HTWK
Hochschule für Technik,
Wirtschaft und Kultur Leipzig

Fakultät Informatik

Bachelorarbeit

Digitalisierung von Prozessen zur Erstellung, Bearbeitung und Verwaltung von Formularen

Prüfer: Herr Prof. Dr. rer. nat. Thomas Riechert
Betreuer: Herr M. Sc. Tobias Höppner

Vorgelegt von:
Nouralrahman Hussain, Matrikelnummer: 79968
leipzig, 14.12.2025

Inhaltsverzeichnis

1 Einleitung **4**

1.1 Motivation und Problemstellung 4
1.2 Zielsetzung und Forschungsfragen 5
1.3 Abgrenzung und Rahmen der Arbeit 6
1.4 Aufbau der Arbeit 7

2 Anforderungen an das digitale Formularsystem **9**

2.1 Rahmenbedingungen der Hochschule 9
2.2 Rollenmodell und Zugriffskonzept 9
2.2.1 Rolle: Studienamt/Management 10
2.2.2 Rolle: Dozierende (mit Token-Link) 10
2.3 Funktionale Anforderungen 10
2.3.1 Arbeiten mit Dozentenblättern und Zuarbeitsformularen 10
2.3.2 Stammdaten und Autofill 11
2.3.3 Fristen und planungsübersicht 11
2.3.4 Feedback und Fehlermeldungen 11
2.4 Nicht-funktionale Anforderungen 12
2.4.1 Benutzerfreundlichkeit 12
2.4.2 Sicherheit 12
2.4.3 Wartbarkeit und Weiterentwicklung 12
2.4.4 Transparenz und Auswertbarkeit 12
2.5 Zusammenfassung 12

3 Stand der Technik **13**

3.1 Frontend-Grundlagen: HTML, CSS und JavaScript 13
3.1.1 HTML: Struktur, Semantik und Formularelemente 13
3.1.2 CSS: Darstellung, Layout und Zustände im Formular 14
3.1.3 JavaScript: Interaktion, Validierung und Kommunikation 14
3.2 UI-Frameworks und Build-Tooling 14
3.2.1 React als verbreitete Basis für komponentenbasierte UIs 15
3.2.2 Next.js als React-Framework für produktionsnahe Webanwendungen 15
3.2.3 Angular als Alternative mit umfassendem Framework-Anspruch 16
3.3 Formularframeworks: schema-getriebene Ansätze 16
3.3.1 RDForms (RDF-orientierte Formularerstellung) 17
3.3.2 `shacl-form` (Formulare aus SHACL-Shapes als Web Component) 18
3.3.3 SurveyJS (JSON-definierte Formulare mit Fokus auf Surveys) 19
3.3.4 JSON Forms (JSON Schema + UI Schema, besonders relevant für React) 20
3.3.5 Form.io (Plattformansatz: Form Builder, JSON-Modelle, Backend-Integration) 21
3.3.6 Zusammenfassende Einordnung der Frameworks 22

3.4 Backend und API-Stil: Node.js und REST über HTTP . . . 22
3.5 Datenhaltung: RDF/Turtle sowie JSON-Dateispeicher . . . 22
3.6 Deployment: Containerisierung mit Docker und Podman . . . 23
3.7 Zusammenfassung . . . 23

4 Architektur und Design **24**
4.1 Zielarchitektur: Schichten, Schnittstellen und Verantwortlichkeiten . . . 24
4.2 Standardvariante: HTML/CSS/JavaScript und RDF/Turtle-Persistenz . . . 25
4.2.1 Komponentenübersicht und Datenfluss . . . 25
4.2.2 Backend-Design mit Node.js und Express . . . 25
4.2.3 Persistenz als RDF-Graph und Turtle-Dateien . . . 25
4.3 Framework-Variante: React, JSON Forms/JSON Schema und JSON-Dateispeicher . . . 26
4.3.1 Komponentenübersicht und Motivation . . . 26
4.3.2 Frontend-Design mit React und schema-getriebenem Rendering . . . 26
4.3.3 Backend und Persistenz als JSON-Dateien . . . 27
4.3.4 Authentifizierung, Rollen und Share-Links . . . 27
4.4 Vergleich der Varianten als Entwurfsentscheidung . . . 27

5 Implementierung **28**
5.1 Variante A: Standard-Implementierung (HTML/CSS/JS) . . . 28
5.1.1 Frontend: Seiten, Layout und Interaktion . . . 29
5.1.2 Clientseitige Validierung und Feedback . . . 31
5.1.3 Autofill aus Modulstammdaten . . . 31
5.1.4 Backend: Express-Endpunkte und Serverlogik . . . 31
5.1.5 Persistenz: Transformation nach RDF und Serialisierung als Turtle . . . 32
5.2 Variante B: Framework-basierte Implementierung
(React + JSON Forms + JSON-Dateispeicher) . . . 33
5.2.1 Frontend: Einstieg, Login und Formularansichten . . . 33
5.2.2 Projektstruktur und Build-Setup . . . 36
5.2.3 Formularerzeugung, Validierung und Fehlermeldungen . . . 37
5.2.4 Autofill-Mechanismus in React . . . 37
5.2.5 Backend und JSON-basierte Persistenz . . . 37
5.3 Zwischenfazit . . . 38

6 Tests und Evaluation **39**
6.1 Testkontext und Grundannahmen . . . 39
6.2 Funktionale Tests: Korrektheit der Kernabläufe . . . 39
6.3 Robustheit und Fehlerrückmeldungen . . . 40
6.4 Pilotmessung zu Optimierungseffekten und Bearbeitungszeiten . . . 40
6.5 Datenqualität: Vollständigkeit und Validierungsfehler . . . 41
6.6 Änderbarkeit und Umsetzungsaufwand . . . 41
6.7 Usability und Nutzererlebnis . . . 42
6.8 Einordnung zu Anforderungen, Self-Hosting und Open Source . . . 42
6.9 Zusammenfassung der Evaluation . . . 43

7 Diskussion **44**
7.1 Einordnung zu FF1: Anforderungen im Hochschulkontext . . . 44
7.2 Einordnung zu FF2: Vergleich nach Änderbarkeit, Datenqualität und Aufwand . . . 45
7.2.1 Änderbarkeit und Wartbarkeit . . . 45
7.2.2 Datenqualität und Validierung . . . 45

7.2.3 Umsetzungsaufwand 45
7.3 Einordnung zu FF3: Messbare Optimierungseffekte 46
7.4 Einordnung zu FF4: Usability und Nutzererlebnis 46
7.5 RDF/Turtle vs. JSON: Interpretation der Persistenzentscheidung 46
7.6 Limitationen und Bedrohungen der Validität 47
7.7 Praktische Implikationen für den Hochschulbetrieb 47
7.8 Zusammenfassung der Diskussion 48

8 Fazit und Ausblick **49**
8.1 Ausblick 49

Kapitel 1

Einleitung

1.1 Motivation und Problemstellung

An vielen Hochschulen sind Lehr- und Lernprozesse inzwischen weitgehend digital organisiert: Veranstaltungen werden über Lernplattformen verwaltet, Prüfungsleistungen online verbucht und Materialien elektronisch bereitgestellt. In der Verwaltung zeigt sich jedoch ein anderes Bild. Zahlreiche Abläufe – etwa bei der Lehrplanung, der Antragsbearbeitung oder der Dokumentation – basieren weiterhin auf Papier- oder PDF-Formularen. Studien und Erfahrungsberichte beschreiben diese Prozesse als „sehr papierlastig“ und heben hervor, dass durchgängige digitale Workflows in Verwaltungen noch immer die Ausnahme sind [16].

Gleichzeitig belegt die E-Government-Literatur seit Jahren deutliche Effizienzgewinne durch die Digitalisierung von Verwaltungsprozessen. Digitale Verfahren können sowohl die *Bearbeitungszeit für die Nutzenden* (user time) als auch die *Durchlaufzeit von Vorgängen* (elapsed processing time) reduzieren [42, 1]. Kontrollierte Studien zeigen, dass die digitale Datenerfassung gegenüber Papier nicht nur Zeit spart, sondern auch die Datenqualität verbessert, etwa durch weniger Übertragungsfehler und fehlende Pflichtangaben [24, 63]. Fallbeispiele aus Hochschulen und Forschungseinrichtungen berichten von erheblichen Einsparungen: So konnten in einem komplexen Formular-Workflow an der University of California San Diego über 800 Arbeitsstunden pro Jahr eingespart werden [36]. Ein Universitätsklinikum reduzierte die Durchlaufzeit in der Rechnungsverarbeitung von 20 auf 3 Tage und senkte die Bearbeitungszeit pro Rechnung um mehrere Minuten [70].

Neben der reinen Zeiteinsparung spielt die *digitale Verfügbarkeit und Wiederverwendbarkeit von Daten* eine zentrale Rolle. Wenn Informationen konsistent und strukturiert erfasst werden, lassen sie sich einfacher aktualisieren, auswerten und für Folgeprozesse nutzen [42, 24]. Dies ist insbesondere für Hochschulen relevant, in denen Lehr-, Forschungs- und Verwaltungsdaten in unterschiedlichen Systemen vorgehalten und verknüpft werden müssen.

Ein weiterer wichtiger Aspekt ist die Barrierefreiheit. Als öffentliche Einrichtungen sind Hochschulen rechtlich verpflichtet, ihre digitalen Angebote schrittweise barrierefrei zu gestalten (Richtlinie (EU) 2016/2102, EN 301 549; fachlich konkretisiert durch WCAG 2.1/2.2 [6, 68, 69]). Klassische PDF-Formulare sind häufig nur mit hohem Zusatzaufwand barrierefrei nutzbar. Webbasierte, strukturiert aufgebaute Formulare bieten hier deutliche Vorteile, da sie semantische Informationen, Tastaturbedienbarkeit und klare Fehlerrückmeldungen von Beginn an berücksichtigen können.

Zusammengefasst ergibt sich somit eine klare Problemstellung: Verwaltungsprozesse an Hochschulen sind oft noch papier- bzw. PDF-basiert, obwohl durch digitale, gut gestaltete Formular-Workflows nachweislich Zeit gespart, Fehler reduziert, Daten besser nutzbar gemacht und rechtliche Anforderungen an Barrierefreiheit besser erfüllt werden könnten. Diese Arbeit nimmt diese Lücke am Beispiel der Lehrplanungsprozesse „Dozentenblatt“ und „Zuarbeitsformular“ in den Blick.

1.2 Zielsetzung und Forschungsfragen

Ziel der Arbeit

Ziel dieser Bachelorarbeit ist es, einen praxistauglichen, barrierearmen Ansatz zur Digitalisierung papierbasierter Formularprozesse im Hochschulkontext zu entwickeln und prototypisch umzusetzen. Im Mittelpunkt stehen zwei technische Realisierungen, die miteinander verglichen werden:

1. eine *Standardvariante*, bei der die Formulare klassisch mit HTML, CSS und JavaScript umgesetzt werden,[1]
2. eine *Framework-Variante*, bei der Formulare deklarativ über JSON Schema und ein Formular-Framework (z. B. JSON Forms in einer React-Umgebung) beschrieben werden.[2]

Beide Varianten sollen ein nutzerfreundliches Web-Frontend bereitstellen und die erfassten Daten im Backend strukturiert und auswertbar speichern. Darüber hinaus integrieren sie Optimierungsfunktionen wie Autofill, eine einfache Fristenlogik sowie Status- und Zeit-Feedback und berücksichtigen grundlegende Anforderungen an Datenschutz, Selbst-Hosting und Barrierefreiheit. Die Standardvariante dient dabei als Ausgangspunkt und „Baseline“. Die Framework-Variante soll zeigen, ob sich mit einem schemagetriebenen Ansatz die Änderbarkeit und Datenqualität verbessern und Entwicklungsaufwände langfristig reduzieren lassen.

Forschungsfragen

Auf dieser Basis werden folgende Forschungsfragen untersucht:

FF1: Anforderungen: Welche funktionalen und nicht-funktionalen Anforderungen sind für die Digitalisierung formularbasierter Prozesse im Hochschulkontext besonders wichtig (z. B. Datenschutz/DSGVO, Selbst-Hosting, Validierung, Rollen- und Rechtekonzept, Nachvollziehbarkeit, Barrierefreiheit, einfache Anpassbarkeit)?

FF2: Vergleich der Ansätze: Inwiefern unterscheiden sich die Standardvariante und die Framework-Variante hinsichtlich

(a) Änderbarkeit (z. B. Zeit für Feld- und Regeländerungen),

(b) Datenqualität (z. B. Vollständigkeit, Anzahl von Validierungsfehlern),

(c) und Umsetzungsaufwand (z. B. Entwicklungszeit, Umfang des Quellcodes)?

FF3: Optimierungseffekte: Tragen Funktionen wie Autofill, Fristenlogik oder Status-/Zeit-Feedback messbar dazu bei, Bearbeitungszeiten zu verkürzen und Rückfragen im Prozess zu reduzieren?

FF4: Usability und Nutzererlebnis: Wie erleben Nutzerinnen und Nutzer die Übersichtlichkeit und Bedienbarkeit der beiden Varianten (z. B. Verständnis der Formulare, Orientierung im Ablauf, Rückmeldungen des Systems)?

Zur Beantwortung dieser Fragen werden beide Varianten technisch umgesetzt und anschließend im Rahmen verschiedener Tests miteinander verglichen.

[1] `https://github.com/HussainNour/Formulare_Digitalisierung`
[2] `https://github.com/HussainNour/Formular-Digitalisierung-Framework-Edition`

Messkonzept und Erfolgsmaße

Um die Forschungsfragen systematisch zu beantworten, werden mehrere Arten von Kennzahlen erhoben, die sich grob in vier Gruppen einteilen lassen:

- **Zeitbezogene Kennzahlen:** Für ausgewählte Aufgaben, etwa das Anlegen eines neuen Zuarbeitsformulars, wird erfasst, wie lange Teilnehmende für die Bearbeitung benötigen (Bearbeitungszeit) und wie lange ein Vorgang vom Anlegen bis zur endgültigen Freigabe braucht (Durchlaufzeit). Ergänzend wird dokumentiert, bei wie vielen Vorgängen Rückfragen oder Korrekturen erforderlich sind, woraus sich eine Rückfragenquote ergibt.
- **Kennzahlen zur Datenqualität:** In diesem Bereich wird untersucht, ob Pflichtfelder vollständig ausgefüllt werden (Vollständigkeit), wie viele Validierungsfehler im Durchschnitt pro Einreichung auftreten (z. B. fehlende Felder oder falsche Formate) und wie häufig Einreichungen serverseitig erneut scheitern, obwohl sie clientseitig zunächst als gültig galten (Re-Validierungsquote).
- **Usability und Nutzererlebnis:** Mit einem kurzen Fragebogen und ergänzenden offenen Kommentaren wird erfasst, wie verständlich die Formulare sind, wie gut sich die Teilnehmenden im Ablauf zurechtfinden, wie sie Übersichtlichkeit und Bedienbarkeit der Anwendung bewerten und wie hilfreich sie die Rückmeldungen des Systems erleben.
- **Nutzung von Optimierungsfunktionen:** Schließlich wird betrachtet, wie die angebotenen Komfortfunktionen tatsächlich genutzt werden. Dazu gehört unter anderem, wie häufig vorgeschlagene Autofill-Werte übernommen werden, wie groß der Unterschied in der Bearbeitungszeit zwischen Eingaben mit und ohne Autofill ist und inwieweit automatisch berechnete Fristen und Hinweise fachlich korrekt sind und von den Nutzenden beachtet werden.

Diese Kennzahlen erlauben es, die beiden Varianten nicht nur qualitativ, sondern auch quantitativ miteinander zu vergleichen und die Arbeitshypothesen zu Zeitersparnis, Datenqualität und Benutzerfreundlichkeit zu prüfen.

1.3 Abgrenzung und Rahmen der Arbeit

Gegenstand und Umfang

Die Bachelorarbeit konzentriert sich auf die Digitalisierung zweier konkreter formularbasierter Abläufe in der Lehrplanung: das Dozentenblatt und das Zuarbeitsformular. Im Zentrum stehen dabei die Gestaltung und Umsetzung der Web-Formulare, ausgewählte Optimierungsfunktionen wie ein Modul-Autofill, ein einfaches Rollen- und Rechtekonzept für Manager und Dozierende sowie die strukturierte Speicherung der Formulardaten.

Nicht Bestandteil der Arbeit sind hingegen die endgültige Erzeugung rechtsverbindlicher PDF-Dokumente und der Umgang mit qualifizierten elektronischen Signaturen, ein produktiver Rollout für die gesamte Hochschule sowie eine umfassende Integration in bestehende Identity-Management- oder Campus-Management-Systeme. Ebenfalls nicht umgesetzt werden Themen wie Langzeitarchivierung, Hochverfügbarkeit oder detaillierte Monitoring- und Auditkonzepte. Diese Aspekte werden in der Arbeit zwar konzeptionell adressiert, aber nicht technisch realisiert, da sie den Rahmen eines Bachelorprojekts deutlich überschreiten würden.

Systemgrenzen und Speicheroptionen

Die Prototypen folgen einer klaren Trennung von Frontend, Backend und Persistenzschicht. Das Frontend ist jeweils als Webanwendung umgesetzt, das Backend als schlanker Node.js-Server mit REST-artigen Endpunkten. Die Formulardaten werden über HTTP übertragen und im Backend in strukturierter Form persistiert.

Es wurden drei Speicherwege prototypisch umgesetzt und gegenübergestellt:

1. **RDF/Turtle als Exportformat:** Formulardaten werden in ein RDF-Graphmodell überführt und als Turtle-Dateien gespeichert. Dieser Weg eignet sich vor allem, wenn eine spätere Integration in semantische Infrastrukturen oder Triple-Stores mit SPARQL-Abfragen vorgesehen ist.
2. **Zentrale JSON-Datei (`db.json`):** Alle Einträge werden in einer gemeinsamen JSON-Datei als Array gehalten. Der Ansatz erlaubt sehr schnelles Prototyping, stößt aber bei konkurrierenden Zugriffen und differenzierten Rechtekonzepten schnell an Grenzen.
3. **Getrennte JSON-Dateien pro Dozentin/Dozent:** Hier erhält jede Person bzw. jeder Vorgang eine eigene Datei. Ein leichtgewichtiger JSON-Store verwaltet die Dateien und stellt eine einfache API bereit. Dieser Ansatz ist für kleine bis mittlere Datenmengen gut geeignet und unterstützt eine klare Trennung der Daten je Dozierendem.

