata'
7  Invoke-WebRequest -Uri '$baseUrl/eng.traineddata' -OutFile '$tessDataPath\eng.
       traineddata'
8  Write-Host 'Tessdata download completed.'
```

Die Dateien sind nicht klein: `deu.traineddata` ist ca. 16 MB, `eng.traineddata` ca. 11 MB. In einem Docker-Container werden sie beim Build-Schritt heruntergeladen.[15]

8.1.2. Tesseract Engine-Initialisierung

Beim Start einer OCR-Verarbeitung wird eine `TesseractEngine`-Instanz erstellt:

Listing 33: Tesseract Engine Setup

```
1  var tempImagePath = Path.GetTempFileName() + Path.GetExtension(fileName);
2  using (var fs = new FileStream(tempImagePath, FileMode.Create))
3      await imageStream.CopyToAsync(fs);
4
5  try
6  {
7      using var engine = new TesseractEngine(_tessDataPath, 'deu+eng',
8                                              EngineMode.Default);
9      engine.SetVariable('tessedit_char_whitelist',
10         '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz' +
11         'aeoeuaeoeueAeOeUess.,:-+/()euro$GBP \n\r\t');
12     engine.SetVariable('preserve_interword_spaces', '1');
13     // Verarbeitung folgt...
14 }
15 finally { if (File.Exists(tempImagePath)) File.Delete(tempImagePath); }
```

Tesseract.NET braucht einen echten Dateipfad, daher muss der Stream zuerst in eine temporäre Datei geschrieben werden. Der `finally`-Block sorgt dafür, dass die Temp-Datei auch bei Fehlern gelöscht wird.

Was die Parameter bedeuten:

- `deu+eng`: Beide Sprachdateien gleichzeitig aktiv - das hilft bei Rechnungen mit gemischtem Deutsch/Englisch.
- `EngineMode.Default`: Nutzt das neuere LSTM-Netz aus Tesseract 4.0, das deutlich besser ist als die alte Methode.
- `tessedit_char_whitelist`: Schränkt die erlaubten Zeichen ein, damit Bildartefakte nicht als Sonderzeichen erkannt werden.

[15] Vgl. Tesseract OCR: *Data Files*, https://github.com/tesseract-ocr/tessdata, letzter Zugriff am 06.01.2026

- `preserve_interword_spaces`: Mehrfache Leerzeichen bleiben erhalten, was für Tabellenspalten wichtig ist.

8.2. OCR-Endpunkte und Integration

8.2.1. Anbindung an den ImageProcessingController

Der `ImageProcessingController` nimmt Bilddateien entgegen und leitet sie an den `OcrExtractionService` weiter:

Listing 34: OCR-Endpoint im Controller

```
[HttpPost('ocr')]
[Consumes('multipart/form-data')]
public async Task<IActionResult> ProcessImage(IFormFile imageFile)
{
    if (imageFile == null || imageFile.Length == 0)
        return BadRequest('Keine Datei');

    if (!_ocrService.IsImageFormatSupported(imageFile.FileName))
        return BadRequest($'Format nicht unterstuetzt: ' +
            Path.GetExtension(imageFile.FileName));

    try
    {
        var extractedText = await _ocrService.ExtractTextFromImageAsync(imageFile);
        var llmResponse  = await _llmService.ConvertToXmlAsync(extractedText);

        return Ok(new {
            success      = true,
            extractedText,
            ebInterfaceXml = llmResponse.EbInterfaceXml,
            zugferdXml     = llmResponse.ZugferdXml,
            validationOk   = llmResponse.IsValidXml
        });
    }
    catch (InvalidOperationException ex)
    {
        _logger.LogError(ex, 'OCR fehlgeschlagen fuer {File}', imageFile.FileName);
        return StatusCode(500, new { error = ex.Message });
    }
}
```

8.2.2. Formatprüfung und Validierung

Listing 35: Bildformat-Pruefung

```
public bool IsImageFormatSupported(string fileName)
{
    var extension = Path.GetExtension(fileName).ToLowerInvariant();
    return SupportedFormats.Contains(extension);
}
```

Die Prüfung läuft über die Dateiendung. Eine sicherere Variante würde die ersten Bytes der Datei lesen (Magic Bytes), aber für den Projekteinsatz ist die Endungsprüfung ausreichend.

8.2.3. Ablauf von Bild zu XML

Der vollständige Ablauf für eine gescannte Rechnung sieht so aus:

1. Frontend lädt Bilddatei via `POST /api/ImageProcessing/ocr` hoch
2. Controller prüft Dateiendung und Dateigröße
3. `OcrExtractionService` speichert das Bild temporär und führt Tesseract aus
4. Extrahierter Text wird an `LlmConversionService` weitergegeben
5. KI-Modell erzeugt JSON, das zu ebInterface und ZUGFeRD XML gemappt wird
6. Antwort mit Text und fertigen XMLs geht zurück ans Frontend

Der zweistufige Ablauf OCR → KI hat einen praktischen Vorteil: Tesseract macht manchmal kleine Fehler (z. B. „O“ statt „0“). Das Sprachmodell kennt den Kontext und kann solche Tippfehler oft selbst korrigieren.

8.3. Qualität, Sprachpakete und Nachbearbeitung

8.3.1. Confidence-Scoring

Tesseract gibt nach der Erkennung einen Konfidenzwert zurück, der zeigt wie sicher es bei der Erkennung war:

Listing 36: OCR-Verarbeitung mit Confidence

```
using var img  = Pix.LoadFromFile(tempImagePath);
using var page = engine.Process(img);

extractedText = page.GetText();
var confidence = page.GetMeanConfidence();

_logger.LogInformation('OCR fertig. Konfidenz: {Conf:F2}%, Zeichen: {Len}',
    confidence * 100, extractedText?.Length ?? 0);

if (confidence < 0.5)
    _logger.LogWarning('Niedrige Konfidenz -- Bildqualitaet pruefen');
```

Werte unter 0,5 bedeuten meistens dass das Bild zu unscharf, zu dunkel oder stark verzerrt war. Im aktuellen Stand wird das nur geloggt. In einer erweiterten Version könnte der Benutzer direkt eine Warnung bekommen.[16]

[16] Vgl. Tesseract Documentation: *Improving Quality*, https://tesseract-ocr.github.io/tessdoc/ImproveQuality.html, letzter Zugriff am 06.01.2026

8.3.2. Preprocessing-Optionen

Die Erkennungsqualität hängt stark von der Bildvorbereitung ab. Folgende Schritte sind dokumentiert und könnten in einer späteren Version eingebaut werden:

Deskewing

Tesseract hat eingebaute Schräglagekorrektur, kommt aber bei starker Neigung (über 5°) an seine Grenzen. Manuell geht das so:

Listing 37: Deskewing (nicht implementiert)

```
using var img = Pix.LoadFromFile(imagePath);
var angle = img.FindSkew();
if (Math.Abs(angle) > 0.1)
{
    using var deskewed = img.Rotate(angle);
    using var page = engine.Process(deskewed);
    extractedText = page.GetText();
}
```

Adaptive Binarisierung

Bei ungleichmäßiger Beleuchtung (Schatten, Farbstich) hilft Otsu-Binarisierung:

Listing 38: Adaptive Binarisierung (nicht implementiert)

```
using var binary = img.BinarizeOtsuAdaptiveThreshold(
    tileWidth: 300, tileHeight: 300, smoothing: 1, scoreFraction: 0.1f);
using var page = engine.Process(binary);
```

Otsu's Method berechnet pro Bildkachel einen eigenen Schwellwert statt einen globalen für das ganze Bild.

8.3.3. Häufige Fehler und deren Ursachen

- **TessdataNotFoundException**: Die Sprachmodell-Dateien fehlen. Ursache: `download-tessdata.ps1` wurde nicht ausgeführt oder der Pfad in `appsettings.json` stimmt nicht.
- **UnsupportedImageFormatException**: Leptonica (die interne Bildverarbeitungsbibliothek) kann die Datei nicht lesen. Kommt bei beschädigten Bilddateien vor.
- **OutOfMemoryException**: Bei sehr hochauflösenden Bildern (über 20 Megapixel). Lösung: Bild vor der OCR auf 300 DPI herunterskalieren.

8.3.4. Engine-Pooling für hohe Last

Das Erstellen einer neuen `TesseractEngine` dauert 300 bis 500 ms. Bei vielen gleichzeitigen Anfragen ist das ein Problem. Ein Pool hält vorkonfigurierte Engines bereit:

Listing 39: TesseractEngine-Pool (Konzept)

```
public class TesseractEnginePool
{
    private readonly ObjectPool<TesseractEngine> _pool;

    public TesseractEnginePool(string tessDataPath, int poolSize = 4)
    {
        var policy = new DefaultPooledObjectPolicy<TesseractEngine>
        {
            CreateFunc  = () => new TesseractEngine(tessDataPath, 'deu+eng',
                                                    EngineMode.Default),
            ReturnFunc  = engine => true
        };
        _pool = new DefaultObjectPool<TesseractEngine>(policy, poolSize);
    }
}
```

Der Pool hält vier Engines im Speicher. Jede Anfrage leiht sich eine aus, gibt sie nach der Verarbeitung zurück und die nächste Anfrage kann sie sofort nutzen.[17]

[17] Vgl. Microsoft: *Object Pooling in .NET*, https://learn.microsoft.com/en-us/aspnet/core/performance/objectpool, letzter Zugriff am 06.01.2026

9. KI-Normalisierung und Mapping

Nachdem der Text aus einem PDF oder Scan extrahiert wurde, kommt der schwierigste Teil: aus unstrukturierten Textzeilen sollen strukturierte, maschinenlesbare Daten werden. Dafür wird ein Large Language Model eingesetzt. Dieses Kapitel beschreibt, wie der Prompt gestaltet ist, welche Modelle evaluiert wurden und wie die Antwort des Modells in die XML-Strukturen überführt wird.

9.1. Prompt-Strategien

9.1.1. Das interne Datenmodell: InvoiceData

Zwischen LLM-Ausgabe und XML-Generierung liegt eine Zwischenstruktur: das `InvoiceData`-POCO. Das Modell ist absichtlich flach gebaut, damit das LLM ein einfaches JSON befüllen kann:

Listing 40: InvoiceData Datenmodell

```
1  public class InvoiceData
2  {
3      public string? InvoiceNumber { get; set; }
4      public decimal InvoiceAmount { get; set; }
5      public DateTime InvoiceDate { get; set; }
6      public DateTime DueDate { get; set; }
7      public string? VatNumber { get; set; }
8      public string? IBAN { get; set; }
9      public decimal? ShippingCost { get; set; }
10     public string? BillerName { get; set; }
11     public string? BillerStreet { get; set; }
12     public string? BillerZIP { get; set; }
13     public string? BillerTown { get; set; }
14     public string? RecipientName { get; set; }
15     public string? RecipientStreet { get; set; }
16     public List<InvoiceItem>? Items { get; set; }
17     public bool IsNetto { get; set; } = false;
18     public string? PriceAnalysis { get; set; }
19 }
```

Die `IsNetto`-Property löst ein konkretes Problem: Rechnungen können Beträge auf zwei Arten angeben - als Nettobetrag mit Mehrwertsteuer darauf, oder als Bruttobetrag der die Steuer schon enthält. Das LLM muss das erkennen und dem Backend melden, damit die Steuerberechnung stimmt.

9.1.2. Der System-Prompt

Der Prompt ist das eigentliche "Programm" für das Sprachmodell. Er kam erst nach vielen Iterationen zustande - jede Regel darin löst ein Problem, das in Tests aufgetaucht ist:

Listing 41: Gemini System-Prompt (Auszug)

```
private string BuildSystemPrompt() => @'
Du bist ein hochpraeziser Datenextraktions-Assistent fuer Rechnungen.

KRITISCH - NETTO vs. BRUTTO:
Setze 'isNetto': true wenn explizite Worte wie 'zzgl. MwSt.',
'Nettobetrag' oder separate Steuerzeilen forhanden sind.
Setze 'isNetto': false wenn 'inkl. MwSt.', 'Bruttobetrag' steht.

KRITISCH - ZAHLEN EXAKT LESEN:
- 150,00 → 150.00  (nicht 15000.00!)
- 2.000,00 → 2000.00 (Tausendertrennzeichen beachten!)
- NIEMALS raten -- nur Werte die im Text stehen!