Für den Prototyp dieser Arbeit wird die dritte Variante bevorzugt, da sie das Risiko versehentlicher Querzugriffe reduziert und das einfach umzusetzen ist. Für größere Installationen könnte dieser Ansatz später durch ein Datenbank- oder Bucket-Konzept mit vergleichbarer Trennung abgelöst werden.

Arbeitsteilung

Die Arbeit selbst umfasst die Konzeption und Implementierung des Frontends (Standard- und Framework-Variante), die Entwicklung der Optimierungsfunktionen, die grundlegende Serverlogik zur Datenspeicherung, das einfache Rollen- und Rechtekonzept sowie die Planung und Durchführung der Evaluation.

Aufgaben wie die Ausgestaltung einer produktionsreifen Betriebsumgebung, die Definition von Backup- und Recovery-Strategien oder die Integration in bestehende zentrale Systeme werden in der Arbeit nur umrissen und wären Gegenstand nachgelagerter Projekte.

1.4 Aufbau der Arbeit

Die Arbeit gliedert sich in acht Kapitel, die inhaltlich aufeinander aufbauen:

- **Kapitel 1 (Einleitung)** beschreibt Motivation und Problemstellung, formuliert Zielsetzung und Forschungsfragen, grenzt den Umfang der Arbeit ab und gibt einen Überblick über den Aufbau.
- **Kapitel 2 (Anforderungen)** Welche funktionalen und nicht-funktionalen Anforderungen sind für die Digitalisierung formularbasierter Prozesse im Hochschulkontext zentral, insbesondere im Hinblick auf Selbst-Hosting und Datenschutz, einfache und sichere Anmeldung, rollenbasierten Zugriff, nutzerfreundliche Oberflächen und lizenzfreie Open-Source-Technologien?
- **Kapitel 3 (Stand der Technik)** ordnet die in der Arbeit verwendeten Ansätze in die aktuelle Fachliteratur und bestehende Lösungen ein, etwa klassische Web-Stacks, schemabasierte Formframeworks und Beispiele.
- **Kapitel 4 (Architektur und Design)** beschreibt die Zielarchitektur der Anwendung, die beiden Varianten (Standard- und Framework-Variante) sowie zentrale Entwurfsentscheidungen zu Rollen, Schnittstellen und Datenhaltung.
- **Kapitel 5 (Implementierung)** dokumentiert die konkrete Umsetzung der Prototypen: Frontend, Backend, Persistenz und die realisierten Optimierungsfunktionen.
- **Kapitel 6 (Tests und Evaluation)** stellt das Evaluationsdesign, die eingesetzten Metriken und die Durchführung der Nutzertests vor und berichtet die erhobenen Ergebnisse.
- **Kapitel 7 (Diskussion)** ordnet die Ergebnisse in Bezug auf die Forschungsfragen und den Stand der Technik ein, diskutiert Limitationen und leitet Implikationen für die Praxis ab.

- **Kapitel 8 (Fazit und Ausblick)** fasst die Arbeit zusammen, beantwortet die Forschungsfragen, benennt zentrale Einschränkungen und gibt einen Ausblick auf mögliche Weiterentwicklungen und zukünftige Forschungsarbeiten.

Damit schafft die Einleitung den Rahmen für die weitere Arbeit: Die folgenden Kapitel führen jeweils einen Schritt näher an die zentrale Frage heran, wie papierbasierte Formularprozesse an Hochschulen so digitalisiert werden können, dass sie effizienter, robuster und benutzerfreundlicher werden – ohne die organisatorischen und rechtlichen Anforderungen aus dem Blick zu verlieren.

Kapitel 2

Anforderungen an das digitale Formularsystem

In diesem Kapitel werden die Anforderungen an das digitale Formularsystem beschrieben, das die papierbasierten Prozesse „Dozentenblatt“ und „Zuarbeitsformular“ ablösen soll. Der Fokus liegt auf den Rahmenbedingungen der Hochschule, den Rollen und Zugriffswegen sowie auf den wichtigsten funktionalen und nicht-funktionalen Anforderungen an die Anwendung.

Die in diesem Kapitel beschriebenen Anforderungen wurden in mehreren fachlichen Abstimmungen mit Mitarbeitenden des Studienamts der HTWK Leipzig erarbeitet und geben die dort definierten Prozesse und Rahmenbedingungen wieder. **Es handelt sich** damit überwiegend um **interne Vorgaben und nicht um aus der Literatur übernommene Anforderungen.**

2.1 Rahmenbedingungen der Hochschule

Das System wird im Umfeld einer Hochschule eingesetzt und soll dort langfristig betrieben und weiterentwickelt werden. Daraus ergeben sich mehrere grundlegende Rahmenbedingungen.

Erstens soll die Anwendung auf hochschulnaher Infrastruktur betrieben werden können, zum Beispiel im Rechenzentrum der Fakultät. Externe Cloud-Dienste sollen nach Möglichkeit vermieden werden, damit personenbezogene Daten die Hochschule nicht unnötig verlassen. Diese Form des Self-Hostings unterstützt eine datenschutzkonforme Verarbeitung nach der Datenschutz-Grundverordnung (DSGVO).

Zweitens sollen, wo immer möglich, lizenzfreie oder Open-Source-Komponenten verwendet werden. Es sollen keine laufenden Lizenzkosten entstehen, und der Quellcode soll bei Bedarf geprüft und an die lokalen Anforderungen angepasst werden können. Dies ist nicht nur aus Kostengründen sinnvoll, sondern erleichtert auch die langfristige Wartung.

Drittens spielt der Schutz personenbezogener Daten eine zentrale Rolle. Die Formulare enthalten personenbezogene Daten von Dozierenden und teilweise weitere sensible Angaben zur Lehrplanung. Der Zugriff auf diese Daten soll auf die Personen und Rollen beschränkt sein, die sie tatsächlich für ihre Arbeit benötigen. Wenn Daten außerhalb der Hochschule verarbeitet oder in Drittländer übermittelt werden, müssen die Vorgaben der DSGVO eingehalten werden.

Diese Rahmenbedingungen stecken den organisatorischen und technischen Rahmen ab, in dem das System gestaltet werden soll.

2.2 Rollenmodell und Zugriffskonzept

Für die betrachteten Prozesse sind zwei Hauptrollen relevant: das Studienamt bzw. Management und die Dozierenden. Das Zugriffskonzept ist bewusst einfach gehalten, um den administrativen Aufwand gering zu halten

und die Nutzung zu vereinfachen.

2.2.1 Rolle: Studienamt/Management

Mitarbeitende im Studienamt oder vergleichbare Managerinnen und Manager sind für die Planung und Koordination der Lehrveranstaltungen verantwortlich. Sie arbeiten regelmäßig mit dem System und erhalten deshalb einen klassischen Login. Die Anmeldung erfolgt über Benutzername und Passwort, zum Beispiel über ein zentrales Hochschul-Login.

Nach der Anmeldung können diese Personen neue Dozentenformulare und Zuarbeitsformulare für kommende Semester anlegen. Sie pflegen außerdem die dazugehörigen Stammdaten, also insbesondere die Liste der Module mit Modulnummer, Modulnamen, Modulverantwortlichen sowie den Standard-Semesterwochenstunden (SWS) für Vorlesung, Seminar und Praktikum. Darüber hinaus greifen sie auf Übersichten zu, in denen alle Formulare mit ihrem aktuellen Bearbeitungsstatus sichtbar sind. Sie können die Einträge nach Semester, Modul oder Name der Dozierenden filtern, Rückfragen stellen und Formulare nach Prüfung als freigegeben markieren.

2.2.2 Rolle: Dozierende (mit Token-Link)

Dozierende sollen ihre Formulare möglichst einfach ausfüllen können, ohne ein eigenes Benutzerkonto im System anlegen zu müssen. Sie erhalten daher einen persönlichen Link zu ihren Formularen. Dieser Link enthält einen eindeutigen, zufällig erzeugten Token.

Wenn das Studienamt ein neues Formular für eine dozierende Person anlegt, erzeugt das System automatisch einen solchen Token und bettet ihn in einen Link ein. Dieser Link wird an die Dozentin oder den Dozenten übermittelt, zum Beispiel per E-Mail. Über den Link kann die Person ihr eigenes Formular aufrufen, ausfüllen, zwischenspeichern und später erneut bearbeiten. Der Token-Link erlaubt nur den Zugriff auf die eigenen Formulare; andere Einreichungen oder Übersichten sind auf diese Weise nicht zugänglich.

Technisch sind solche Token-Links eine verbreitete Methode, um personalisierten Zugriff ohne vollwertiges Accounts-System zu ermöglichen. Wichtig ist, dass die Tokens ausreichend lang und zufällig sind, damit sie nicht erraten werden können. Sicherheitsrichtlinien wie die OWASP Top 10 und der OWASP Application Security Verification Standard (ASVS) empfehlen robuste Token und eine strikte Zugriffskontrolle.

2.3 Funktionale Anforderungen

Im Folgenden werden die wichtigsten fachlichen Anforderungen an das System beschrieben. Sie betreffen insbesondere den Umgang mit Dozentenblättern und Zuarbeitsformularen, den Einsatz von Stammdaten und Autofill-Funktionen, die Behandlung von Fristen und Statusinformationen sowie die Rückmeldung von Fehlern.

2.3.1 Arbeiten mit Dozentenblättern und Zuarbeitsformularen

Das System soll den gesamten Ablauf für Dozentenblätter und Zuarbeitsformulare abbilden. Das Studienamt legt für jedes Semester die benötigten Formulare an und verknüpft sie mit den passenden Modulen und Dozierenden. Beim Anlegen erzeugt das System automatisch Token-Links, die an die jeweiligen Dozierenden übermittelt werden.

Die Dozierenden können über ihren persönlichen Link das Formular aufrufen, Daten eintragen und den Stand zunächst als Entwurf speichern. Sie haben die Möglichkeit, später zurückzukehren und Änderungen vorzunehmen. Sobald alle Informationen vollständig sind, kann das Formular endgültig eingereicht werden. Das Studienamt sieht in seiner Übersicht, welche Formulare noch im Entwurf sind, welche eingereicht wurden und welche nach Prüfung freigegeben sind. Bei Bedarf können Rückfragen gestellt und Formulare zur Überarbeitung zurückgegeben werden.

2.3.2 Stammdaten und Autofill

Viele Informationen in den Formularen wiederholen sich oder lassen sich aus bereits vorhandenen Daten ableiten. Deshalb spielt die zentrale Pflege von Stammdaten eine wichtige Rolle. Zu diesen Stammdaten gehören unter anderem die Modulnummer, der Modulname, die oder der Modulverantwortliche sowie die Standard-SWS für Vorlesungen, Seminare und Praktika. Ggf. können auch weitere Attribute wie die Zuordnung zu Studiengängen oder Modulkategorien hinterlegt werden.

Das Studienamt pflegt diese Stammdaten in einer eigenen Übersicht. Wenn ein neues Formular angelegt wird oder eine dozierende Person ein Modul im Formular auswählt, übernimmt das System die zugehörigen Stammdaten automatisch in die Eingabefelder. Diese Autofill-Funktion reduziert den Eingabeaufwand und hilft, Fehler zu vermeiden. Die Literatur zu Webformularen weist darauf hin, dass sinnvolle Voreinstellungen und automatische Ausfüllhilfen die Bearbeitung vereinfachen und die Datenqualität verbessern können.

Gleichzeitig müssen Dozierende die Möglichkeit haben, die automatisch eingetragenen Werte zu prüfen und bei Bedarf anzupassen, etwa wenn es abweichende Lehrformen gibt oder sich die SWS-Aufteilung für ein bestimmtes Semester unterscheidet. Das System soll daher klar erkennbar machen, welche Werte aus Stammdaten stammen und welche individuell geändert wurden.

2.3.3 Fristen und planungsübersicht

Für die Lehrplanung ist es wichtig, dass das Studienamt jederzeit einen klaren Überblick über alle Formulare und deren zeitliche Einordnung hat. Das System soll deshalb eine übersichtliche Darstellung bieten, in der für jede Einreichung das zugehörige Semester, das Kalenderjahr, der aktuelle Status und die wichtigsten Eckdaten sichtbar sind.

Die Zuordnung zu Sommer- und Wintersemester erfolgt automatisch auf Basis der Systemzeit des Servers. Als Annahme gilt: Die Formulare werden jeweils im vorangehenden Semester vorbereitet. Befindet sich das System also im Sommersemester eines Jahres, werden die Formulare für das kommende Wintersemester angelegt und bearbeitet; im Wintersemester werden entsprechend die Formulare für das nächste Sommersemester vorbereitet. Auf dieser Grundlage kann das System Standardfristen, zum Beispiel „Abgabe bis Ende des vorangehenden Semesters“, automatisch ableiten und anzeigen.

Zu jedem Formular werden außerdem die zugehörige Modulnummer, der Modulname und die Anzahl der geplanten Gruppen berechnet und angezeigt. So ist für das Studienamt auf einen Blick sichtbar, wie viele Veranstaltungsgruppen pro Modul für ein bestimmtes Semester geplant sind, etwa mehrere Seminar- oder Praktikumsgruppen.

Inhaltlich wird zusätzlich angenommen, dass bei Formularen mit den Veranstaltungstypen Vorlesung, Seminar oder Praktikum die in den Stammdaten hinterlegte modulverantwortliche Person automatisch als erste lehrende Person vorgeschlagen wird. Sowohl Dozierende als auch das Studienamt können weitere Lehrende hinzufügen, die Reihenfolge ändern oder Einträge anpassen. Die Kombination aus automatisch berechneten Semestern und Fristen, klarer Statusübersicht und Vorbelegung der Lehrenden soll den organisatorischen Aufwand reduzieren und typische Eingabefehler verringern.

2.3.4 Feedback und Fehlermeldungen

Damit möglichst wenige Rückfragen nötig sind, soll das System Fehler bei der Eingabe früh erkennen und verständlich anzeigen. Dazu gehört, dass Pflichtfelder nicht leer bleiben dürfen und einfache Formatprüfungen durchgeführt werden, zum Beispiel bei E-Mail-Adressen oder Zahlenfeldern. Wo sinnvoll, sollen Plausibilitätsprüfungen unterstützen, etwa bei der Summe von SWS-Angaben.

Fehlermeldungen sollen direkt am betroffenen Feld erscheinen und in einfacher, klarer Sprache formuliert sein. Nutzerinnen und Nutzer sollen sofort erkennen können, was sie korrigieren müssen. Studien zur Gestaltung von Webformularen zeigen, dass gut platzierte und verständliche Fehlermeldungen die Datenqualität erhöhen und Rückfragen deutlich verringern können.

2.4 Nicht-funktionale Anforderungen

Neben den konkreten Funktionen spielen qualitative Anforderungen eine wichtige Rolle. Sie bestimmen, wie gut das System im Alltag eingesetzt und langfristig weiterentwickelt werden kann.

2.4.1 Benutzerfreundlichkeit

Die Benutzeroberfläche soll für Dozierende ebenso wie für Mitarbeitende im Studienamt leicht verständlich und gut bedienbar sein. Dazu gehört eine klare Struktur der Formulare mit sinnvollen Abschnittsüberschriften, leicht verständlichen Feldbezeichnungen und kurzen Hilfetexten. Schaltflächen wie „Speichern“ oder „Einreichen“ sollen an konsistenten Stellen platziert sein, und die Navigation soll ohne tief verschachtelte Menüs auskommen.

2.4.2 Sicherheit

Die Anwendung muss grundlegende Sicherheitsanforderungen erfüllen, um personenbezogene Daten angemessen zu schützen. Dazu gehört eine verschlüsselte Übertragung über HTTPS. Token-Links sollten so behandelt werden, dass sie nicht unnötig in Logdateien oder Analysesystemen im Klartext auftauchen. Alle Eingaben müssen serverseitig geprüft werden, auch wenn bereits eine Validierung im Browser stattfindet. Die Zugriffsrechte zwischen Studienamt und Dozierenden sind klar zu trennen, sodass Dozierende nur ihre eigenen Formulare einsehen und bearbeiten können.

Darüber hinaus sollen typische Web-Schwachstellen vermieden werden, etwa fehlerhafte Zugriffskontrolle oder Injection-Angriffe. Also Risiken die besonders häufig auftreten sollen reduziert werden.

2.4.3 Wartbarkeit und Weiterentwicklung

Formulare und Regeln in der Lehrplanung ändern sich im Laufe der Zeit, etwa durch neue Prüfungsordnungen oder organisatorische Anpassungen. Das System soll daher so aufgebaut sein, dass Änderungen möglichst einfach vorgenommen werden können. Idealerweise sind Formularfelder und Validierungsregeln an wenigen, gut dokumentierten Stellen definiert, sodass Anpassungen nicht im ganzen Quellcode verteilt sind. Fachliche Änderungen, zum Beispiel neue Felder oder geänderte Pflichtangaben, sollen ohne komplette Neuentwicklung möglich sein.

Auch die Struktur des Quellcodes spielt eine Rolle: Eine klare Modularisierung erleichtert es späteren Entwicklerinnen und Entwicklern, sich einzuarbeiten und das System weiterzuentwickeln.

2.4.4 Transparenz und Auswertbarkeit

Schließlich soll das System eine grundlegende Transparenz über den Ablauf der Formularprozesse bieten. Wichtige Ereignisse wie das Einreichen oder Freigeben eines Formulars sollen protokolliert werden. Es soll nachvollziehbar sein, wann ein Formular zuletzt geändert wurde und durch welche Rolle.

Darüber hinaus sollen einfache Kennzahlen erhoben werden können, zum Beispiel die Anzahl der Formulare pro Semester oder die durchschnittliche Bearbeitungszeit bis zur Freigabe. Diese Informationen sind sowohl für die tägliche Arbeit im Studienamt als auch für spätere Auswertungen und Verbesserungsmaßnahmen hilfreich.