Dokumentiere deine Netto/Brutto-Entscheidung in 'priceAnalysis'.
Antworte NUR mit validem JSON.';
```

Beispiel für eine gelernte Regel: Ohne die explizite Dezimalstellen-Warnung hat das Modell aus „150,00“ regelmäßig „15000.00“ gemacht, weil es das deutsche Komma als Tausendertrennzeichen interpretiert hat.

9.1.3. JSON-Schema für strukturierten Output

Damit das Modell garantiert gültiges JSON zurückgibt, wird ein Schema mitgeschickt:

Listing 42: JSON Schema Builder

```
private object BuildJsonSchema()
{
    return new
    {
        type = 'object',
        properties = new
        {
            invoiceNumber = new { type = 'string' },
            invoiceDate   = new { type = 'string', format = 'date' },
            invoiceAmount = new { type = 'number' },
            vatNumber     = new { type = 'string' },
            iban          = new { type = 'string' },
            billerName    = new { type = 'string' },
            billerStreet  = new { type = 'string' },
            billerZIP     = new { type = 'string' },
            billerTown    = new { type = 'string' },
            recipientName = new { type = 'string' },
            items = new
            {
                type  = 'array',
                items = new
                {
                    type       = 'object',
                    properties = new
                    {
                        position    = new { type = 'integer' },
                        description = new { type = 'string' },
                        quantity    = new { type = 'number' },
```

```
29                          unitPrice    = new { type = 'number' },
30                          totalPrice   = new { type = 'number' },
31                          taxRate      = new { type = 'number' }
32                      },
33                      required = new[] { 'position','description','quantity',
34                                         'unitPrice','totalPrice','taxRate' }
35                  }
36              },
37              isNetto       = new { type = 'boolean' },
38              priceAnalysis = new { type = 'string' }
39          },
40          required = new[] { 'invoiceNumber','invoiceDate','invoiceAmount','isNetto'
                }
41      };
42  }
```

Das Schema erzwingt Typsicherheit: `invoiceAmount` muss eine Zahl sein, kein String. `required`-Arrays definieren Pflichtfelder - das Modell kann kein JSON ohne `invoiceNumber` erzeugen.

9.2. Modelldiskussion

Im Laufe des Projekts wurden mehrere Sprachmodelle für die Rechnungsextraktion getestet. Die Wahl des richtigen Modells hat großen Einfluss auf Genauigkeit, Geschwindigkeit und Kosten.

9.2.1. Google Gemini 2.0 Flash (eingesetzt)

Gemini 2.0 Flash ist das Modell, das am Ende im Projekt eingesetzt wird. Der Hauptgrund: es unterstützt *Constrained Decoding* direkt in der API. Das bedeutet, man kann ein JSON-Schema mitschicken und das Modell kann buchstäblich kein Invalid-JSON erzeugen - auf Token-Ebene.

Tabelle 2.: Gemini 2.0 Flash - Eigenschaften

Eigenschaft	**Wert**
Kontext-Fenster	1M Tokens
JSON-Schema-Support	Nativ (response_schema)
Kosten	Kostenlos bis 15 req/min
Antwortzeit	2-5 Sekunden
Netto/Brutto-Erkennung	Sehr gut

Der Free-Tier mit 15 Anfragen pro Minute ist für den Projektzweck ausreichend. In einem echten Produktionssystem wäre ein kostenpflichtiger Tier nötig.[18]

[18]Vgl. Google AI: *Gemini API - Rate Limits*, https://ai.google.dev/gemini-api/docs/rate-limits, letzter Zugriff am 06.01.2026

9.2.2. Mistral 7B / Mistral Large

Mistral ist ein französisches Open-Source-Modell das auf selbst gehosteten Servern betrieben werden kann. Das wurde als interessante Alternative gesehen, weil damit der API-Key-Abhängigkeit von Google entgangen werden könnte.

Das Problem: Mistral 7B hat beim Testen mit deutschen Rechnungen deutlich schlechtere Ergebnisse gezeigt als Gemini. Besonders die Netto/Brutto-Erkennung war unzuverlässig - das Modell hat das `isNetto`-Flag oft falsch gesetzt, selbst wenn es im Text klar stand. Mistral Large (das größere Modell) wäre besser, ist aber kostenpflichtig und bringt keinen Vorteil gegenüber Gemini.

Ein weiterer Nachteil: Mistral unterstützt kein natives JSON-Schema wie Gemini. Man muss JSON via dem Prompt erzwingen („Antworte nur mit JSON“), was öfter mal zu kaputten JSON-Antworten führt.

9.2.3. Qwen 2.5

Qwen ist ein Modell von Alibaba Cloud, das auf Chinesisch und Englisch trainiert ist aber auch gutes Multilingual-Verständnis hat. Es wurde kurz getestet weil es in Benchmarks sehr gute Extraction-Ergebnisse zeigt.

Für das Projekt war es letztlich nicht geeignet: Die API ist in Europa langsamer (die Server stehen hauptsächlich in Asien), und die deutschen Steuerregeln (UStG-Rundungsvorschriften, österreichische UID-Formate) kennt es weniger gut als Gemini.

9.2.4. Gemma 2 (lokal)

Gemma ist Googles open-weights Modell, das lokal auf dem eigenen Rechner laufen kann - komplett offline. Das wäre ideal für Datenschutz, weil keine Rechnungsdaten an externe APIs geschickt werden.

Das getestete Gemma 2B ist aber für die Aufgabe zu klein: Es halluziniert Felder die nicht im Text stehen, und die Steuerberechnungs-Logik versteht es nicht zuverlässig. Das 9B-Modell wäre besser, braucht aber eine GPU mit mindestens 16 GB VRAM was in einer Schulumgebung nicht realistisch ist.

Fazit: Gemini 2.0 Flash bleibt die beste Wahl für dieses Projekt wegen der nativen JSON-Schema-Unterstützung, guter Genauigkeit bei deutschen Dokumenten und dem kostenlosem Tier für Entwicklung und Tests.

9.3. Parsing und Validierung der KI-Ausgaben

9.3.1. HTTP-Client und API-Aufruf

Listing 43: Gemini API-Aufruf

```
var requestBody = new
{
    contents = new[] { new { role = 'user',
        parts = new[] { new { text = systemPrompt + '\n\n' + userPrompt } } } },
    generationConfig = new
    {
        temperature        = 0.0,
        candidateCount     = 1,
        response_mime_type = 'application/json',
        response_schema    = BuildJsonSchema()
    }
};

var response = await _httpClient.PostAsync(
    $'{_apiEndpoint}?key={_apiKey}',
    new StringContent(JsonSerializer.Serialize(requestBody),
                      Encoding.UTF8, 'application/json'));

if (!response.IsSuccessStatusCode)
    throw new HttpRequestException($'Gemini API Fehler: {response.StatusCode}');
```

Die `generationConfig` mit `temperature = 0.0` macht das Modell so deterministisch wie möglich - bei gleicher Eingabe kommt immer die gleiche Ausgabe. Das ist wichtig für reproduzierbare Testergebnisse.

9.3.2. Response-Parsing

Die Antwort von Gemini kommt als JSON-Struktur mit mehreren Verschachtelungsebenen:

Listing 44: Gemini Response parsen