2.5 Zusammenfassung

In diesem Kapitel wurden die Anforderungen an das digitale Formularsystem zusammengefasst. Im Zentrum stehen ein lokaler, lizenzfreier Betrieb, ein einfaches Rollen- und Zugriffskonzept mit Login für das Studienamt und Token-Links für Dozierende, die zentrale Pflege von Stammdaten mit Autofill-Funktionen sowie klare Fristen- und Statusinformationen. Ergänzend wurden qualitative Anforderungen wie Benutzerfreundlichkeit, Sicherheit, Wartbarkeit und Transparenz beschrieben. Diese Anforderungen bilden die Grundlage für die in den folgenden Kapiteln betrachtete technische Umsetzung und Evaluation.

Kapitel 3

Stand der Technik

Dieses Kapitel beschreibt den Stand der Technik, in den die in dieser Arbeit umgesetzten Prototypen einzuordnen sind. Im Mittelpunkt stehen Web-Technologien für formularbasierte Anwendungen sowie verbreitete Architekturbausteine für Backend, Schnittstellen und Datenhaltung. Die Darstellung folgt dabei der Logik der späteren Implementierung: Zuerst werden Frontend-Grundlagen erläutert, danach Framework-Ökosysteme und Formularframeworks, anschließend Backend und API-Stil, und abschließend Persistenz und Deployment.

3.1 Frontend-Grundlagen: HTML, CSS und JavaScript

Abbildung 3.1: Standard Technologiern der Web

Formularbasierte Webanwendungen beruhen unabhängig vom verwendeten Framework auf drei Basistechnologien: HTML beschreibt die Struktur und Semantik, CSS gestaltet die Darstellung und JavaScript implementiert die Interaktion. Auch moderne UI-Frameworks erzeugen letztlich HTML, wenden CSS-Regeln an und führen JavaScript aus. Für die Einordnung der in dieser Arbeit implementierten Prototypen ist es daher sinnvoll, diese Grundlagen kurz und in ihrem Beitrag zu Formularsystemen zu erläutern.

3.1.1 HTML: Struktur, Semantik und Formularelemente

HTML ist die Kern-Auszeichnungssprache des Webs. Der maßgebliche Referenzstandard ist der *HTML Living Standard*, der von der WHATWG gepflegt wird [66]. Für Formularanwendungen ist HTML besonders wichtig, weil es Formularelemente und ihre Beziehungen standardisiert. Dazu zählen das Formular selbst (`<form>`) sowie Eingabefelder wie `<input>`, `<textarea>` und `<select>`. Ein zentraler Punkt ist die semantische Zuordnung von

Beschriftungen zu Feldern über `<label>` und `id/for`. Diese Zuordnung ist nicht nur eine Frage der „Sauberkeit“ des Markups, sondern hat unmittelbare Auswirkungen auf Bedienbarkeit und Verständlichkeit, weil Assistenztechnologien Feldnamen, Hilfetexte und Fehlermeldungen nur dann zuverlässig vermitteln können, wenn die Verknüpfungen korrekt gesetzt sind.

In klassischen Formularen reichen HTML-Elemente häufig aus, um robuste und barrierearme Eingaben zu ermöglichen. Sobald jedoch komplexere Widgets entstehen, etwa Autocomplete- oder Combobox-Komponenten, stößt reines HTML oft an Grenzen, weil das Interaktionsverhalten (Tastatursteuerung, Fokusführung, Screenreader-Ankündigungen) präzise modelliert werden muss. In solchen Fällen werden ergänzend ARIA-Rollen und -Attribute verwendet, die im WAI-ARIA-Standard beschrieben sind [57]. Für typische Interaktionsmuster stellt der W3C mit den *WAI-ARIA Authoring Practices* zudem konkrete Umsetzungsleitlinien bereit, die definieren, welche Rollen, Zustände und Tastaturinteraktionen erwartet werden [62]. Damit bildet HTML die semantische Grundlage, während ARIA in Spezialfällen das Verhalten strukturiert ergänzt. [39]

3.1.2 CSS: Darstellung, Layout und Zustände im Formular

CSS beschreibt die visuelle Präsentation von Webinhalten und beeinflusst damit wesentlich, wie gut Formulare verständlich und effizient bedienbar sind. Als verlässlicher Überblick über den aktuellen Stand der CSS-Module dient der W3C-Bericht *CSS Snapshot 2025* [58]. Im Formular-Kontext betrifft CSS insbesondere die Lesbarkeit (Typografie, Abstände), die Strukturierung (z. B. Grid-/Flex-Layout für Abschnitte) sowie die responsive Darstellung auf verschiedenen Geräten.

Darüber hinaus spielt CSS eine zentrale Rolle bei der Darstellung von Zuständen. Formulare benötigen klare, wiedererkennbare Visualisierungen für Fokus, Pflichtfelder, Fehler und Hilfetexte. Ein sichtbarer Fokus ist für Tastaturnutzende entscheidend, und eine konsistente Fehlerdarstellung reduziert Rückfragen, weil Eingabefehler frühzeitig und verständlich erkennbar sind. Für die praktische Umsetzung ist es zudem wichtig, dass Fehlzustände nicht ausschließlich über Farbe kommuniziert werden, sondern durch zusätzliche Hinweise wie Icons, Text oder Rahmen eindeutig werden. CSS ist damit nicht „nur Design“, sondern ein technischer Baustein zur Unterstützung von Usability und Barrierearmut. [38]

3.1.3 JavaScript: Interaktion, Validierung und Kommunikation

JavaScript ist die Programmiersprache des Webs. In Formularsystemen wird JavaScript genutzt, um dynamische Interaktionen umzusetzen, etwa das Ein- und Ausblenden von Formularbereichen, die Berechnung von abgeleiteten Werten oder clientseitige Validierungslogik. Gerade in prozessualen Formularen ist JavaScript außerdem wichtig, um Statusinformationen anzuzeigen und Nutzenden Rückmeldungen zu geben, ohne dass jede Aktion einen vollständigen Seitenreload auslöst. [40].

Ein weiterer Kernaspekt ist die Kommunikation mit dem Backend. In modernen Webanwendungen werden Daten typischerweise asynchron übertragen, um das Ausfüllen von Formularen flüssig zu halten und Zwischenspeichern zu ermöglichen. Für diese Kommunikation ist in vielen Implementierungen die `fetch`-Schnittstelle relevant, deren Verhalten im Fetch-Standard beschrieben ist [65]. Damit verbindet JavaScript die Eingabelogik im Browser mit dem REST-orientierten Backend und ist zugleich der Ort, an dem Validierungs- und Komfortfunktionen (z. B. Autofill oder Plausibilitätsprüfungen) implementiert werden.

Zusammenfassend bilden HTML, CSS und JavaScript die technische Basis für beide Varianten dieser Arbeit. Der Unterschied zwischen hardcodiertem und framework-basiertem Ansatz liegt weniger in den Grundtechnologien selbst, sondern darin, ob Struktur und Regeln überwiegend manuell im Code gepflegt werden oder stärker aus Schemata und wiederverwendbaren Komponenten abgeleitet werden. [40].

3.2 UI-Frameworks und Build-Tooling

Während HTML, CSS und JavaScript die technische Basis jeder Webanwendung bilden, werden bei größeren Projekten häufig UI-Frameworks eingesetzt, um die Entwicklung zu strukturieren. In formularbasierten

Anwendungen ist dies besonders relevant, weil Formulare mit wachsender Komplexität typischerweise aus wiederkehrenden Bausteinen bestehen (z. B. Abschnitte, wiederholbare Gruppen, dynamische Tabellenzeilen oder Validierungs- und Fehlermeldungslogik). UI-Frameworks liefern hierfür ein Komponentenmodell und etablierte Muster zur Zustandsverwaltung, wodurch sich Formulare konsistenter und wartbarer entwickeln lassen.

3.2.1 React als verbreitete Basis für komponentenbasierte UIs

Abbildung 3.2: React

React ist eine Bibliothek zur Erstellung von Benutzeroberflächen, die UIs als Komposition wiederverwendbarer Komponenten modelliert [49]. Für Formulare ist dieses Komponentenmodell besonders nützlich, weil es eine klare Trennung zwischen einzelnen Eingabeteilen erlaubt und UI-Zustände (z. B. Entwurf vs. eingereicht, Fehlermeldungen, dynamische Feldsichtbarkeit) systematisch abbildet. In der Praxis haben sich deshalb viele Formularbibliotheken und schema-getriebene Renderer im React-Ökosystem etabliert oder bieten eine React-Integration als primären Zielpfad.

Im Kontext dieser Arbeit ist React zudem relevant, weil es einen gut dokumentierten, weit verbreiteten Standardstack darstellt, auf den sich die Framework-Variante der Prototypen stützen kann. Dadurch lassen sich technische Entscheidungen zu Rendering, Komponentenaufteilung und Zustandslogik nachvollziehbar begründen und mit dem Stand der Technik vergleichen.

3.2.2 Next.js als React-Framework für produktionsnahe Webanwendungen

Abbildung 3.3: Next.js

Next.js ist ein Framework, das auf React aufbaut und typische Anforderungen moderner Webanwendungen als integrierte Plattformfunktionen bereitstellt, insbesondere Routing und unterschiedliche Rendering-Strategien [54]. Für formularbasierte Systeme ist Next.js vor allem dann relevant, wenn die Anwendung über reine Formulareingabe hinaus eine vollständige Webplattform darstellen soll, etwa mit Login-Mechanismen, serverseitiger Datenbereitstellung, Performance-Optimierungen oder einem konsolidierten Deployment.

In dieser Arbeit wird Next.js als Stand-der-Technik-Referenz eingeordnet, weil viele React-basierte Enterprise-Anwendungen und Formularsysteme in der Praxis in solchen Rahmenframeworks betrieben werden. Für Prototypen kann Next.js vorteilhaft sein, ist aber nicht zwingend erforderlich, wenn bewusst ein schlankeres Setup gewählt wird.

3.2.3 Angular als Alternative mit umfassendem Framework-Anspruch

Abbildung 3.4: Angular

Angular ist ein umfassendes Web-Framework, das neben dem UI-Komponentenmodell auch Tooling, Strukturkonventionen und ein ausgeprägtes Forms-Konzept bereitstellt [17]. Insbesondere das Angular-Forms-Modell ist in vielen Organisationen ein etablierter Standard, weil Validierung, Formzustände und Datenbindung sehr systematisch abgebildet werden. Aus Sicht schema-getriebener Formulare ist Angular zudem relevant, weil viele Formrenderer Mehrfach-Integrationen anbieten und Angular damit eine typische Zielplattform neben React darstellt.

Für diese Arbeit dient Angular daher vor allem als Vergleichsrahmen: Die grundlegende Idee deklarativer Formularmodelle tritt in unterschiedlichen Framework-Ökosystemen auf, die konkrete Integration hängt jedoch stark von der jeweiligen Komponenten- und State-Architektur ab.

3.3 Formularframeworks: schema-getriebene Ansätze

Ein zentraler Trend in der Formularentwicklung besteht darin, Formulare nicht vollständig manuell zu implementieren, sondern aus formalen Beschreibungen abzuleiten. Das reduziert Redundanz und erleichtert Änderungen, weil sich Struktur und Validierungsregeln zentral pflegen lassen. In der Praxis wird dabei häufig zwischen zwei Welten unterschieden: JSON-basierte Formulare, die auf *JSON Schema* aufbauen, und RDF-basierte Formulare, die Constraints über *SHACL* ausdrücken.

JSON Schema ist ein Standard, um Struktur, Datentypen und Validierungsregeln für JSON-Dokumente zu beschreiben (z. B. Pflichtfelder, Datentypen, Wertebereiche oder Muster) [30]. SHACL ist eine W3C-Empfehlung, um RDF-Graphen gegen deklarative Constraints (Shapes) zu validieren [61]. Beide Standards ermöglichen, dass „gültige Daten“ formal definiert werden können und nicht nur implizit in UI-Logik oder Backend-Code versteckt sind. Genau diese Trennung von Datenmodell, Validierung und Darstellung ist die Grundlage schema-getriebener Formularframeworks.

Im Folgenden werden fünf Frameworks eingeordnet, die in dieser Arbeit als Stand-der-Technik-Referenzen betrachtet wurden. Die Auswahl deckt sowohl JSON/JSON-Schema-orientierte Lösungen als auch RDF/SHACL-orientierte Ansätze ab und ist deshalb geeignet, die beiden Prototypvarianten dieser Arbeit fachlich zu verorten.

3.3.1 RDForms (RDF-orientierte Formularerstellung)

Abbildung 3.5: RDForms frameworks

RDForms ist ein Framework, das die Bearbeitung von RDF-Daten über Webformulare adressiert. Der zentrale Gedanke ist, dass RDF-Ressourcen und deren Eigenschaften nicht als „klassische“ Formularfelder betrachtet werden, sondern als graphförmige Datenstrukturen, die über geeignete UI-Bausteine editierbar gemacht werden. In der RDForms-Dokumentation werden dazu *Templates* beschrieben, die festlegen, wie RDF-Strukturen im UI dargestellt und bearbeitet werden [48].

Für die praktische Nutzung bedeutet das: Ein Template definiert, welche Eigenschaften einer Ressource angezeigt werden, welche Eingabetypen passend sind (z. B. Text, Auswahl, Wiederholungen) und wie mit Mehrfachwerten (z. B. mehrere Lehrende, mehrere Rollen, mehrere Zugehörigkeiten) umgegangen wird. Der Vorteil liegt darin, dass RDF-spezifische Eigenschaften wie IRIs, Sprach-Tags oder die Nutzung von Vokabularen (Ontologien) direkt berücksichtigt werden können. Damit eignet sich RDForms besonders für Szenarien, in denen Daten ohnehin als Linked Data vorliegen oder später in semantische Infrastrukturen integriert werden sollen[48].

Im Kontext eines Hochschul-Formularsystems ist RDForms deshalb interessant, weil viele Hochschuldaten grundsätzlich in semantischen Modellen abbildbar sind (z. B. Module, Personen, Rollen, Studiengänge). Gleichzeitig ist der Ansatz für klassische Verwaltungsformulare oft nur dann effizient, wenn RDF und entsprechende Vokabulare bereits etabliert sind. Andernfalls entsteht zusätzlicher Modellierungsaufwand, weil das Datenmodell zuerst als RDF-Graph gedacht und gepflegt werden muss. RDForms bildet somit vor allem einen Referenzpunkt für semantische Integrationspfade und weniger eine „Standardwahl“ für JSON-zentrierte Webformulare. Konkrete Beispiele für RDForms-Formulare zeigen Abbildung 3.6 und Abbildung 3.7.

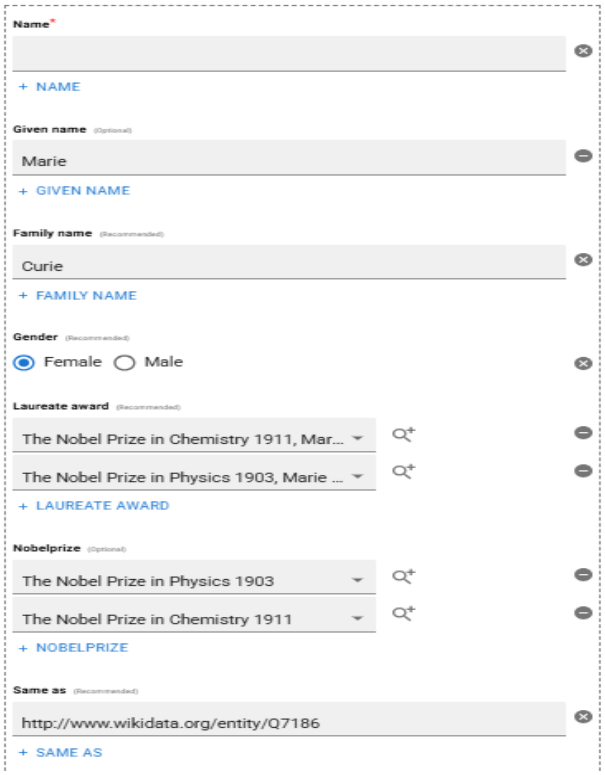

Abbildung 3.6: RDForms Beispiel Teil 1

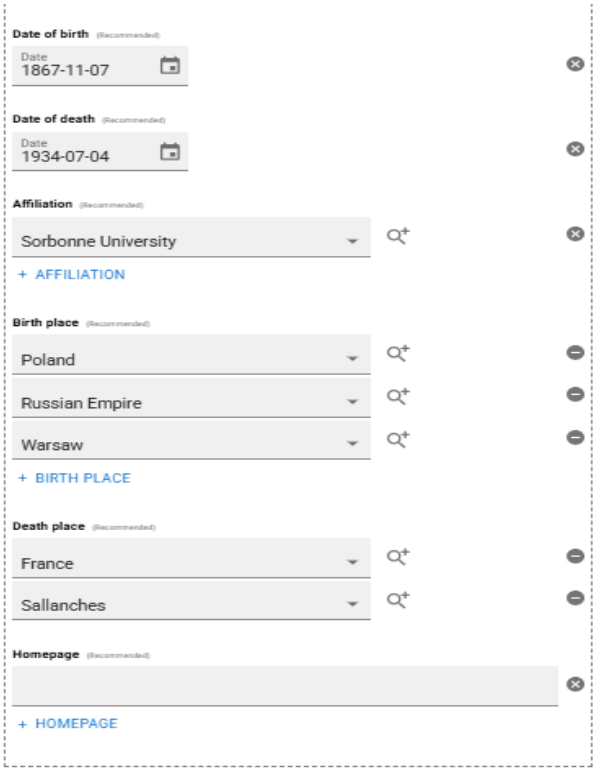

Abbildung 3.7: RDForms Beispiel Teil 2

3.3.2 `shacl-form` (Formulare aus SHACL-Shapes als Web Component)

Abbildung 3.8: shacl-form

`shacl-form` ist ein SHACL-basierter Ansatz, der als Web Component umgesetzt ist. Die Idee ist, dass ein SHACL-Shape die zulässige Struktur und die Constraints eines Datensatzes beschreibt und daraus automatisiert eine Formularoberfläche abgeleitet wird [53]. Damit wird SHACL nicht nur zur nachträglichen Validierung verwendet, sondern als primäre Spezifikation, aus der das Formular entsteht.