```
private InvoiceData ParseGeminiResponse(string responseBody)
{
    var doc = JsonDocument.Parse(responseBody);
    var text = doc.RootElement
        .GetProperty('candidates')[0]
        .GetProperty('content')
        .GetProperty('parts')[0]
        .GetProperty('text')
        .GetString();

    return JsonSerializer.Deserialize<InvoiceData>(text ?? '{}',
        new JsonSerializerOptions { PropertyNameCaseInsensitive = true })
        ?? new InvoiceData();
}
```

`PropertyNameCaseInsensitive = true` ist nötig, weil das LLM manchmal `invoiceNumber` und manchmal `InvoiceNumber` zurückgibt. Der Fallback `new InvoiceData()` verhindert NullPointerException bei leerem oder kaputtem JSON.

9.4. Mapping auf Domänen- und XML-Modelle

9.4.1. Netto/Brutto-Berechnung

Das ist der kritische Teil des Mappings. Das LLM hat bestimmt ob die Rechnung Netto- oder Brutto-Beträge enthält - jetzt müssen die richtigen Formeln angewendet werden:

Listing 45: Steuerberechnung Netto vs. Brutto

```
foreach (var item in data.Items)
{
    decimal netAmount, taxAmount, bruttoAmount;

    if (data.IsNetto)
    {
        // totalPrice vom LLM ist der Nettobetrag
        netAmount    = RoundAmount(item.TotalPrice);
        taxAmount    = RoundAmount(netAmount * item.TaxRate / 100);
        bruttoAmount = RoundAmount(netAmount + taxAmount);
    }
    else
    {
        // totalPrice vom LLM ist der Bruttobetrag
        bruttoAmount = RoundAmount(item.TotalPrice);
        netAmount    = RoundAmount(bruttoAmount / (1 + item.TaxRate / 100));
        taxAmount    = RoundAmount(bruttoAmount - netAmount);
    }

    _logger.LogInformation('Item {Pos}: Netto={Net}, USt={Tax}, Brutto={Brutto}',
        item.Position, netAmount, taxAmount, bruttoAmount);
}

private decimal RoundAmount(decimal amount) =>
    Math.Round(amount, 2, MidpointRounding.AwayFromZero);
```

Die Formeln:

- **Netto**: $\text{Brutto} = \text{Netto} \times \left(1 + \frac{\text{Steuersatz}}{100}\right)$
- **Brutto**: $\text{Netto} = \dfrac{\text{Brutto}}{1 + \frac{\text{Steuersatz}}{100}}$

`RoundAmount` rundet kaufmännisch (`MidpointRounding.AwayFromZero`): 2,125 wird auf 2,13 gerundet, nicht auf 2,12. Das entspricht den steuerrechtlichen Anforderungen.[19]

[19] Vgl. § 16 UStG: *Bemessungsgrundlage*, Rundung auf zwei Nachkommastellen

9.4.2. Von InvoiceData zu XML

Das befüllte `InvoiceData`-Objekt dient als gemeinsame Quelle für beide XML-Formate. ebInterface bekommt die flachen Felder direkt, ZUGFeRD braucht eine tiefere Verschachtelung über mehrere Objekte. Das `InvoiceData`-Modell wurde absichtlich so gestaltet, dass beide Mappings ohne Datenverlust möglich sind - die Biller/Recipient-Trennung deckt die Anforderungen beider Standards ab.[20]

[20]Vgl. ebInterface 6.1 Specification: *Austrian Standard*, Bundesrechenzentrum, 2022, Abschnitt 4.3

10. Generierung von ebInterface 6.1

Nachdem das KI-Modell die Rechnungsdaten in das interne `InvoiceData`-Objekt extrahiert hat, muss dieses in ein gültiges ebInterface 6.1 XML umgewandelt werden. Für das Bundesrechenzentrum und österreichische Behörden ist dieses Format Pflicht.

10.1. Objektmodell-Mapping

10.1.1. C#-Klassen für die XML-Struktur

Die ebInterface-Struktur wird direkt als C#-Objektmodell abgebildet, weil der `XmlSerializer` dann die Arbeit des XML-Schreibens übernimmt:

Listing 46: ebInterface Root-Element

```
[XmlRoot('Invoice', Namespace = 'http://www.ebinterface.at/schema/6p1/')]
public class EbInterfaceInvoice
{
    [XmlElement('InvoiceNumber')]
    public string InvoiceNumber { get; set; } = string.Empty;

    [XmlElement('InvoiceDate', DataType = 'date')]
    public DateTime InvoiceDate { get; set; }

    [XmlElement('Biller')]
    public BillerType Biller { get; set; } = new();

    [XmlElement('InvoiceRecipient')]
    public InvoiceRecipientType InvoiceRecipient { get; set; } = new();

    [XmlElement('Details')]
    public DetailsType Details { get; set; } = new();

    [XmlElement('Tax')]
    public TaxType Tax { get; set; } = new();

    [XmlElement('TotalGrossAmount')]
    public decimal TotalGrossAmount { get; set; }

    [XmlAttribute('GeneratingSystem')]
    public string GeneratingSystem { get; set; } = 'SmartBillConverter v1.0';

    [XmlAttribute('InvoiceCurrency')]
    public string InvoiceCurrency { get; set; } = 'EUR';
}
```

`[XmlRoot]` definiert den Root-Tag und den XML-Namespace. `[XmlElement]` mappt eine Property auf einen XML-Tag. `DataType = "date"` erzwingt das ISO-8601-Format (`YYYY-MM-DD`).

Die verschachtelten Typen für Rechnungssteller und Empfänger:

Listing 47: Adress- und Rechnungssteller-Typen

```
public class BillerType
{
    [XmlElement('VATIdentificationNumber')]
    public string VATIdentificationNumber { get; set; } = string.Empty;

    [XmlElement('Address')]
    public AddressType Address { get; set; } = new();
}

public class AddressType
{
    [XmlElement('Name')]
    public string Name { get; set; } = string.Empty;

    [XmlElement('Street')]
    public string? Street { get; set; }

    [XmlElement('ZIP')]
    public string? ZIP { get; set; }

    [XmlElement('Town')]
    public string? Town { get; set; }

    [XmlElement('Country')]
    public string? Country { get; set; }
}
```

Aus `InvoiceData.BillerName` und `BillerStreet` etc. werden die entsprechenden Felder in `BillerType.Address` befüllt. Das Mapping ist dabei direkt und ohne Logik - ein Feld geht in das andere.

10.2. Serialisierung mit XmlSerializer

Listing 48: ebInterface XML-Generierung