Der praktische Vorteil dieses Ansatzes liegt in der engen Kopplung zwischen Modell und Validierung: Wenn ein Shape Kardinalitäten, Datentypen oder erlaubte Wertebereiche beschreibt, kann das Formular diese Regeln direkt sichtbar machen, Pflichtfelder kennzeichnen und ungültige Eingaben frühzeitig erkennen. Zusätzlich sind Web Components technologisch relativ neutral: Sie lassen sich in „plain HTML“ einbetten, aber auch in framework-basierte Anwendungen integrieren, solange diese Web Components unterstützen.

Für diese Arbeit ist `shacl-form` vor allem aus zwei Gründen relevant. Erstens bildet es eine sehr konsequente Form schema-getriebener UI-Generierung im RDF-Ökosystem ab und ist damit ein Gegenstück zu JSON-Schema-basierten Lösungen. Zweitens zeigt es, dass sich formularbasierte Datenerfassung direkt an ein RDF/SHACL-Qualitätssystem koppeln lässt, was insbesondere bei späterer Integration in Triple-Stores und semantische Auswertungen vorteilhaft sein kann. Gleichzeitig gilt auch hier: Wenn der operative Prozess primär JSON-basiert ist (z. B. REST-API und JSON-Storage), wäre ein durchgehend SHACL-zentrierter Stack eine größere Architekturentscheidung, die zusätzliche Infrastruktur und Know-how voraussetzt.

Konkrete Beispiele für shacl-form-Formulare zeigen Abbildung 3.9 und Abbildung 3.10.

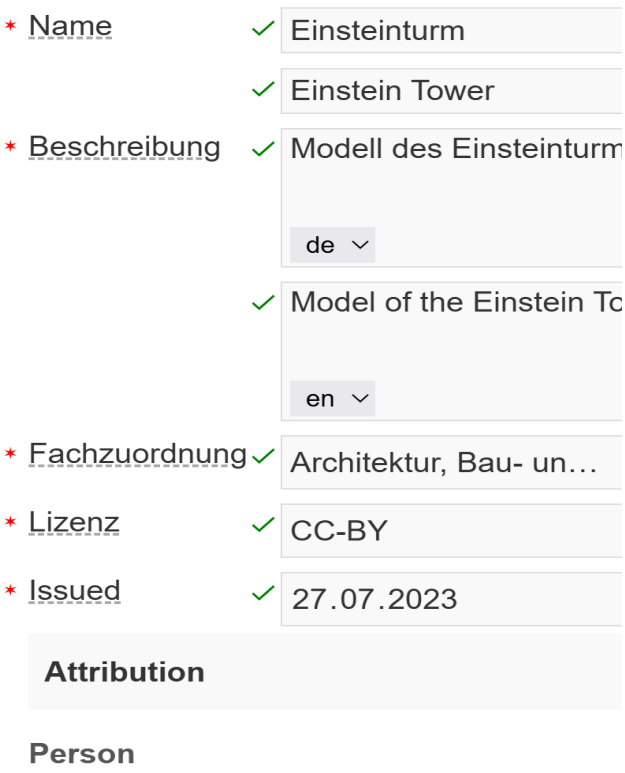

Abbildung 3.9: shacl-form Beispiel Teil 1

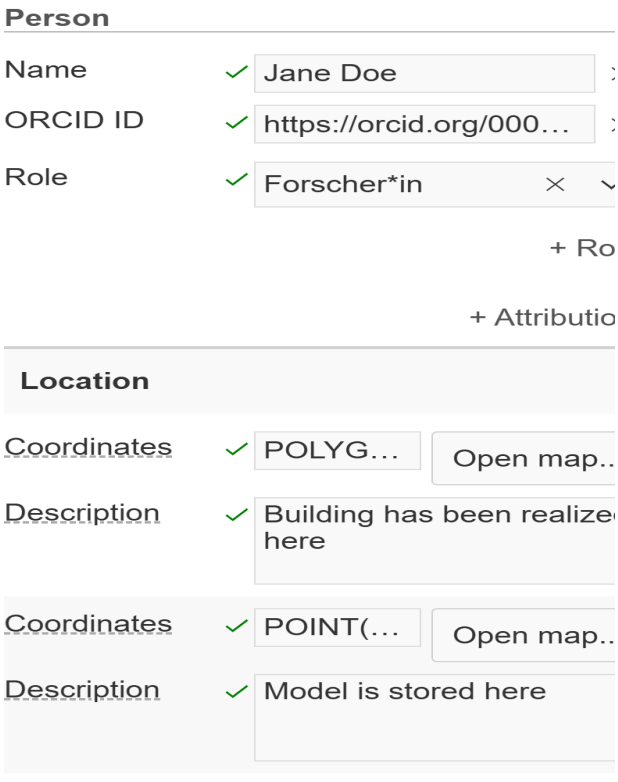

Abbildung 3.10: shacl-form Beispiel Teil 2

3.3.3 SurveyJS (JSON-definierte Formulare mit Fokus auf Surveys)

Abbildung 3.11: SurveyJS

SurveyJS stellt eine Bibliothek bereit, die Formulare aus JSON-Definitionen rendert. Die Dokumentation beschreibt SurveyJS als Form Library, mit der dynamische Formulare bzw. Surveys konfiguriert und in Webanwendungen eingebettet werden können [52]. Der Schwerpunkt liegt dabei typischerweise auf mehrseitigen Fragebögen, logischen Abhängigkeiten zwischen Fragen und einer strukturierten Auswertung von Ergebnissen.

Aus technischer Sicht ist SurveyJS interessant, weil es Formularinhalte als Konfiguration beschreibt: Felder, Seiten, Sichtbarkeitsregeln, Validierungen und teilweise auch UI-Verhalten werden in JSON abgebildet und im Renderer umgesetzt. Damit lassen sich Formulare ohne tiefgreifende Änderungen am UI-Code variieren. In Organisationskontexten ist zusätzlich relevant, dass SurveyJS häufig zusammen mit einem visuellen „Creator“ eingesetzt wird, um Formulare interaktiv zu erstellen und als JSON zu speichern (Low-Code-Charakter).

Für die in dieser Arbeit betrachteten Hochschul-Workflows ist SurveyJS ein plausibler Referenzpunkt, weil es zeigt, wie stark die formularbasierte Datenerhebung bereits konfigurationsgetrieben erfolgen kann. Allerdings ist SurveyJS konzeptionell häufig eher auf „Befragung“ als auf „Verwaltungsworkflow“ optimiert. Prozesse mit Rollen, Statuslogik, Rückfragen, Freigaben und längerfristigen Entwürfen müssen meist stärker in der umgebenden Anwendung modelliert werden. SurveyJS liefert also primär die UI- und Eingabestruktur, nicht automatisch den vollständigen Workflow.

Konkrete Beispiele für SurveyJS-Formulare zeigen Abbildung 3.12 und Abbildung 3.13. Abbildung 3.14 und Abbildung 3.15.

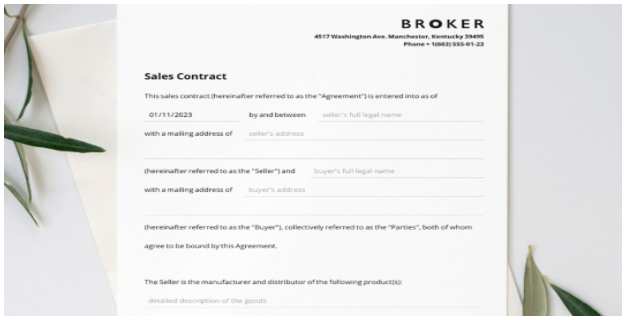

Abbildung 3.12: SurveyJS Beispiel 1

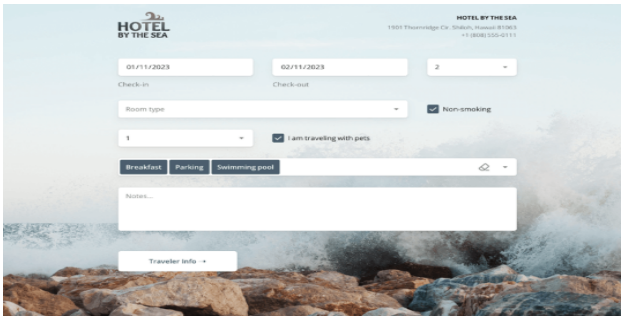

Abbildung 3.13: SurveyJS Beispiel 2

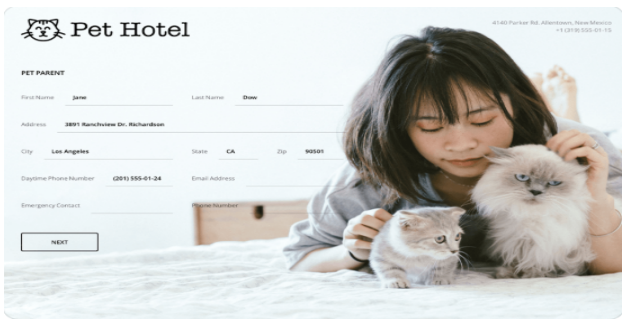

Abbildung 3.14: SurveyJS Beispiel 3

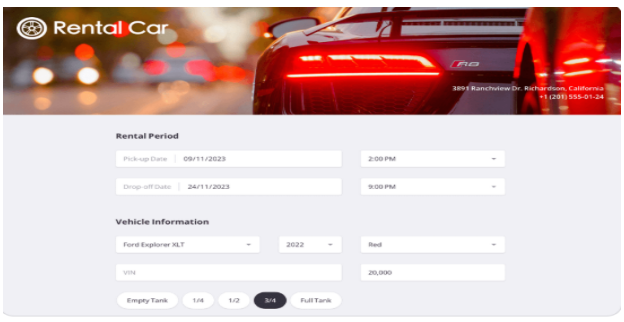

Abbildung 3.15: SurveyJS Beispiel 4

3.3.4 JSON Forms (JSON Schema + UI Schema, besonders relevant für React)

Abbildung 3.16: JsonForms

JSON Forms ist ein schema-getriebenes Framework, das Form-UIs aus einem JSON Schema (Datenmodell und Constraints) und einem separaten UI Schema (Layout und Darstellungsregeln) generiert [26]. Der Kernnutzen liegt in der klaren Arbeitsteilung: Das JSON Schema beschreibt, *welche* Daten erlaubt sind, während das UI Schema beschreibt, *wie* diese Daten im Formular angeordnet und präsentiert werden sollen.

In der Praxis ist das für Verwaltungsformulare besonders hilfreich. Fachliche Änderungen betreffen oft Pflichtfelder, Datentypen oder zusätzliche Attribute (Datenmodell). Layoutänderungen betreffen dagegen Gruppierungen, Reihenfolgen, Abschnittsüberschriften oder das Ein-/Ausblenden bestimmter Bereiche (Darstellung). Wenn beides getrennt gepflegt werden kann, sinkt der Änderungsaufwand und die Gefahr, dass Logik an mehreren Stellen inkonsistent wird. Außerdem unterstützt ein JSON-Schema-basierter Ansatz eine konsistente Validierung, weil Constraints aus dem Schema abgeleitet werden können, anstatt in „Handcode“ verteilt zu sein.

Für diese Arbeit ist JSON Forms deshalb ein besonders geeigneter Stand-der-Technik-Referenzpunkt: Der framework-basierte Prototyp kann JSON Schema als zentrale Quelle nutzen, und die React-Integration passt zu einem modernen UI-Stack. Damit lässt sich in der späteren Evaluation auch nachvollziehbar prüfen, ob ein schema-getriebenes Rendering tatsächlich Änderbarkeit und Datenqualität verbessert, ohne dass die UI-Entwicklung vollständig „unflexibel“ wird.

Konkretes Beispiel für JsonForms-Formular zeigt Abbildung 3.17.

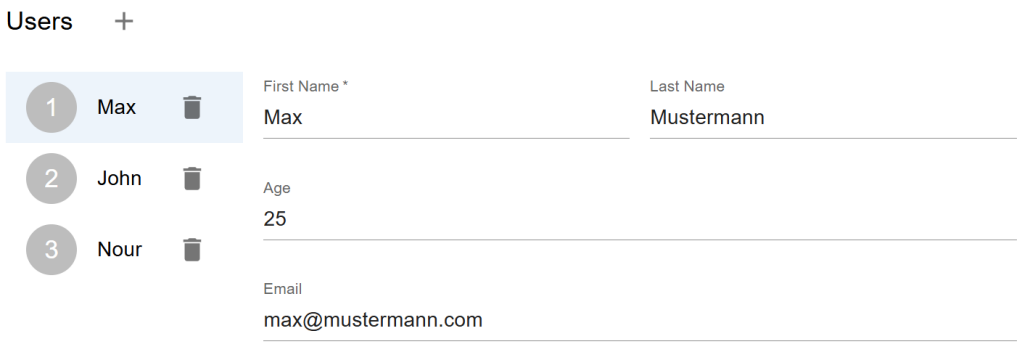

Abbildung 3.17: JsonForms Beispiel

3.3.5 Form.io (Plattformansatz: Form Builder, JSON-Modelle, Backend-Integration)

Abbildung 3.18: Form.io

Form.io steht exemplarisch für Plattformlösungen, die Formularentwicklung als konfigurationsgetriebenen Prozess behandeln. Die offizielle Dokumentation beschreibt Form.io als System, in dem Formulare modelliert und in Anwendungen eingebettet werden können [14]. In vielen Einsätzen ist Form.io nicht nur Renderer, sondern Teil eines größeren Ökosystems aus Form Builder, Datenverwaltung und Integrationsmechanismen.

Technisch ist der Ansatz relevant, weil er die Idee „Form as a Model“ konsequent umsetzt: Formularstruktur, Validierung und teilweise auch Datenflüsse werden als modellierte Artefakte verstanden, die sich versionieren und wiederverwenden lassen. Das kann Änderungen sehr schnell machen, insbesondere wenn Fachbereiche Formulare häufig anpassen. Gleichzeitig entstehen typische Abwägungen: Plattformansätze bringen zusätzliche Betriebs- und Governance-Fragen mit sich (z. B. Rollenverwaltung, Berechtigungen, Datenhaltung, Schnittstellen, Updatezyklen), die je nach Self-Hosting-Anforderung und Datenschutzkontext bewusst bewertet werden müssen.

Für den Hochschulkontext ist Form.io daher vor allem als Referenz für den Markttrend relevant: Formsysteme werden zunehmend als Plattformbausteine verstanden, nicht nur als UI-Komponenten. Für diese Bachelorarbeit dient Form.io entsprechend als Vergleichs- und Einordnungsrahmen, während der implementierte Prototyp bewusst schlanker bleibt.

Konkretes Beispiel für Form-IO-Formular zeigt Abbildung 3.19.

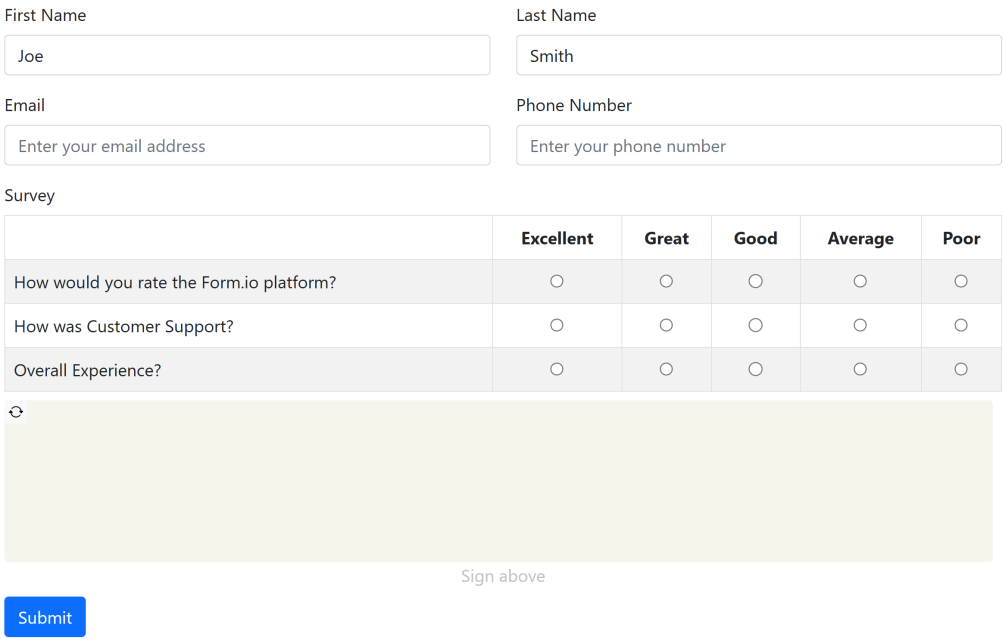

Abbildung 3.19: JsonForms Beispiel

3.3.6 Zusammenfassende Einordnung der Frameworks

Die fünf betrachteten Frameworks lassen sich vor allem entlang zweier Dimensionen einordnen. Die erste Dimension ist das zugrundeliegende Daten- und Validierungsmodell: RDForms und `shacl-form` arbeiten RDF-/SHACL-zentriert, während SurveyJS, JSON Forms und Form.io primär JSON-basiert sind und sich gut mit JSON Schema verbinden lassen [30, 61]. Die zweite Dimension betrifft den Grad, in dem ein Framework „nur“ UI-Rendering liefert oder darüber hinaus als Plattform fungiert: JSON Forms ist stark renderer-orientiert, SurveyJS fokussiert die Formular- und Survey-Logik, während Form.io stärker Plattformfunktionen bündelt.

Für die Prototypen dieser Arbeit ist diese Einordnung wichtig, weil sie erklärt, warum ein JSON-Schema-zentrierter Ansatz im framework-basierten Prototyp naheliegt, während RDF/SHACL-orientierte Frameworks als Referenz für Interoperabilität und spätere semantische Integration dienen.