```
private string SerializeToEbInterfaceXml(EbInterfaceInvoice invoice)
{
    var namespaces = new XmlSerializerNamespaces();
    namespaces.Add('', 'http://www.ebinterface.at/schema/6p1/');

    var serializer = new XmlSerializer(typeof(EbInterfaceInvoice));
    var settings = new XmlWriterSettings
    {
        Indent            = true,
        IndentChars       = '  ',
        Encoding          = Encoding.UTF8,
        OmitXmlDeclaration = false
    };

    using var stringWriter = new StringWriter();
    using var xmlWriter    = XmlWriter.Create(stringWriter, settings);
    serializer.Serialize(xmlWriter, invoice, namespaces);
    return stringWriter.ToString();
}
```

`XmlSerializerNamespaces` verhindert, dass der Serializer automatisch unerwünschte Namespace-Präfixe wie xsi und xsd hinzufügt. `XmlWriterSettings` sorgt für einen lesbaren Einzug und stellt sicher, dass der XML-Header (`<?xml version="1.0"?>`) ganz oben steht.[21]

10.3. XSD-Validierung und Korrekturstrategien

10.3.1. Validierung gegen das offizielle Schema

Nach der Serialisierung wird das generierte XML gegen das offizielle ebInterface 6.1 XSD-Schema geprüft:

Listing 49: XSD-Validierung

```
1  private List<string> ValidateAgainstXsd(string xml, string xsdPath)
2  {
3      var errors  = new List<string>();
4      var schemas = new XmlSchemaSet();
5      schemas.Add('http://www.ebinterface.at/schema/6p1/', xsdPath);
6
7      var settings = new XmlReaderSettings
8      {
9          ValidationType = ValidationType.Schema,
10         Schemas        = schemas
11     };
12     settings.ValidationEventHandler += (sender, args) =>
13     {
14         errors.Add($'{args.Severity}: {args.Message}');
15         _logger.LogWarning('XSD: {Sev} - {Msg}', args.Severity, args.Message);
16     };
17
18     using var xmlReader = XmlReader.Create(new StringReader(xml), settings);
19     try { while (xmlReader.Read()) { } }
20     catch (XmlException ex) { errors.Add($'Parse Error: {ex.Message}'); }
21
22     return errors;
23 }
```

`XmlSchemaSet` lädt das XSD-File einmalig in den Speicher. `ValidationEventHandler` wird für jeden Fehler aufgerufen und sammelt alle Probleme, nicht nur den ersten. Das Durchlaufen des gesamten XML mit `while (xmlReader.Read())` sorgt dafür, dass auch Fehler auf den letzten Seiten gefunden werden.

10.3.2. Typische Validierungsfehler

- **Fehlendes Pflichtfeld**: `Element 'Invoice' missing required child 'InvoiceNumber'` - passiert wenn das KI-Modell keine Rechnungsnummer im Text findet

[21] Vgl. Microsoft: *XmlSerializer Class*, https://learn.microsoft.com/en-us/dotnet/api/system.xml.serialization.xmlserializer, letzter Zugriff am 06.01.2026

- **Falsches Datumsformat**: `Value '01.01.2024' is invalid according to datatype 'date'` - das deutsche Datumsformat muss erst in ISO-Format konvertiert werden
- **Ungültiger Währungscode**: `Value 'Euro' is not valid for attribute 'InvoiceCurrency'` - nur dreistellige ISO-4217-Codes wie `EUR` sind erlaubt

10.3.3. Korrekturstrategie bei Fehlern

Die `ValidationErrors`-Liste in der `LlmResponse` gibt dem Frontend genaue Informationen, was schiefgelaufen ist. Bei kritischen Fehlern (fehlendes Pflichtfeld) sollte der Benutzer aufgefordert werden, die Rechnung manuell zu überprüfen. Bei Formatfehlern (falsches Datum) kann das Backend oft automatisch korrigieren:

Listing 50: Datum-Korrektur bei Validierungsfehlern

```
// Deutschen Datumsformat zu ISO konvertieren
if (DateTime.TryParseExact(rawDate, 'dd.MM.yyyy',
    CultureInfo.InvariantCulture, DateTimeStyles.None, out var date))
{
    invoice.InvoiceDate = date;
}
else if (!DateTime.TryParse(rawDate, out date))
{
    _logger.LogWarning('Datum nicht parsebar: {RawDate}', rawDate);
    invoice.InvoiceDate = DateTime.Today; // Fallback
}
```

22

[22] Vgl. Bundesrechenzentrum: *ebInterface 6.1 Spezifikation*, https://www.ebinterface.at/download/ebinterface-6.1-specification.pdf, letzter Zugriff am 06.01.2026

11. Generierung von ZUGFeRD 2.3 / EN 16931

ZUGFeRD ist vom Aufbau her deutlich komplizierter als ebInterface. Die CII-XML-Struktur (Cross Industry Invoice) ist so tief verschachtelt, dass es mehrere C#-Klassen braucht, um einen einzigen Preis zu beschreiben. Dieses Kapitel erklärt, wie das Mapping und die Serialisierung im Projekt umgesetzt wurden.

11.1. CII-Mapping

11.1.1. ZUGFeRD-spezifisches Datenmodell

Für ZUGFeRD gibt es ein eigenes Datenmodell, das von `InvoiceData` befüllt wird:

Listing 51: ZUGFeRD Datenmodell

```
public class ZugferdInvoiceData
{
    public string InvoiceNumber { get; set; } = string.Empty;
    public DateTime InvoiceDate { get; set; }
    public decimal TotalAmount { get; set; }
    public decimal NetAmount { get; set; }
    public decimal TaxAmount { get; set; }
    public string Currency { get; set; } = 'EUR';
    public ZugferdParty Seller { get; set; } = new();
    public ZugferdParty Buyer { get; set; } = new();
    public List<ZugferdLineItem> LineItems { get; set; } = new();
    public string? IBAN { get; set; }
    public decimal TaxRate { get; set; } = 20.0m;
    public bool IsNetto { get; set; } = false;
}

public class ZugferdParty
{
    public string Name { get; set; } = string.Empty;
    public string Street { get; set; } = string.Empty;
    public string ZIP { get; set; } = string.Empty;
    public string City { get; set; } = string.Empty;
    public string Country { get; set; } = 'AT';
    public string? VatNumber { get; set; }
}
```

Das Modell verwendet `Seller/Buyer` statt `Biller/Recipient` - das ist die ZUGFeRD-typische Benennung.

11.1.2. XML-Namespaces

ZUGFeRD verwendet vier verschiedene XML-Namespaces für verschiedene Teile des Schemas:

Listing 52: ZUGFeRD Namespaces

```
public static class Namespaces
{
    public const string Rsm =
        'urn:un:unece:uncefact:data:standard:CrossIndustryInvoice:100';
    public const string Ram =
        'urn:un:unece:uncefact:data:standard:
            ReusableAggregateBusinessInformationEntity:100';
    public const string Udt =
        'urn:un:unece:uncefact:data:standard:UnqualifiedDataType:100';
    public const string Qdt =
        'urn:un:unece:uncefact:data:standard:QualifiedDataType:100';
}