3.4 Backend und API-Stil: Node.js und REST über HTTP

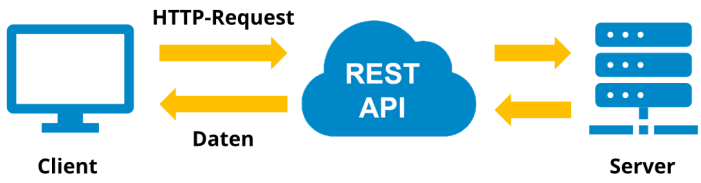

Abbildung 3.20: REST-API

Auf der Serverseite ist Node.js eine verbreitete Laufzeitumgebung, um JavaScript als Backend-Technologie zu nutzen. Die offizielle Node.js-Dokumentation beschreibt die bereitgestellten Standardmodule und die typische Nutzung als HTTP-Server sowie für Dateizugriffe [43]. Für einen prototypischen Formularworkflow ist Node.js besonders geeignet, weil sich REST-Endpunkte schlank implementieren lassen und Persistenz zunächst dateibasiert umgesetzt werden kann.

Als Schnittstellenstil wird in dieser Arbeit ein REST-orientiertes API-Verständnis zugrunde gelegt. REST wurde als Architekturstil von Fielding beschrieben und betont u. a. eine einheitliche Schnittstelle und zustandslose Kommunikation [12]. Die technische Grundlage bildet HTTP; dessen Semantik ist in RFC 9110 standardisiert [21]. In der Praxis bedeutet das für Formularsysteme: Formulare und Einreichungen werden als Ressourcen modelliert, und typische HTTP-Methoden und Statuscodes werden genutzt, um Lesen, Erstellen, Aktualisieren und Fehlerfälle klar zu kommunizieren.

3.5 Datenhaltung: RDF/Turtle sowie JSON-Dateispeicher

Für Prototypen werden häufig leichtgewichtige Persistenzformen gewählt, die ohne zusätzliche Datenbankinfrastruktur auskommen. In dieser Arbeit werden drei Ansätze eingeordnet: RDF/Turtle-Dateien, eine zentrale JSON-Datei sowie eine dateibasierte Ablage pro Datensatz.

RDF ist ein Standard zur Repräsentation von Daten als Graph; die Grundlagen sind in den W3C-Empfehlungen zu RDF und Turtle beschrieben [59, 60]. Für diese Arbeit ist RDF/Turtle vor allem als interoperables Export- und Integrationsformat relevant, weil RDF sich gut mit semantischen Hochschul- und Forschungsdaten verbinden lässt.

JSON ist als leichtgewichtiges Austauschformat in RFC 8259 standardisiert [20]. Eine einfache Persistenzform ist eine zentrale Datei (z. B. `db.json`), in der Einreichungen als Liste gespeichert werden. Ein zweiter, häufig praxisnaher Ansatz ist die Ablage einer JSON-Datei pro Vorgang oder pro Person (z. B. `jfs.json` als Prinzip

eines Dateistores), um Daten stärker zu trennen und Konflikte zu reduzieren. Beide Varianten sind für prototypische Systeme schnell umsetzbar und gut nachvollziehbar, unterscheiden sich aber hinsichtlich Nebenläufigkeit und Zugriffstrennung.

3.6 Deployment: Containerisierung mit Docker und Podman

Abbildung 3.21: Podman and Docker

Für Self-Hosting und reproduzierbaren Betrieb werden Anwendungen häufig containerisiert. Docker beschreibt Container als isolierte Prozesse, die in Images paketiert werden und konsistent über Umgebungen hinweg laufen [4]. Podman ist eine alternative Container-Engine, die ohne zentralen Daemon auskommt und häufig als sicherheitsorientierte Variante im Linux-Betrieb genutzt wird [45]. In der Praxis können Frontend und Backend getrennt als Services betrieben werden, was Wartung und Deployment in Hochschulumgebungen vereinfacht.

3.7 Zusammenfassung

Der Stand der Technik lässt sich für die Ziele dieser Arbeit wie folgt zusammenfassen: Webformulare beruhen auf HTML, CSS und JavaScript [66, 58]. Für größere Anwendungen sind UI-Frameworks wie React/Next.js oder Angular verbreitet, und Tooling wie Vite unterstützt einen reproduzierbaren Build-Prozess [49, 54, 17]. Schema-getriebene Formularframeworks nutzen JSON Schema oder SHACL als formale Grundlage und reichen von JSON-orientierten Ansätzen (SurveyJS, JSON Forms, Form.io) bis zu semantisch orientierten Ansätzen (RDForms, `shacl-form`) [30, 61, 48, 53, 52, 26, 14]. Backendseitig ist Node.js als schlanke Serverlaufzeit etabliert, während REST über HTTP einen verbreiteten Schnittstellenstil liefert [43, 12, 21]. Für die Persistenz werden im Prototyp pragmatische dateibasierte Varianten (JSON und RDF/Turtle) genutzt [20, 59, 60]. Containerisierung mit Docker oder Podman unterstützt schließlich Self-Hosting und reproduzierbare Deployments [4, 45].

Kapitel 4

Architektur und Design

Dieses Kapitel beschreibt die Zielarchitektur des digitalen Formularsystems und erläutert die wesentlichen Entwurfsentscheidungen. Im Mittelpunkt steht der Vergleich zweier Prototypvarianten, die denselben fachlichen Prozess abbilden (Dozentenblatt und Zuarbeitsformular), sich jedoch in Umsetzung und Datenhaltung unterscheiden.

Die erste Variante (im Folgenden *Standardvariante*) setzt das Frontend klassisch mit HTML, CSS und JavaScript um. Die erfassten Formulardaten werden serverseitig in ein RDF-Datenmodell überführt und als Turtle-Dateien (`.ttl`) persistiert. Die zweite Variante (im Folgenden *Framework-Variante*) realisiert das Frontend als React-Anwendung mit schema-getriebenem Rendering auf Basis von JSON Schema und JSON Forms. Die Formulardaten werden dabei als JSON gespeichert, wobei pro Datensatz separate JSON-Dateien im Dateisystem abgelegt werden.

Beide Varianten verfolgen bewusst eine schlanke, self-hosting-fähige Architektur: Browser-Frontend, Node.js/Express-Backend und dateibasierte Persistenz. Dadurch lassen sich die Varianten fair vergleichen, weil der fachliche Umfang gleich bleibt und die Unterschiede gezielt auf UI-Erzeugung, Validierungsmechanismen und Datenrepräsentation zurückgeführt werden können [13, 41].

4.1 Zielarchitektur: Schichten, Schnittstellen und Verantwortlichkeiten

Unabhängig von der Variante ist das System als klassische Webanwendung strukturiert: Die Interaktion erfolgt im Browser, die Verarbeitung und Persistenz übernimmt ein Server. Die Kommunikation zwischen Frontend und Backend erfolgt über HTTP-Requests, wobei die Semantik von HTTP (Methoden, Statuscodes, Ressourcen) als Grundlage dient [41, 13].

Das Frontend ist ausschließlich für Darstellung und Eingabe zuständig: Formulare werden ausgefüllt, Zwischenspeichern und Feedback (z. B. Validierungs- und Statushinweise) werden angezeigt. Das Backend stellt Endpunkte bereit, nimmt Daten entgegen, prüft sie (mindestens grundlegende Plausibilität und Format), und schreibt sie in die jeweilige Persistenzform.

Diese Trennung reduziert Kopplung: Änderungen an der UI-Technologie sollen möglich sein, ohne die Serverlogik komplett neu zu entwerfen. Gleichzeitig bleibt klar, an welcher Stelle Datenformate (RDF/Turtle oder JSON) festgelegt werden und wo Validierung „verbindlich“ sein muss (spätestens serverseitig).

Ein besonderes Augenmerk liegt auf Formular-Usability und Barrierearmut. Für Webformulare sind korrekt verknüpfte Labels, verständliche Fehlermeldungen und klare Anweisungen zentral; hierfür dienen die WCAG 2.2 sowie die WAI-Formular-Tutorials als maßgebliche Referenzen [64, 15, 37].

4.2 Standardvariante: HTML/CSS/JavaScript und RDF/Turtle-Persistenz

4.2.1 Komponentenübersicht und Datenfluss

Die Standardvariante besteht aus statischen HTML-Seiten mit CSS-Layout und JavaScript-Logik im Browser. Formulardaten werden nach dem Ausfüllen per HTTP an das Backend übertragen. Serverseitig werden die eingehenden Werte in RDF-Strukturen abgebildet und anschließend als Turtle serialisiert. Turtle ist eine kompakte Textsyntax, mit der ein RDF-Graph vollständig beschrieben werden kann [46, 47].

Abbildung 4.1 zeigt die Komponenten dieser Variante als UML-Komponentendiagramm.

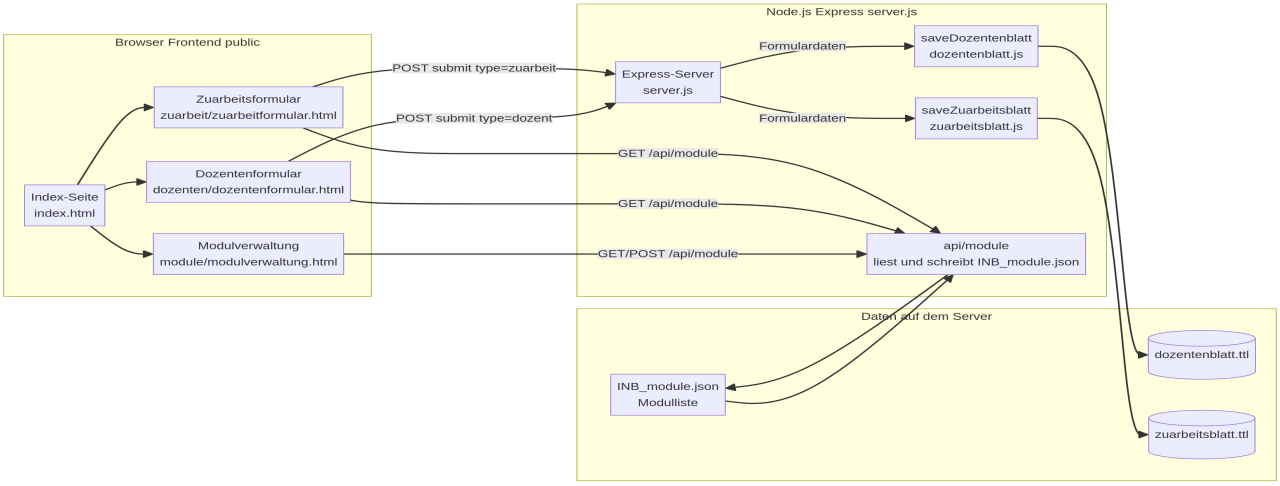

Abbildung 4.1: Architektur der Standardvariante (HTML/CSS/JS im Frontend, Node/Express im Backend, RDF/Turtle-Persistenz).

4.2.2 Backend-Design mit Node.js und Express

Das Backend wird als schlanker Webserver umgesetzt, der Endpunkte für das Speichern der Formulare und für das Laden bzw. Speichern von Modulstammdaten bereitstellt. Node.js dient dabei als Laufzeitumgebung; Express liefert ein minimales, weit verbreitetes Server-Framework, inklusive Middleware-Konzept und Routing [19, 7, 50].

Statische Dateien (HTML, CSS, JavaScript) werden über Express ausgeliefert. Für eingehende Formulardaten wird Request-Parsing verwendet, sodass sowohl URL-encodete Form-Submits als auch JSON-Payloads verarbeitet werden können [8, 7]. Damit bleibt das Backend bewusst „thin“: Es übernimmt nicht die komplette UI-Logik, sondern konzentriert sich auf Annahme, Transformation und Persistenz.

4.2.3 Persistenz als RDF-Graph und Turtle-Dateien

Die zentrale Entwurfsentscheidung dieser Variante ist die Speicherung der Formulardaten als RDF. RDF modelliert Informationen als Tripel aus Subjekt, Prädikat und Objekt und erlaubt es, Daten als Graph auszudrücken [46]. Für die vorliegende Aufgabenstellung ist das interessant, weil sich Formulardaten später leichter semantisch anreichern und mit anderen Datenquellen verknüpfen lassen, z. B. über Ontologien oder über nachgelagerte Graph-Auswertungen.

Die Serialisierung erfolgt als Turtle-Datei (`.ttl`) [47]. Praktisch bedeutet das: Jede Einreichung erzeugt RDF-Ressourcen für das jeweilige Formular sowie für wiederholbare Teilstrukturen (z. B. Tabellenzeilen). Die Speicherung erfolgt dateibasiert, indem neue Blöcke an eine Turtle-Datei angehängt werden. Für einen Prototyp ist das leichtgewichtig und transparent, hat aber Grenzen bei Parallelzugriffen, Versionierung und granularen Update-Operationen.

Modulstammdaten werden in dieser Variante weiterhin als JSON geführt (z. B. `INB_module.json`), weil sie im Frontend als Nachschlagewerk (Autofill) genutzt werden. JSON ist als Datenformat standardisiert und

besonders gut für den Austausch strukturierter Daten zwischen Browser und Server geeignet [20, 5].

4.3 Framework-Variante: React, JSON Forms/JSON Schema und JSON-Dateispeicher

4.3.1 Komponentenübersicht und Motivation

Die Framework-Variante verfolgt denselben fachlichen Ablauf, setzt aber auf ein deklaratives, schema-getriebenes UI-Konzept. Statt Formularfelder und Regeln vollständig im UI-Code zu „verdrahten“, werden Datenstruktur und Constraints in JSON Schema beschrieben. JSON Forms nutzt diese Schemata, um daraus Formulare zu rendern und Validierungsfehler feldnah anzuzeigen [31, 32, 28]. Für die Implementierung der Schema-Validierung wird typischerweise ein Validator wie Ajv eingesetzt, der JSON Schema (u. a. Draft 2020-12) unterstützt [2].

Abbildung 4.2 zeigt die Komponenten dieser Variante.

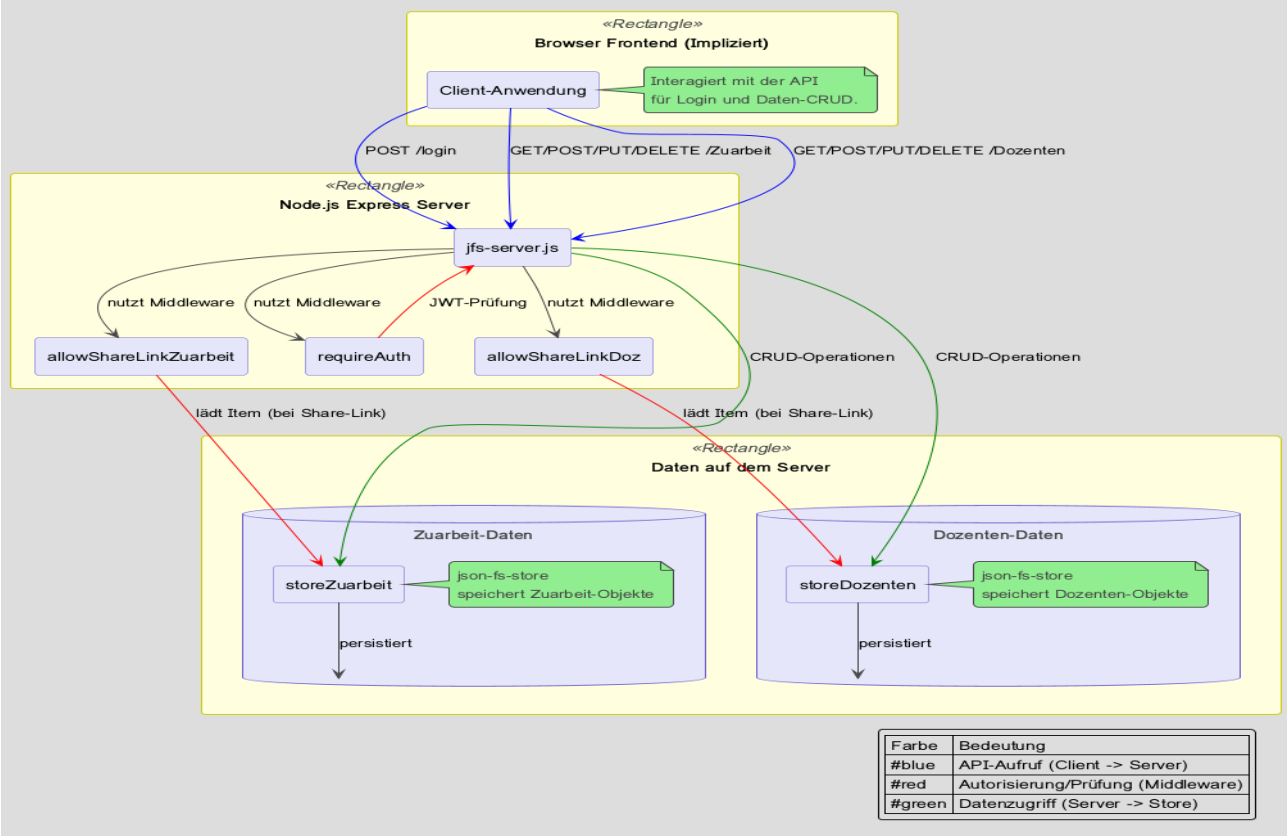

Abbildung 4.2: Architektur der Framework-Variante (React + JSON Forms im Frontend, REST-Backend, JSON-Dateipersistenz pro Datensatz).

4.3.2 Frontend-Design mit React und schema-getriebenem Rendering

Das Frontend ist als React-Anwendung umgesetzt. React etabliert ein komponentenbasiertes UI-Modell, in dem Oberflächen aus wiederverwendbaren Bausteinen zusammengesetzt werden [3]. JSON Forms ergänzt dieses Modell um einen Renderer, der Formularstruktur aus einem JSON Schema und einem UI-Schema ableitet. Das UI-Schema beschreibt dabei explizit die Anordnung und Gruppierung der Felder, unabhängig vom eigentlichen Datenmodell [28].