[XmlRoot('CrossIndustryInvoice', Namespace = Namespaces.Rsm)]
public class CrossIndustryInvoice
{
    [XmlNamespaceDeclarations]
    public XmlSerializerNamespaces Xmlns { get; set; } =
        new XmlSerializerNamespaces(new[]
        {
            new XmlQualifiedName('rsm', Namespaces.Rsm),
            new XmlQualifiedName('ram', Namespaces.Ram),
            new XmlQualifiedName('udt', Namespaces.Udt),
            new XmlQualifiedName('qdt', Namespaces.Qdt)
        });

    [XmlElement('ExchangedDocumentContext', Namespace = Namespaces.Rsm)]
    public ExchangedDocumentContextType ExchangedDocumentContext { get; set; } =
        new();

    [XmlElement('ExchangedDocument', Namespace = Namespaces.Rsm)]
    public ExchangedDocumentType ExchangedDocument { get; set; } = new();

    [XmlElement('SupplyChainTradeTransaction', Namespace = Namespaces.Rsm)]
    public SupplyChainTradeTransactionType SupplyChainTradeTransaction { get; set;
        } = new();
}
```

`[XmlNamespaceDeclarations]` definiert die Namespace-Präfixe (rsm, ram, udt, qdt). Jedes `[XmlElement]` muss den passenden Namespace angeben, sonst scheitert die Validierung.

11.1.3. Profil-Deklaration (EN 16931)

ZUGFeRD definiert Profile. Das Projekt targetiert EN 16931 (COMFORT):

Listing 53: Profil-Deklaration

```
public class DocumentContextParameterType
{
    [XmlElement('ID', Namespace = Namespaces.Ram)]
    public string ID { get; set; } =
        'urn:cen.eu:en16931:2017#compliant#urn:factur-x.eu:1p0:extended';
}
```

Dieses Feld teilt dem Empfänger mit welches ZUGFeRD-Profil verwendet wird. Der Wert `urn:cen.eu:en16931:2017` bedeutet, dass das Dokument der europäischen Norm EN 16931 entspricht, was für Rechnungen an Behörden (B2G) Pflicht ist.[23]

11.1.4. Tiefe Verschachtelung: Preise in CII

Das CII-Modell ist deutlich tiefer verschachtelt als ebInterface. Um einen einzigen Preis zu setzen, müssen vier Klassen durchlaufen werden:

Listing 54: CII Preis-Verschachtelung

```
// 4 Ebenen fuer einen einzigen Preis:
// LineItem -> SpecifiedLineTradeAgreement
//          -> NetPriceProductTradePrice
//          -> ChargeAmount.Value
public class TradePriceType
{
    [XmlElement('ChargeAmount', Namespace = Namespaces.Ram)]
    public AmountType ChargeAmount { get; set; } = new();
}

public class AmountType
{
    [XmlAttribute('currencyID')]
    public string Currency { get; set; } = 'EUR';

    [XmlText]
    public decimal Value { get; set; }
}
```

Das ist der Preis für den universellen Ansatz von UN/CEFACT. Im Projekt wurden Helper-Methoden geschrieben, die diese Tiefe verbergen und das Befüllen einfacher machen.

11.2. Serialisierung, Namespace/Order-Postprocessing

Die Serialisierung läuft ähnlich wie bei ebInterface, aber das Namespace-Handling ist komplexer:

Listing 55: ZUGFeRD Serialisierung

```
private string SerializeToZugferdXml(CrossIndustryInvoice invoice)
{
    var serializer = new XmlSerializer(typeof(CrossIndustryInvoice));
    var settings = new XmlWriterSettings
    {
        Indent            = true,
        IndentChars       = '  ',
        Encoding          = Encoding.UTF8,
        OmitXmlDeclaration = false
    };

```

[23] Vgl. European Commission: *Directive 2014/55/EU on electronic invoicing*, https://ec.europa.eu/digital-building-blocks/wikis/display/DIGITAL/eInvoicing, letzter Zugriff am 06.01.2026

```
12      using var stringWriter = new Utf8StringWriter();
13      using var xmlWriter    = XmlWriter.Create(stringWriter, settings);
14      serializer.Serialize(xmlWriter, invoice);
15      return stringWriter.ToString();
16  }
```

Das Besondere bei ZUGFeRD: Die Reihenfolge der XML-Elemente innerhalb `SupplyChainTradeTransaction` ist vom Standard vorgegeben. `[XmlElement(Order = ...)]` steuert die Reihenfolge bei der Serialisierung:

Listing 56: Element-Reihenfolge steuern