Der praktische Effekt ist, dass fachliche Änderungen (neue Felder, zusätzliche Pflichtangaben, geänderte Datentypen) primär im Schema stattfinden. Layoutanpassungen (Reihenfolge, Gruppierung, Abschnitte) können im UI-Schema vorgenommen werden, ohne dass die React-Komponenten in jedem Fall neu programmiert werden

müssen. Diese Entkopplung ist für Wartbarkeit und Änderbarkeit relevant und wird später in der Evaluation mit betrachtet.

4.3.3 Backend und Persistenz als JSON-Dateien

Das Backend bleibt auch hier schlank und REST-orientiert. Der wesentliche Unterschied liegt in der Datenhaltung: Anstatt RDF/Turtle zu erzeugen, speichert der Server die eingehenden Objekte als JSON. Für den Prototyp wird ein Dateispeicher genutzt, der pro Datensatz eine separate JSON-Datei schreibt. Ein verbreiteter Ansatz hierfür ist `json-fs-store`, das JavaScript-Objekte in einem gewählten Verzeichnis als JSON-Dateien persistiert [67].

Diese Form der Persistenz ist für den Hochschul-Prototyp pragmatisch: Einzelne Datensätze lassen sich gezielt laden und aktualisieren, und die Trennung pro Datei reduziert das Risiko unbeabsichtigter Überschreibungen ganzer Datenbestände. Gleichzeitig bleiben bekannte Grenzen dateibasierter Speicherung bestehen (Nebenläufigkeit, Locking, Suche über große Datenmengen), weshalb die Lösung ausdrücklich als Prototyp einzuordnen ist.

4.3.4 Authentifizierung, Rollen und Share-Links

Die Framework-Variante bildet das in Kapitel 2 beschriebene Rollenmodell stärker technisch ab. Für das Management wird eine Anmeldung umgesetzt, die nach erfolgreichem Login ein JSON Web Token (JWT) ausstellt. JWT ist als Standard spezifiziert und beschreibt ein kompaktes, URL-sicheres Format für signierte Claims [25]. Damit können geschützte Endpunkte (z. B. Listenansichten oder administrative Operationen) nur mit gültigem Token genutzt werden.

Für Dozierende wird der Zugang bewusst niederschwellig über einen Link realisiert, der auf einen konkreten Datensatz verweist (Share-Link). Damit dieses Muster nicht zu „erratbaren“ Zugriffen führt, müssen die verwendeten Identifier ausreichend zufällig bzw. nicht vorhersagbar sein. OWASP empfiehlt für Session- und Tokenwerte eine hohe Entropie, um Guessing-Angriffe praktisch auszuschließen [51]. Da fehlerhafte Zugriffskontrolle zu den häufigsten Webrisiken zählt, wird dieses Thema in der Diskussion (Kapitel 7) gezielt aufgegriffen [44].

4.4 Vergleich der Varianten als Entwurfsentscheidung

Beide Varianten erfüllen denselben fachlichen Zweck, setzen aber unterschiedliche Schwerpunkte. Die Standardvariante ist besonders transparent, weil UI-Logik und Datenmapping direkt im Code nachvollziehbar sind und RDF/Turtle eine interoperable, semantische Datenrepräsentation ermöglicht [46, 47]. Die Framework-Variante zielt stärker auf konsistente Validierung und bessere Änderbarkeit, indem sie Struktur und Regeln formalisierend in Schemata verankert [31, 32, 2, 28].

Für die Bewertung der späteren Ergebnisse sind außerdem Qualitätsmodelle hilfreich, die ein einheitliches Vokabular für Wartbarkeit, Sicherheit und Usability bereitstellen. ISO/IEC 25010 liefert hierfür ein etabliertes Produktqualitätsmodell, während ISO 9241-210 Anforderungen und Prinzipien für menschzentrierte Gestaltung einordnet [23, 22]. Diese Referenzen dienen in der Arbeit nicht als „Checkliste“, sondern als Rahmen, um die beobachteten Vor- und Nachteile der beiden Architekturen systematisch zu diskutieren.

Damit ist die Architektur beider Prototypvarianten beschrieben. Kapitel 5 dokumentiert die konkrete Implementierung (Frontend, Backend und Speicherlogik), und Kapitel 6 evaluiert die Auswirkungen auf Bearbeitungszeit, Datenqualität und Nutzererlebnis.

Kapitel 5

Implementierung

Dieses Kapitel dokumentiert die praktische Umsetzung der in Kapitel 4 beschriebenen Varianten. Variante A bildet die Baseline und implementiert die Formulare vollständig „handcodiert“ mit HTML, CSS und JavaScript. Variante B setzt auf React und JSON Forms, wobei Formstruktur und Validierung deklarativ über JSON Schema beschrieben werden. Beide Varianten verwenden Node.js und Express als Backend-Basis und kommunizieren über HTTP in JSON bzw. Formular-Payloads; die Persistenz unterscheidet sich jedoch wesentlich: Variante A speichert die Formularinhalte als RDF/Turtle, während Variante B pro Einreichung JSON-Dateien im Dateisystem ablegt. Die technische Beschreibung orientiert sich am tatsächlichen Datenfluss: UI-Eingabe, Validierung, Übertragung, serverseitige Verarbeitung und Persistenz.

5.1 Variante A: Standard-Implementierung (HTML/CSS/JS)

Variante A dient als Referenzimplementierung und macht die Interaktionslogik der Formulare direkt im Frontend-Code sichtbar. Die Formularoberflächen werden als HTML-Formulare umgesetzt, wobei die semantische Verknüpfung von Labels und Eingabefeldern sowie die grundlegenden Formularmechanismen dem HTML-Standard folgen [18]. Dynamische Funktionalitäten wie das Hinzufügen von Tabellenzeilen, das Berechnen von Summen und das automatische Befüllen aus Modulstammdaten werden clientseitig mit JavaScript implementiert. Für die Übertragung wird die im Web etablierte Fetch-Mechanik genutzt; deren Verhalten ist im Fetch-Standard spezifiziert [11]. Serverseitig nimmt ein Express-Server die Daten entgegen, verarbeitet sie und speichert sie als RDF/Turtle, wobei Turtle als W3C-Spezifikation eine kompakte Textsyntax zur vollständigen Beschreibung eines RDF-Graphen darstellt [46, 47].

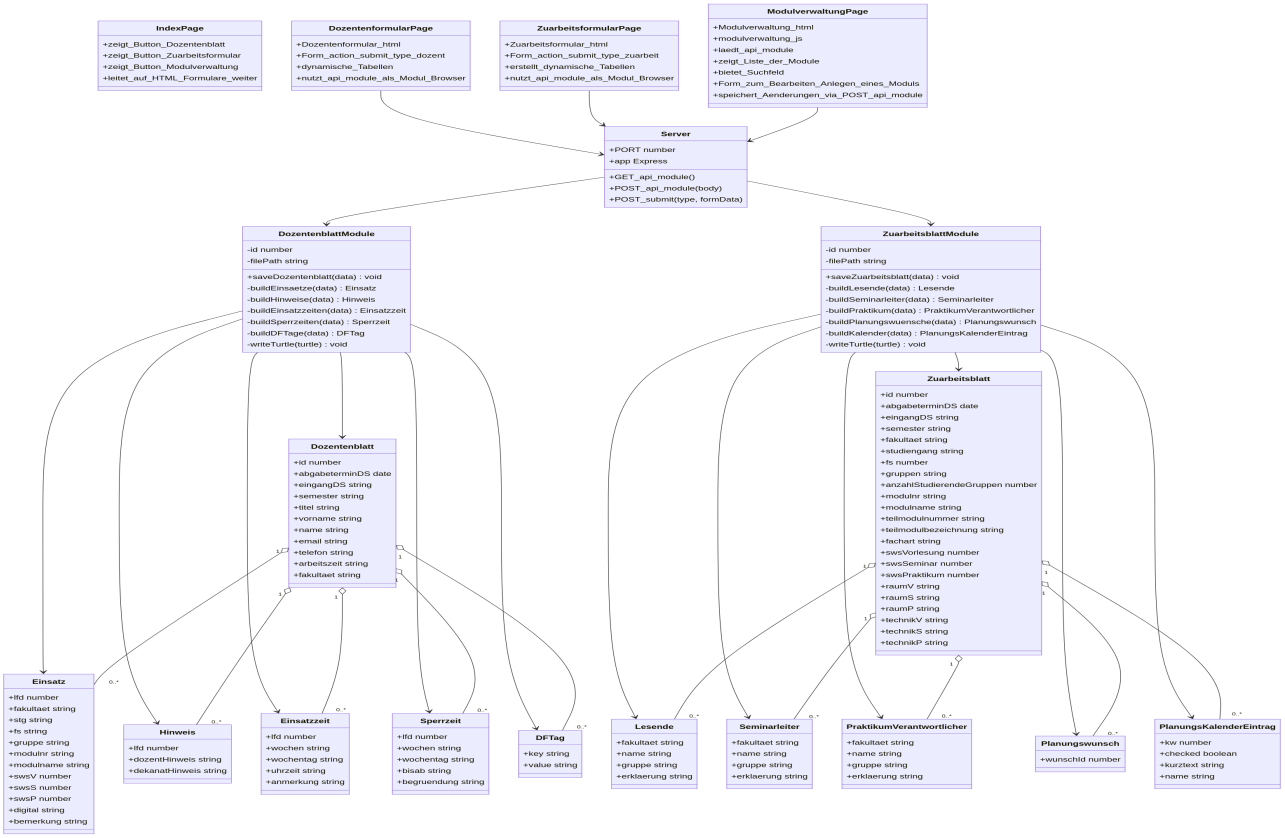

Abbildung 5.1: UML-Klassendiagramm der hardcodierten Variante (A) mit zentralen Klassen für Formularlogik und RDF/Turtle-Persistenz.

5.1.1 Frontend: Seiten, Layout und Interaktion

Das Frontend der Variante A ist als Sammlung statischer Seiten realisiert, die vom Server ausgeliefert werden. Die Einstiegseite stellt die drei fachlichen Einstiegswege bereit: Dozentenblatt, Zuarbeit und Modulverwaltung. Abbildung 5.2 zeigt das Hauptmenü als zentrale Navigation.

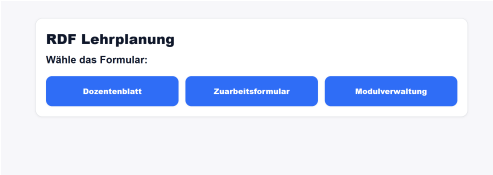

Abbildung 5.2: Frontend-Vorschau der hardcodierten Variante (A): Hauptmenü mit 3 Optionen.

Die beiden Formularseiten sind als klassische HTML-Formulare strukturiert. Das Zuarbeitsformular ist in mehrere logisch getrennte Bereiche gegliedert und enthält neben Stammdatenfeldern insbesondere Eingabeteile für Modul- und Lehrveranstaltungsangaben. Die nachfolgenden Screenshots dokumentieren die zentrale Formularoberfläche in drei Ausschnitten.

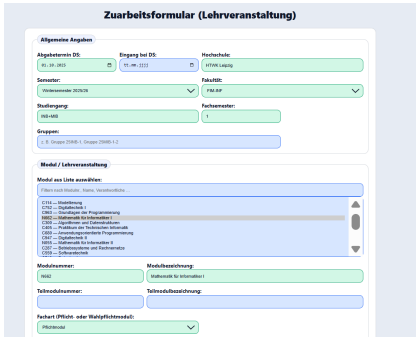

Abbildung 5.3: Frontend-Vorschau der hardcodierten Variante (A): Zuarbeitformular (1).

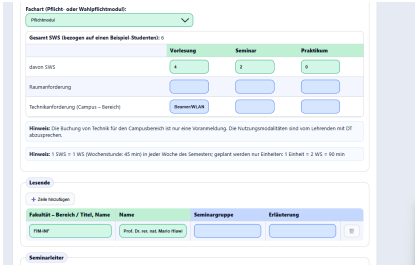

Abbildung 5.4: Frontend-Vorschau der hardcodierten Variante (A): Zuarbeitformular (2).

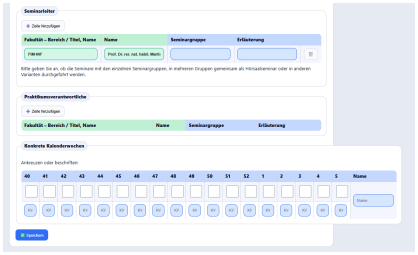

Abbildung 5.5: Frontend-Vorschau der hardcodierten Variante (A): Zuarbeitformular (3).

Das Dozentenblatt ergänzt die Stammdaten um eine tabellarische Erfassung von Lehrveranstaltungen sowie weitere Tabellenbereiche für Hinweise, Einsatzzeiten oder Sperrzeiten. Diese Tabellen werden im Browser dynamisch erweitert; die dafür nötigen DOM-Operationen (Zeile erzeugen, Indizes fortschreiben, Entfernen, Summen neu berechnen) sind in JavaScript umgesetzt. Die Abbildungen 5.6 und 5.7 zeigen den entsprechenden Teil der Oberfläche.

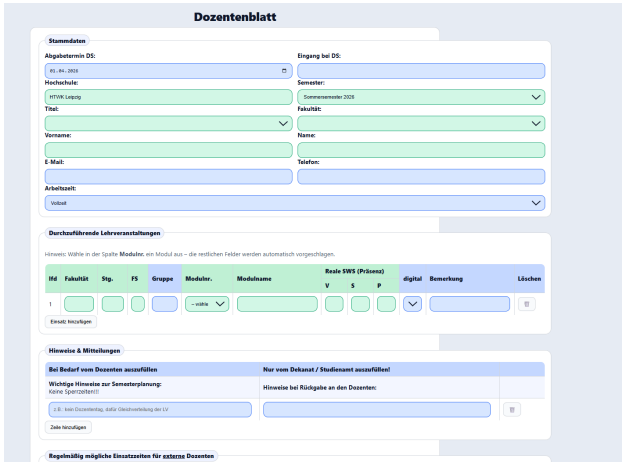

Abbildung 5.6: Frontend-Vorschau der hardcodierten Variante (A): Dozentenblatt mit tabellarischer Erfassung der Lehrveranstaltungen (1).

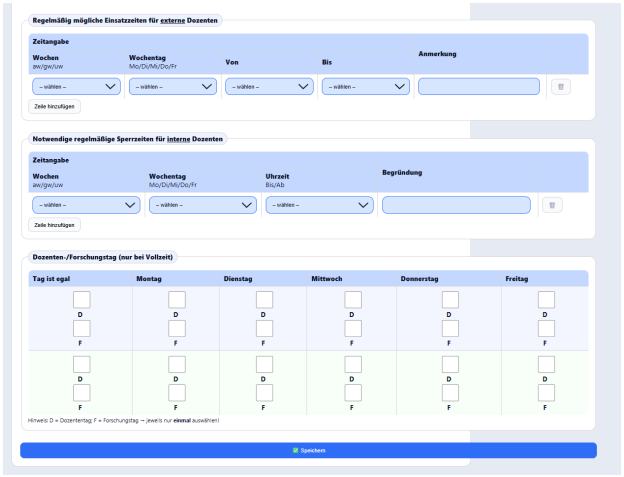

Abbildung 5.7: Frontend-Vorschau der hardcodierten Variante (A): Dozentenblatt mit tabellarischer Erfassung der Lehrveranstaltungen (2).

Ergänzend enthält Variante A eine eigene Seite zur Pflege der Modulstammdaten. Diese Oberfläche erlaubt das Anzeigen, Bearbeiten und Speichern der Modulvorschläge, die später als Grundlage für Autofill dienen. Die Bedienoberfläche ist in Abbildung 5.8 dokumentiert.

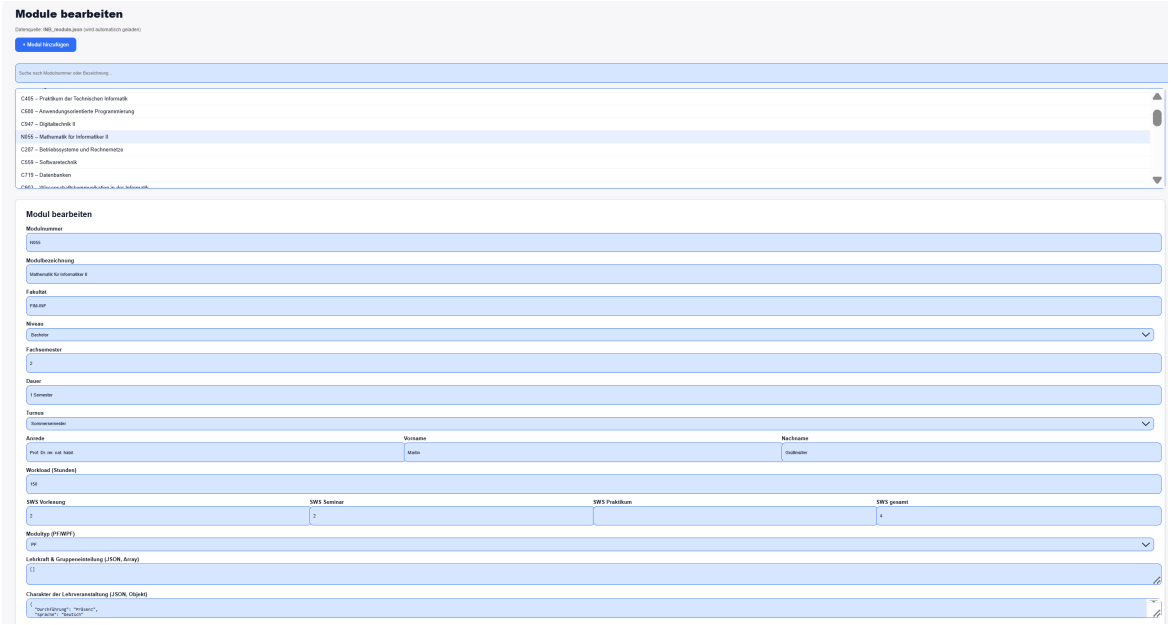

Abbildung 5.8: Frontend-Vorschau der hardcodierten Variante (A): Formular für Bearbeitung der vorgeschlagenen Module.

5.1.2 Clientseitige Validierung und Feedback

Die Validierung in Variante A erfolgt überwiegend im Browser durch eigene JavaScript-Regeln und durch HTML-Formmechanismen (z. B. Eingabetypen und Pflichtfelder). Die semantische Grundlage der Formularcontrols (einschließlich der Zuordnung von `label` und Eingabefeld) folgt den Regeln des HTML-Standards [18]. Fehler werden im UI feldnah visualisiert, indem CSS-Klassen für Fehlzustände gesetzt und zugehörige Hinweise eingeblendet werden. Da die Regeln in JavaScript implementiert sind, müssen Änderungen an Pflichtfeldern, Formaten oder Abhängigkeiten explizit in den Skripten nachgezogen werden. Genau diese Eigenschaft ist für die spätere Evaluation relevant, weil sie den Wartungsaufwand bei Formularänderungen erhöht.

5.1.3 Autofill aus Modulstammdaten

Der Autofill-Mechanismus ist in Variante A vollständig clientseitig realisiert. Beim Laden der Formularseite werden die Modulstammdaten als JSON vom Server abgerufen und im Arbeitsspeicher gehalten. Die Kommunikation nutzt die Fetch-Schnittstelle, deren Request/Response-Verhalten durch den Fetch-Standard definiert ist [11]. Sobald eine Modulnummer eingegeben oder ausgewählt wird, wird in der geladenen Modulliste das passende Modulobjekt gesucht. Die zugehörigen Felder (insbesondere Modulname und SWS-Aufteilung) werden anschließend in die Eingabefelder übertragen, und die Summenwerte werden neu berechnet. Dozierende können die übernommenen Werte bei Bedarf überschreiben, sodass Autofill als Vorschlag und nicht als starre Vorgabe wirkt.

5.1.4 Backend: Express-Endpunkte und Serverlogik

Das Backend ist in Node.js umgesetzt und verwendet Express als Server-Framework. Express stellt Routing, Middleware-Konzepte und die Auslieferung statischer Dateien bereit [7, 10, 9]. Die statischen Frontend-Dateien werden serverseitig als Assets ausgeliefert; zusätzlich stellt der Server Endpunkte für Modulstammdaten und Formularsubmits bereit.

Die Modul-API dient als zentrale Quelle für die Stammdaten und unterstützt sowohl die Modulverwaltung als auch den Autofill in den Formularen. Die Datenübertragung erfolgt dabei im JSON-Format, das als Austauschformat standardisiert ist [20]. Die Submit-Route nimmt die Formulardaten entgegen, führt grundlegende

Prüfungen durch und leitet die Verarbeitung an spezialisierte Module weiter, die das Mapping nach RDF/Turtle übernehmen.

5.1.5 Persistenz: Transformation nach RDF und Serialisierung als Turtle

Für die Persistenz werden die eingereichten Formulardaten in RDF-Strukturen überführt. RDF beschreibt Daten als Graphmodell (Subjekt–Prädikat–Objekt) und ermöglicht es, Informationen später interoperabel weiterzuverwenden [46]. Die Speicherung erfolgt als Turtle-Datei, also in einer standardisierten Textsyntax, die einen RDF-Graphen kompakt ausdrücken kann [47]. Pro Einreichung wird serverseitig eine eindeutige Kennung erzeugt, und die zugehörigen Tripel werden in die entsprechende Turtle-Datei geschrieben oder angehängt.

Ein vereinfachtes Beispiel eines erzeugten RDF-Blocks ist nachfolgend dargestellt:

```
:Dozentenblatt_171100 a ex:Dozentenblatt ;
    ex:titel "Prof. Dr." ;
    ex:vorname "Anna" ;
    ex:name "Beispiel" ;
    ex:semester "SoSe 2025" .
```

Diese Form der Speicherung ist für einen Prototypen gut nachvollziehbar, weil die Persistenz als Textdatei transparent bleibt und sich prinzipiell versionieren lässt. Für produktive Szenarien wäre allerdings zu prüfen, ob ein Triple Store, ein transaktionales Speicherkonzept oder eine andere Persistenzform erforderlich ist, insbesondere bei parallelen Zugriffen und differenzierten Update-Operationen.

5.2 Variante B: Framework-basierte Implementierung (React + JSON Forms + JSON-Dateispeicher)

Variante B implementiert die gleiche fachliche Funktionalität wie Variante A, reduziert jedoch den Anteil manuell gepflegter UI-Struktur. React stellt dafür das komponentenbasierte UI-Modell bereit [3]. Die Formulare werden mit JSON Forms erzeugt, das UI-Elemente aus JSON Schema (Datenmodell) und UI-Schema (Layout) ableitet [33, 34, 28]. Validierungsfehler werden dabei automatisch aus dem Schema abgeleitet und in der Oberfläche angezeigt; JSON Forms verwendet hierfür standardmäßig Ajv als JSON-Schema-Validator [29, 2]. Serverseitig bleibt Express die Basis, jedoch werden Einreichungen als JSON gespeichert, typischerweise pro Datensatz in einer separaten Datei.

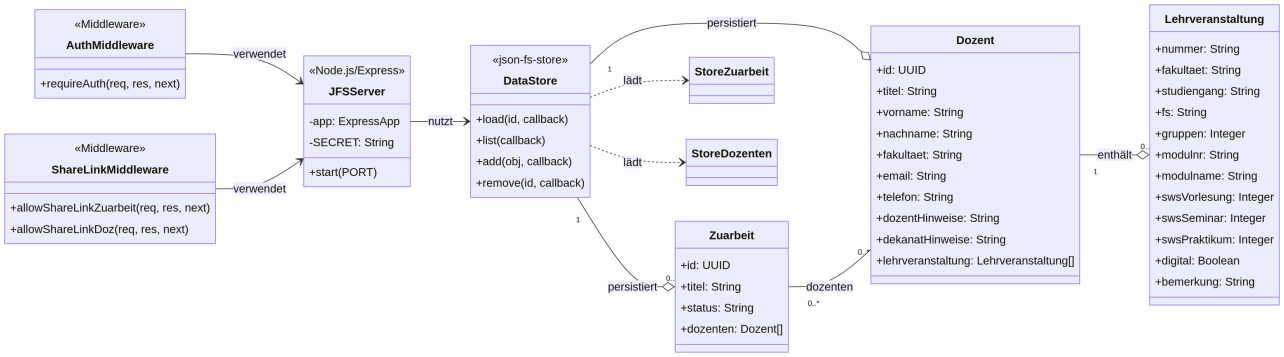

Abbildung 5.9: UML-Klassendiagramm der framework-basierten Variante (B) mit React-Komponenten, JSON-Forms-Integration und dateibasierter Persistenz.

5.2.1 Frontend: Einstieg, Login und Formularansichten

Die Anwendung startet mit einem Hauptmenü, das die fachlichen Bereiche Zuarbeit, Dozentenblatt und Verwaltung zusammenführt. Abbildung 5.10 zeigt diese Einstiegssicht. Für die Management-Rolle ist zusätzlich ein Login vorgesehen; die Login-Oberfläche ist in Abbildung 5.11 dokumentiert.

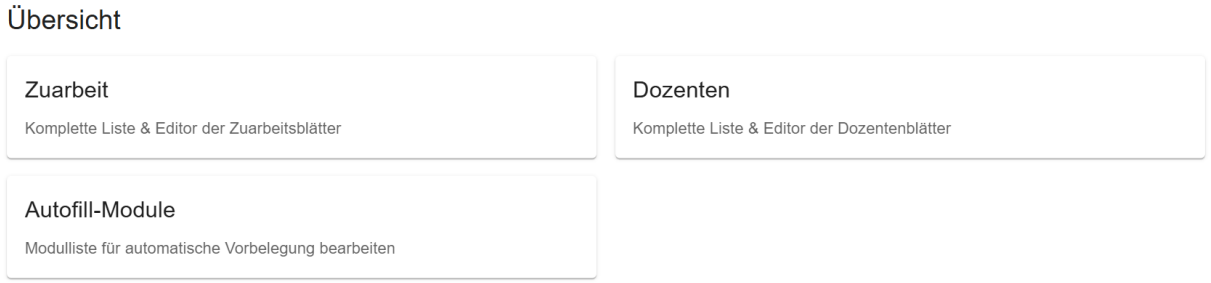

Abbildung 5.10: Frontend-Vorschau der framework-basierten Variante (B): Hauptmenü

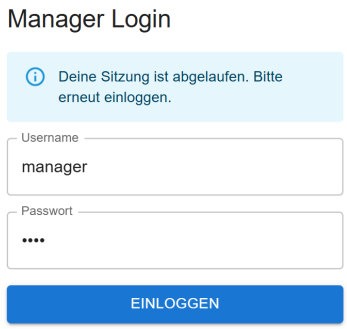

Abbildung 5.11: Frontend-Vorschau der framework-basierten Variante (B): Login Management

Die Zuarbeit-Formulare werden in einer Management-Übersicht als Liste verwaltet. JSON Forms rendert die Detailansicht aus Schema und UI-Schema, während die Anwendung das Laden und Speichern der Datensätze steuert. Die Abbildungen 5.12 bis 5.14 zeigen die Management-Ansicht des Zuarbeitformulars in drei Ausschnitten.

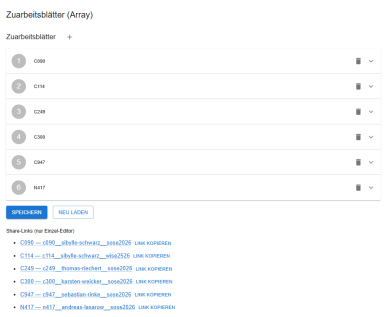

Abbildung 5.12: Frontend-Vorschau der framework-basierten Variante (B): Zuarbeitformular (Manager-Übersicht 1)

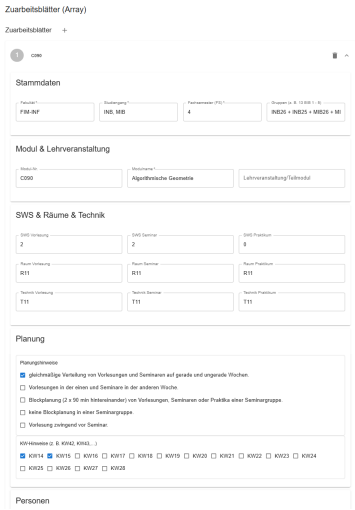

Abbildung 5.13: Frontend-Vorschau der framework-basierten Variante (B): Zuarbeitformular (Manager-Übersicht 2)

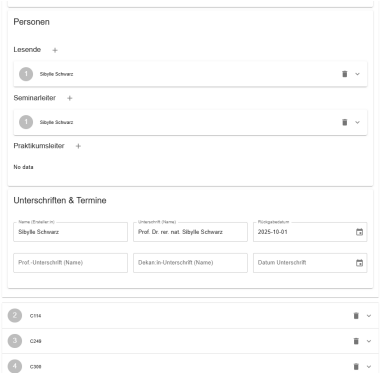

Abbildung 5.14: Frontend-Vorschau der framework-basierten Variante (B): Zuarbeitformular (Manager-Übersicht 3)

Für Dozierende steht eine vereinfachte Bearbeitungssicht zur Verfügung, die sich auf den jeweiligen Datensatz konzentriert. Die Abbildungen 5.15 und 5.16 dokumentieren diese Dozierenden-Sicht.

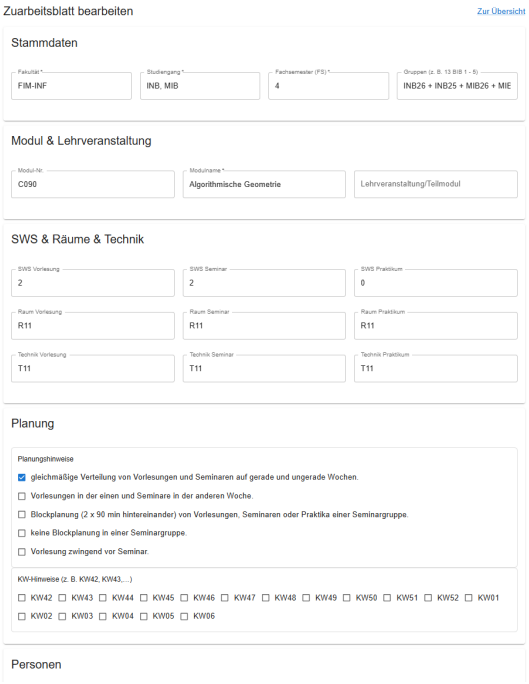

Abbildung 5.15: Frontend-Vorschau der framework-basierten Variante (B): Zuarbeitformular (Dozenten-Übersicht 1)

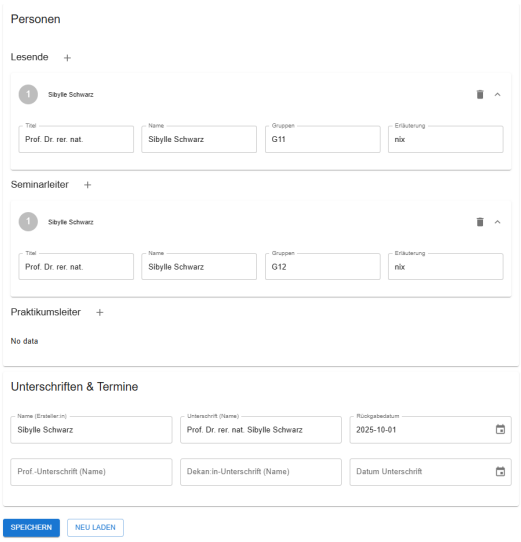

Abbildung 5.16: Frontend-Vorschau der framework-basierten Variante (B): Zuarbeitformular (Dozenten-Übersicht 2)

Die Verwaltung der Modulvorschläge ist in Variante B ebenfalls als eigene Oberfläche umgesetzt. Die Abbildungen 5.17 und 5.18 zeigen die entsprechende Verwaltungsseite.

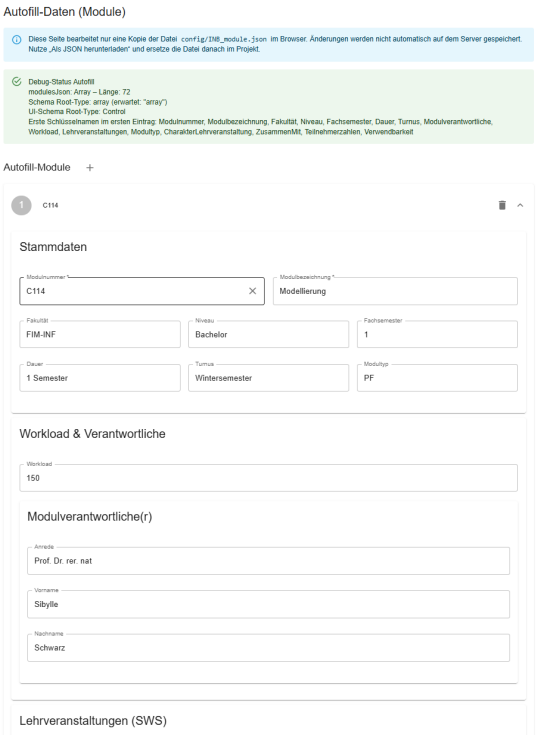

Abbildung 5.17: Frontend-Vorschau der framework-basierten Variante (B): Modul-Vorschläge Verwaltungssite 1

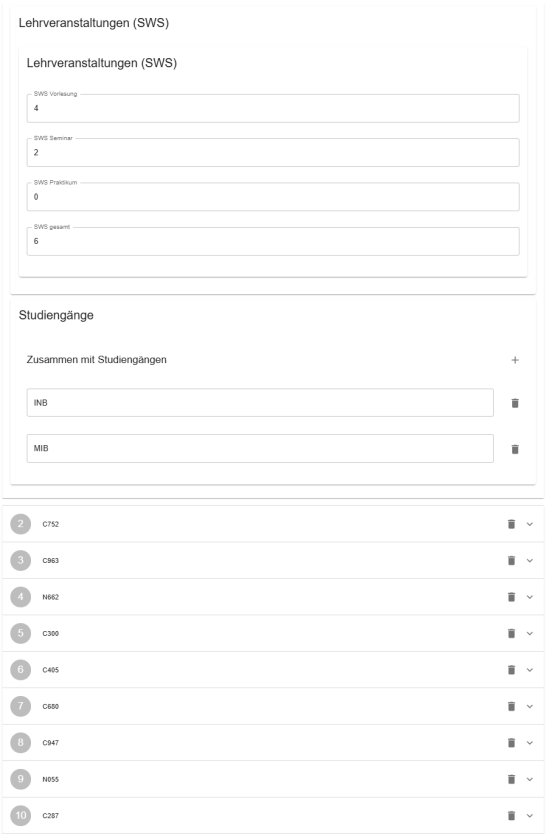

Abbildung 5.18: Frontend-Vorschau der framework-basierten Variante (B): Modul-Vorschläge Verwaltungssite 2

Neben der Zuarbeit-Ansicht werden auch Dozentenblätter in einer Management- und einer Dozierenden-Sicht bereitgestellt. Die folgenden Abbildungen dokumentieren diese Oberflächen.