```
1  public class SupplyChainTradeTransactionType
2  {
3      [XmlElement('IncludedSupplyChainTradeLineItem', Order = 1,
4                  Namespace = Namespaces.Ram)]
5      public List<SupplyChainTradeLineItemType> LineItems { get; set; } = new();
6
7      [XmlElement('ApplicableHeaderTradeAgreement', Order = 2,
8                  Namespace = Namespaces.Ram)]
9      public HeaderTradeAgreementType TradeAgreement { get; set; } = new();
10
11     [XmlElement('ApplicableHeaderTradeSettlement', Order = 3,
12                 Namespace = Namespaces.Ram)]
13     public HeaderTradeSettlementType TradeSettlement { get; set; } = new();
14 }
```

Ohne die `Order`-Attribute würde der `XmlSerializer` die Reihenfolge nach Alphabet oder Deklarationsreihenfolge wählen, was bei ZUGFeRD zu Validierungsfehlern führt.

11.3. Fallback-Strategien für Minimalvalidität

11.3.1. Pflichtfelder mit Defaultwerten absichern

Wenn das KI-Modell ein Pflichtfeld nicht findet, muss das Backend trotzdem ein gültiges XML produzieren können. Für ZUGFeRD sind besonders heikel:

- `GuidelineSpecifiedDocumentContextParameter.ID` - ohne dieses Feld ist das Dokument kein gültiges ZUGFeRD
- `SellerTradeParty.Name` - Rechnungssteller ist immer Pflicht
- `TaxTotalAmount` - muss mit Positionen übereinstimmen (BR-S-08)

Die Defaultwerte in den C#-Klassen (z. B. `Country = "AT"`, `Currency = "EUR"`) greifen automatisch wenn kein Wert befüllt wird.

11.3.2. PDF/A-3 (noch nicht implementiert)

ZUGFeRD ist eigentlich ein hybrides Format: PDF mit eingebettetem XML. Im aktuellen Projektstand gibt der `ZugferdService` nur XML als Text zurück. Für einen echten Produktionseinsatz wäre die vollständige PDF/A-3-Generierung sinnvoll, etwa über Razor-Templates zu HTML und dann zu PDF mit Playwright oder wkhtmltopdf.[24]

11.3.3. Schematron-Validierung (noch nicht implementiert)

Während XSD nur prüft ob die Struktur stimmt, würde Schematron auch inhaltliche Regeln prüfen können - zum Beispiel ob die Summe aller Positionssteuerbeträge mit dem Gesamtsteuerbetrag übereinstimmt (BR-S-08). Das ist im Projekt nicht eingebaut, wäre aber für einen echten Einsatz sinnvoll.[25]

Die Korrektheit der Steuerberechnung wird aktuell durch den deterministischen Code in `CalculateLineItems` sichergestellt (Kapitel 9.4). Das LLM liefert die Rohdaten, das Backend rechnet selbst nach.

[24] Vgl. iText Software: *iText 7 - PDF/A-3 and attachments*, https://kb.itextpdf.com/home/it7kb/ebooks/itext-7-jump-start-tutorial-for-java/chapter-7-creating-pdfs-with-pdf-a, letzter Zugriff am 06.01.2026

[25] Vgl. KoSIT: *ZUGFeRD 2.3 Schematron Validation Rules*, https://www.ferd-net.de/standards/zugferd-2.x/index.html, letzter Zugriff am 06.01.2026

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.[26]
- **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).[27]

[26] 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

[27] 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:

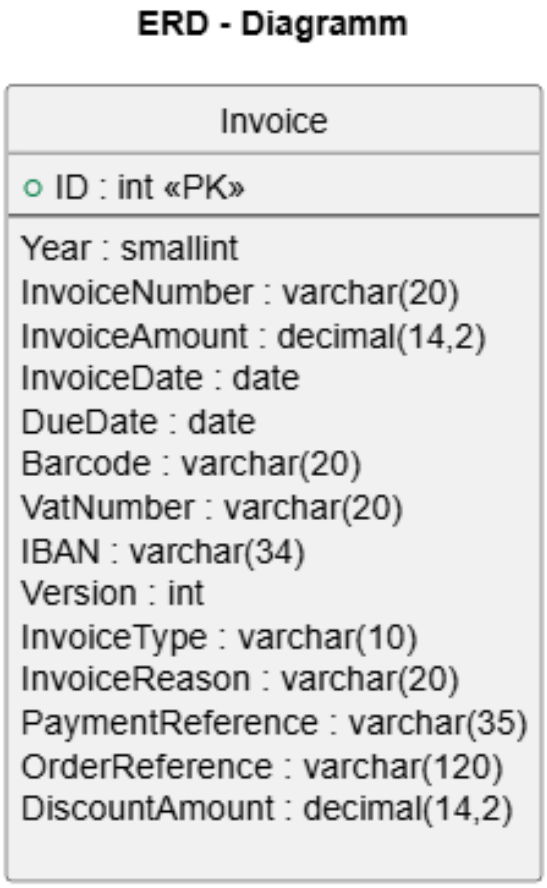

Abbildung 1.: Eigene Darstellung: Entity-Relationship-Diagramm der Invoice-Entität

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 57: TypeScript Invoice-Interface

```typescript
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:

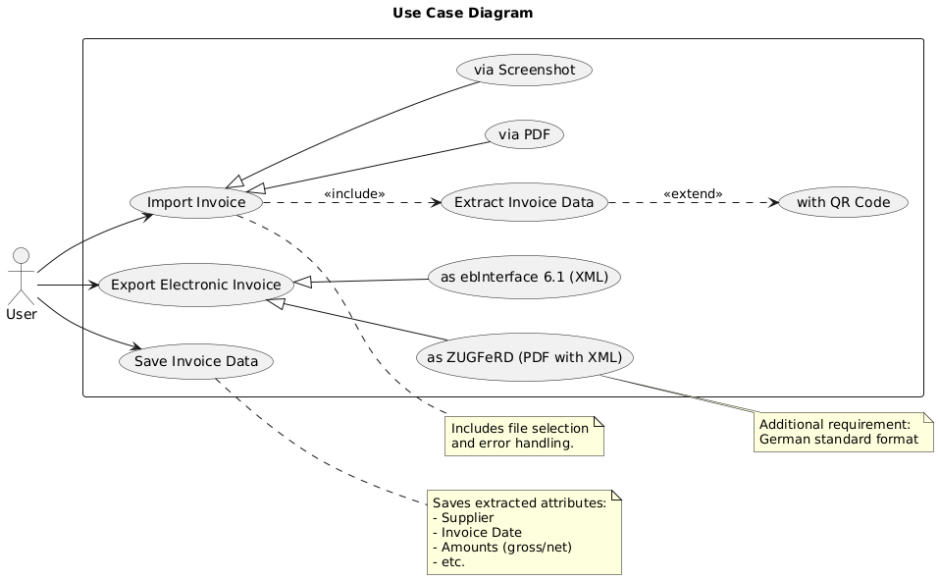

Abbildung 2.: Eigene Darstellung: Use-Case-Diagramm des SmartBillConverter

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:

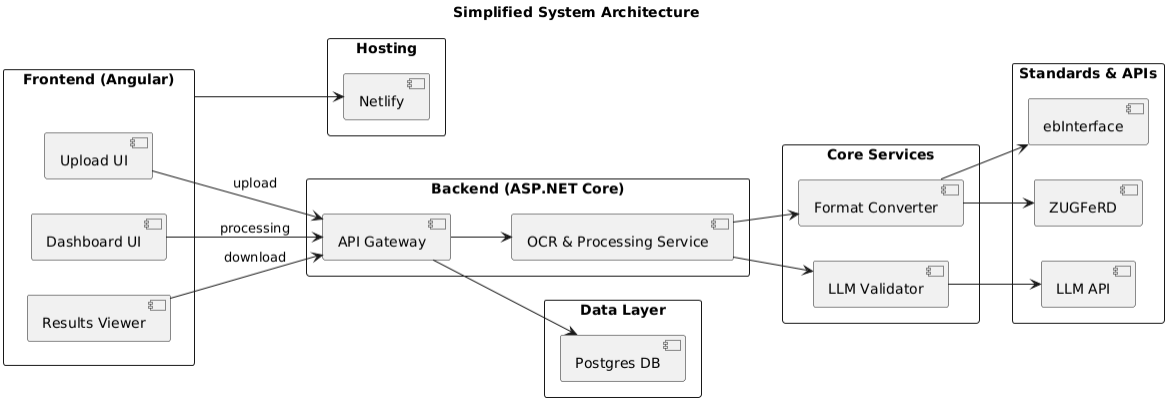

Abbildung 3.: Eigene Darstellung: Systemarchitektur des SmartBillConverter

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.[28] 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.

[28] 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 58: 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.[29]

```
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

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

geeignet, da sie direkt für Angular entwickelt wurde und eine unkomplizierte Integration ermöglicht.[30]

```
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 59: 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.

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

13.3.2. Services und Dependency Injection

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

Listing 60: 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.[31]
- **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.

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

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 61: Routing in app.routes.ts

```
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.[32] 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>
```

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

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).

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 62: 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.[33] 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 63: 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 64: 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 65: 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;
    }
```

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

```
9
10      const fileItem: FileUploadItem = {
11        file: file,
12        id: `file_${Date.now()}_${Math.random()}`,
13        status: 'pending'
14      };
15
16      this.loadFilePreview(fileItem);
17      this.selectedFiles.push(fileItem);
18    });
19  }
```

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

Listing 66: FileReader API für Vorschauen

```
1  private loadFilePreview(fileItem: FileUploadItem): void {
2    const reader = new FileReader();
3
4    if (fileItem.file.type.startsWith('image/')) {
5      reader.onload = (e) => fileItem.previewUrl = e.target?.result as string;
6      reader.readAsDataURL(fileItem.file);
7    } else if (fileItem.file.type === 'application/pdf') {
8      reader.onload = (e) => {
9        fileItem.pdfData = new Uint8Array(e.target?.result as ArrayBuffer);
10     };
11     reader.readAsArrayBuffer(fileItem.file);
12   }
13 }
```

Die HTML5 FileReader API bietet zwei Methoden: `readAsDataURL()` für Bilder (als Base64-String) und `readAsArrayBuffer()` für PDFs (als Binärdaten für den PDF-Viewer).[34]

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 67: ng2-pdf-viewer Installation

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

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

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

Listing 68: PDF-Viewer im Modal-Dialog

```
@if (showPreview && pdfData) {
  <div class='modal fade show d-block'>
    <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 69: 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.[35] Das ist wichtig, weil Browser diese URLs sonst bis zum Schließen der Seite speichern.

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

14.3. Datenstruktur für hochgeladene Dateien

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

Listing 70: 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 für Bildvorschau
  pdfData?: Uint8Array;      // Binärdaten für 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.[36] 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 71: File-Array-Management-Methoden

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

// Status-Update während 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 löschen
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

[36] Vgl. TypeScript Documentation: *Union Types*, typescriptlang.org: Union Types, abgerufen am 15.01.2025

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:

Listing 72: 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 73: 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 74: 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 75: 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 76: 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 77: 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 78: 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.[37] 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 79: Mehrstufige Fehlerbehandlung

```
this.invoiceService.uploadInvoice(currentFile.file).subscribe({
```

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

```
    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++;
      this.processNextFile();  // Continue-on-error Pattern
    }
  });
```

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 80: Error-Display im Template

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

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 81: 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 82: Toggle-Logik in der selectFormat-Methode

```
selectFormat(format: 'ebInterface' | 'ZUGFeRD'): void {
  if (this.selectedFormat === format) {
    // Wenn das Format bereits ausgewählt war, deselektieren
    this.selectedFormat = null;
    this.showUploadSection = false;
    this.clearAllFileData();
  } else {
    // Neues Format auswählen
    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 83: Dateiliste mit Karten-Design

```
<div class='card'>
  <div class='card-header d-flex justify-content-between'>
```

```
3       <h6 class='mb-0'>
4         <i class='bi bi-files me-2'></i>
5         {{ 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 84: 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 85: 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.[38]

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 86: 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 Auswählen</h4>
  <p class='text-muted'>
    PDF, PNG, JPG, BMP, TIFF, GIF - Max. 10MB pro Datei
  </p>
}
```

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

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.

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.[39]

Listing 87: Grundstruktur des InvoiceService

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

  constructor(private http: HttpClient) { }
}
```

Die `apiUrl` wird aus der Environment-Konfiguration geladen. Das ist praktisch, weil man die URL für Entwicklung und Produktion unterschiedlich setzen kann. Für Entwicklung

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

ist es `http://localhost:5000/api`, für Produktion würde man die echte Server-URL eintragen.

16.1.2. Intelligente Upload-Methode

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

Listing 88: 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.[40] Die Komponente kann das Observable mit `.subscribe()` abonnieren und bekommt die Daten, wenn die Server-Antwort eingetroffen ist.

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

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.

16.2.1. Separate Upload-Methode für ZUGFeRD

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

Listing 89: 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 90: 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.[41] Dadurch kann die Upload-Komponente einheitlich mit `result.zugferdXml` oder `result.workflow.ebInterfaceXml` auf das XML zugreifen.

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 91: 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.

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

16.4.1. Download mit Blob URLs

Der Download funktioniert über Blob URLs. Ein Blob ist ein temporäres Objekt, das Dateien im Browser repräsentiert:[42]

Listing 92: 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');
  a.href = url;
  a.download = `${format.toLowerCase()}_${invoiceNumber}.xml`;
  document.body.appendChild(a);
  a.click();
  document.body.removeChild(a);
  window.URL.revokeObjectURL(url);
}
```

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 93: Download aller XML-Dateien

```
downloadAllXml(): void {
  this.allResults.forEach((result, index) => {
    setTimeout(() => {
      this.invoiceService.generateXmlDownload(
        xmlContent,
        invoiceNumber,
        this.selectedFormat!
      );
    }, index * 500);
  });
}
```

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

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 94: Versandkosten-Anweisungen im KI-Prompt

```
- VERSANDKOSTEN: Suche nach Versand, Versandkosten, Paketgebühren,
  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 erwähnen 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 95: 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 96: Universelles Fallback-System

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

// Versandkosten hinzufügen, 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 97: Logging-Statements für Nachvollziehbarkeit

```
_logger.LogInformation($'Versandkosten hinzugefügt: {shippingCost} EUR');
_logger.LogWarning($'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 98: Vollständiger Reset beim Datei-Entfernen

```
1  private clearAllFileData(): void {
2    // File Input zurücksetzen
```

```
3     if (this.fileInput?.nativeElement) {
4       this.fileInput.nativeElement.value = '';
5     }
6
7     // Multi-file data löschen
8     this.selectedFiles = [];
9     this.currentProcessingIndex = -1;
10    this.allResults = [];
11
12    // Alle anderen Variablen zurücksetzen
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:[43]

Listing 99: ViewChild-Deklaration

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

Das entsprechende HTML-Element hat eine Template-Referenz:

Listing 100: 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.

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

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 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 101: 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 102: 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 103: 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 104: 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.[44]

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

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.

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 105: Fehlerbehandlung bei falschen Dateiformaten

```
1  const supportedTypes = ['application/pdf', 'image/png',
2                          'image/jpeg', 'image/bmp',
3                          'image/tiff', 'image/gif'];
4  if (!supportedTypes.includes(file.type)) {
5    this.showFileTypeError(file.name, file.type);
6    return;
7  }
```

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 106: 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