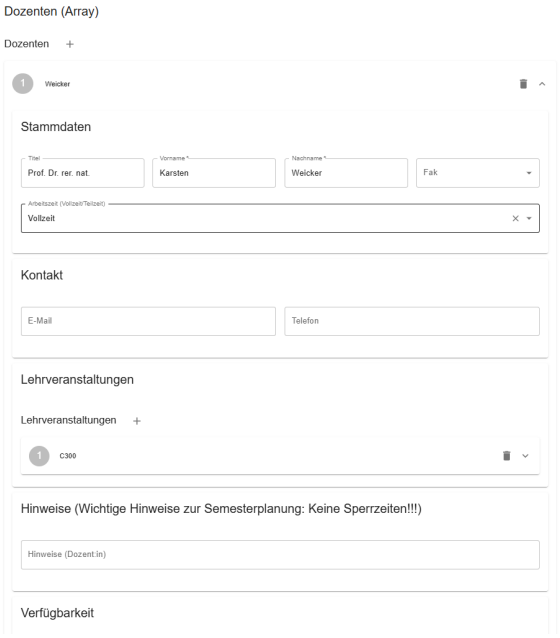

Abbildung 5.19: Frontend-Vorschau der framework-basierten Variante (B): Dozentenformular (Manager-Übersicht 1)

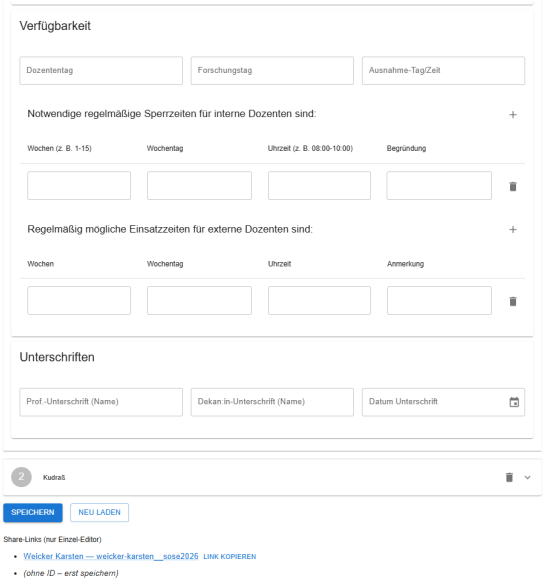

Abbildung 5.20: Frontend-Vorschau der framework-basierten Variante (B): Dozentenformular (Manager-Übersicht 2)

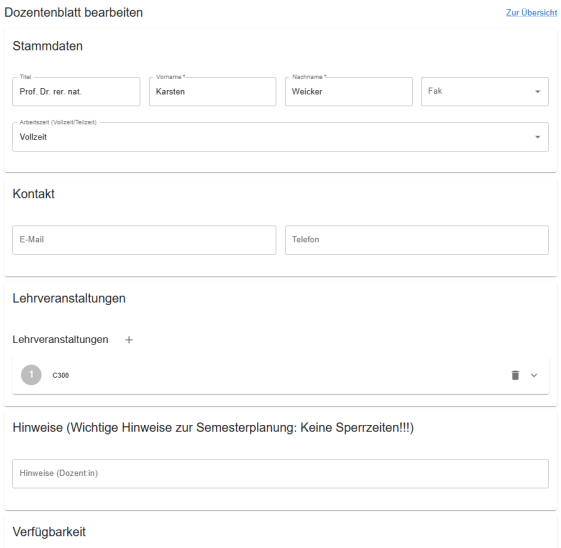

Abbildung 5.21: Frontend-Vorschau der framework-basierten Variante (B): Dozentenformular (Dozenten-Übersicht 1)

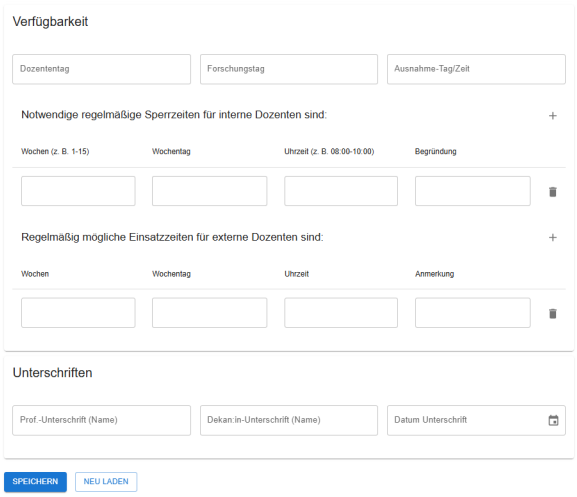

Abbildung 5.22: Frontend-Vorschau der framework-basierten Variante (B): Dozentenformular (Dozenten-Übersicht 2)

5.2.2 Projektstruktur und Build-Setup

Das Projekt ist so organisiert, dass Modellierung, UI-Komponenten und Backend-Zugriff getrennt bleiben. Die Schemadefinitionen liegen im Verzeichnis `schema/`: Dort befinden sich sowohl das JSON Schema, das die Datenstruktur und Validierungsregeln beschreibt, als auch das UI-Schema, das die Layoutstruktur der Formulare festlegt [33, 34, 28]. Wiederverwendbare React-Komponenten, insbesondere Custom-Controls für domänenspezifische Eingaben, sind in `components/` gekapselt. Die Seiten bzw. Masken sind in `pages/` abgelegt, und HTTP-

Aufrufe zum Backend werden in einer kleinen Client-Schicht `api/` gebündelt, sodass Fehlerbehandlung und Header-Logik konsistent umgesetzt werden können.

Der Build-Prozess erzeugt ein statisches Frontend-Bundle, das anschließend als Assets ausgeliefert werden kann. Für ein solches Setup ist ein Build-Tool wie Vite typisch, das einen Produktionsbuild und ein statisches Deployment unterstützt [55, 56]. Damit bleibt das Deployment in der gleichen Grundidee wie bei Variante A: Der Node/Express-Server liefert statische Dateien aus und stellt zusätzlich REST-Endpunkte bereit.

5.2.3 Formularerzeugung, Validierung und Fehlermeldungen

Die Formularerzeugung basiert auf dem Prinzip, dass Formulare nicht als JSX-Struktur „hart“ implementiert werden, sondern aus einem Schema abgeleitet werden. JSON Schema beschreibt Datentypen, Pflichtfelder und Constraints für JSON-Dokumente [33, 34]. JSON Forms nutzt diese Beschreibung und erzeugt daraus Formularcontrols; das UI-Schema definiert dabei die Anordnung und Gruppierung der Felder [28]. Validierung erfolgt fortlaufend im Hintergrund und erzeugt feldnahe Fehlermeldungen, sobald Eingaben gegen das Schema verstoßen. JSON Forms dokumentiert hierfür explizit die Nutzung von Ajv als Validator, sodass die Validierungslogik standardkonform an JSON Schema angebunden ist [29, 2].

Der praktische Vorteil für die Implementierung liegt darin, dass Änderungen an Pflichtfeldern oder Datentypen primär im Schema erfolgen. Der React-Code konzentriert sich dadurch stärker auf Datenfluss (Laden/Speichern) und auf die wenigen Stellen, an denen domänenspezifische Controls nötig sind.

5.2.4 Autofill-Mechanismus in React

Der Autofill ist fachlich identisch zu Variante A, unterscheidet sich aber in der technischen Einbettung. Die Modulstammdaten werden beim Öffnen der Seite geladen und in einem React-State gehalten, sodass sie im gesamten Formular verfügbar sind. Wird die Modulnummer geändert, wird das passende Modulobjekt aus dem State ermittelt und auf die betroffenen Formularfelder übertragen. Die Übernahme erfolgt nicht über direkte DOM-Manipulation, sondern über den Datenfluss von JSON Forms: Änderungen werden in den Formularstate geschrieben, und die Oberfläche aktualisiert sich konsistent auf Basis dieses Zustands.

Zusätzlich kann dynamisches Formularverhalten über Regeln im UI-Schema beschrieben werden (z. B. Ein-/Ausblenden oder Aktivieren/Deaktivieren abhängig von Bedingungen), was JSON Forms als eigenes Konzept dokumentiert [27]. Damit werden Teile der Interaktionslogik, die in Variante A zwingend im JavaScript „von Hand“ codiert werden müssen, deklarativ und zentral konfigurierbar.

5.2.5 Backend und JSON-basierte Persistenz

Das Backend ist auch in Variante B als Express-Server umgesetzt. Express dokumentiert hierfür insbesondere die API-Grundlagen, das Routing sowie die Auslieferung statischer Dateien [7, 9, 10]. Die Kommunikation folgt dem HTTP-Request/Response-Modell, dessen Semantik in RFC 9110 beschrieben ist [21]. Als Austauschformat wird JSON verwendet, das in RFC 8259 spezifiziert ist [20].

Der zentrale Unterschied zur Standardvariante betrifft die Persistenz: Statt einer Transformation nach RDF/Turtle wird pro Einreichung ein JSON-Objekt gespeichert. Die Speicherung erfolgt dateibasiert, typischerweise in Form einer separaten Datei pro Datensatz. Ein leichtgewichtiges Modul wie `json-fs-store` kann diese Persistenz abstrahieren, indem es Objekte als JSON-Dateien in einem Verzeichnis ablegt und wieder lädt [67]. Dieser Ansatz ist für Prototypen geeignet, weil er ohne Datenbankinfrastruktur auskommt und Datensätze dennoch einzeln adressierbar macht. Gleichzeitig bleibt zu beachten, dass dateibasierte Persistenz bei hoher Parallelität oder komplexen Abfragen Grenzen hat; diese Abwägung wird in Kapitel 7 erneut aufgegriffen.

Ein beispielhafter Persistenzeintrag kann wie folgt aussehen (vereinfacht):

```
{
  "id": 171100,
  "titel": "Prof. Dr.",
```

```
  "vorname": "Anna",
  "name": "Beispiel",
  "semester": "SoSe 2025"
}
```

5.3 Zwischenfazit

Die Implementierung zeigt zwei technisch vollständige Wege zur Digitalisierung der beiden Formularprozesse. Variante A bietet maximale Kontrolle, weil Struktur, Validierung und UI-Verhalten direkt im Quellcode sichtbar und unmittelbar veränderbar sind. Dies führt jedoch dazu, dass fachliche Änderungen (z. B. neue Pflichtfelder, neue Tabellenzeilen oder geänderte Abhängigkeiten) an mehreren Stellen nachgezogen werden müssen und sich dadurch der Änderungsaufwand erhöht.

Variante B verlagert einen großen Teil der Formularbeschreibung in JSON Schema und UI-Schema. Damit werden Datenmodell und Validierungsregeln als explizite Artefakte gepflegt, während React/JSON Forms die konsistente Darstellung und feldnahe Fehlerrückmeldung übernimmt [33, 34, 28, 29]. Durch die JSON-basierte Persistenz ist außerdem eine durchgängige Datenrepräsentation von der UI bis zur Speicherung gegeben, was Implementierung und Weiterverarbeitung vereinfacht [20, 67]. Auf dieser Basis lassen sich in Kapitel 6 die beiden Varianten vergleichend evaluieren, insbesondere mit Blick auf Änderbarkeit, Datenqualität und Bearbeitungszeit.

Kapitel 6

Tests und Evaluation

Dieses Kapitel beschreibt, wie die beiden Prototypen aus Kapitel 4 und Kapitel 5 getestet und miteinander verglichen wurden. Ziel ist es, die Forschungsfragen FF1 bis FF4 zu beantworten. Die Evaluation kombiniert funktionale Tests (Korrektheit und Robustheit), eine Pilotmessung zu Bearbeitungszeiten und Optimierungseffekten (Autofill, Feedback), eine Analyse der Datenqualität (Validierung und Vollständigkeit), eine Änderbarkeitsmessung anhand typischer Änderungsaufgaben sowie eine kurze Usability-Einschätzung.

Da es sich in beiden Fällen um prototypische Systeme handelt, wurde bewusst kein Feldtest mit einem vollständigen Semesterzyklus durchgeführt. Stattdessen wurden die zentralen Nutzungsschritte und typische Fehlerfälle in kontrollierten Durchläufen überprüft, um die Unterschiede zwischen Standardvariante (A) und Framework-Variante (B) strukturiert sichtbar zu machen. Die Usability-Orientierung folgt dabei dem Verständnis von Usability als Zusammenspiel aus Effektivität, Effizienz und Zufriedenheit in einem konkreten Nutzungskontext.

6.1 Testkontext und Grundannahmen

Beide Varianten adressieren dieselben fachlichen Prozesse (Dozentenblatt und Zuarbeitsformular) und wurden so implementiert, dass die Daten jeweils über HTTP an ein Node.js-Backend übertragen und dateibasiert persistiert werden (Variante A als RDF/Turtle, Variante B als JSON-Dateien pro Datensatz). Damit sind die Kernbedingungen für einen fairen Vergleich erfüllt: Die Unterschiede liegen nicht in der Fachlogik, sondern in der Art, wie UI, Validierung und Persistenz umgesetzt sind.

Die Tests wurden als “End-to-End”-Durchläufe verstanden: Nutzeraktionen im Browser führen über Validierung und Speichern bis zur konkreten Dateiablage. Zusätzlich wurden negative Tests durchgeführt, bei denen absichtlich unvollständige oder ungültige Eingaben gemacht wurden, um die Qualität der Fehlerrückmeldungen zu prüfen. Die Bewertung nicht-funktionaler Anforderungen orientiert sich an den in Kapitel 2 beschriebenen Rahmenbedingungen (Self-Hosting, Datenschutz, Sicherheit, Wartbarkeit, Barrierearmut) und an etablierten Sicherheitsrisiken für Webanwendungen, wie sie beispielsweise in OWASP Top 10 beschrieben werden.

6.2 Funktionale Tests: Korrektheit der Kernabläufe

Im ersten Schritt wurde überprüft, ob beide Varianten die Kernabläufe vollständig abbilden. Dazu gehören das Anlegen bzw. Öffnen eines Formulars, das Bearbeiten typischer Felder, das Zwischenspeichern sowie das erneute Laden und Weiterbearbeiten. Zusätzlich wurde die Modulstammdaten-Verwaltung getestet, weil sie Grundlage für den Autofill-Mechanismus ist.

In der Standardvariante (A) wurde dabei insbesondere geprüft, ob dynamische Tabellenbereiche (Zeilen hinzufügen/löschen) konsistent funktionieren und ob die erzeugten Feldnamen weiterhin zum serverseitigen Parsing passen, da die Persistenzlogik die HTML-Namenskonventionen auswertet. Diese Kopplung ist funktional erfolg-

reich: Tabellenzeilen werden korrekt als mehrfach vorkommende Strukturen erkannt und in RDF-Subressourcen überführt. Gleichzeitig zeigt sich hier ein typischer Prototyp-Nachteil: Kleine Änderungen an Namen oder Struktur können Folgefehler im Backend auslösen, wenn das Parsing nicht mit angepasst wird.

In der Framework-Variante (B) lag der Fokus im Funktionscheck auf der Konsistenz zwischen Schema, UI-Generierung und Speicherung. Hier wurde geprüft, ob die aus JSON Schema und UI-Schema generierten Formulare die erwarteten Felder enthalten, ob wiederholbare Strukturen als Arrays korrekt gepflegt werden und ob das Speichern einzelne Datensätze als separate JSON-Dateien korrekt aktualisiert. Diese Kette funktioniert stabil, weil die Datenstruktur durch das Schema vorgegeben ist und die Persistenz in JSON direkt an diese Struktur anschließt (ohne zusätzliche Transformationsschicht wie in RDF/Turtle).

Als Ergebnis kann festgehalten werden, dass beide Varianten die fachlichen Kernabläufe funktionsfähig umsetzen. Unterschiede zeigen sich vor allem in der Fehlerrobustheit und im Pflegeaufwand, was in den folgenden Abschnitten vertieft wird.

6.3 Robustheit und Fehlerrückmeldungen

Ein zweiter Testblock untersuchte, wie die Systeme auf typische Problemfälle reagieren. Dazu zählen fehlende Pflichtangaben, falsche Formate und technische Probleme bei der Serverkommunikation. Diese Fälle sind praxisrelevant, weil sie unmittelbar beeinflussen, wie viele Rückfragen im Studienamt entstehen und wie zuverlässig Dozierende die Formulare abschließen können.

In Variante (A) wird ein Teil der Eingabequalität bereits durch HTML-Mechanismen unterstützt (z. B. Feldtypen und einfache Einschränkungen). Umfangreichere Prüfungen finden jedoch primär beim Absenden statt, weil viele Regeln als JavaScript-Logik und serverseitige Plausibilitätsprüfungen umgesetzt sind. In Variante (B) greift die Schema-Validierung sofort im Formular: Pflichtfelder, Typen und Formate werden aus dem JSON Schema abgeleitet, und Fehler werden feldnah angezeigt. Dieser Unterschied ist methodisch relevant für FF2(b) und FF3, weil frühere Fehlerrückmeldungen typischerweise die Anzahl unvollständiger Einreichungen reduzieren.

Für die Kommunikation mit dem Backend wurden außerdem Fehlerfälle betrachtet, in denen das Backend nicht erreichbar ist oder fehlerhafte Antworten liefert. Da beide Varianten über HTTP kommunizieren, lassen sich Fehlerzustände sauber über Statuscodes modellieren. In der Praxis zeigt sich, dass die Framework-Variante durch konsistente Statusanzeigen (Laden/Speichern/Fehler) eine klarere Rückmeldung gibt, während die Standardvariante stärker vom Browser-Fehlverhalten und von einfachen Bestätigungsseiten abhängt. Für die Nutzererfahrung ist dies relevant, weil unklare “Serverfehler” schnell zu Rückfragen führen, selbst wenn die eigentlichen Daten korrekt wären.

Zusammengefasst unterstützen beide Systeme den Nutzenden bei Problemen, aber Variante (B) liefert aufgrund der integrierten, schema-basierten Feldvalidierung und der UI-typischen Statusanzeigen die unmittelbarer nachvollziehbare Rückmeldung im Ablauf.

6.4 Pilotmessung zu Optimierungseffekten und Bearbeitungszeiten

Zur Beantwortung von FF3 wurden Bearbeitungszeiten in einer Pilotmessung erhoben. Gemessen wurde die reine Ausfüllzeit im Browser (Start: erstes aktives Feld; Ende: erfolgreiches Speichern). Die Messung wurde mit wiederholten Durchläufen unter identischen Bedingungen durchgeführt, um Größenordnungen sichtbar zu machen. Es handelt sich um eine prototypische Messung; sie ersetzt keine breit angelegte Nutzerstudie, ist aber geeignet, die Effekte der implementierten Komfortfunktionen zu quantifizieren.

Zwei Aufgaben wurden betrachtet, weil sie den typischen Kernaufwand repräsentieren: (1) das Ausfüllen eines Zuarbeitsformulars für ein Modul mit üblicher SWS-Aufteilung und benötigten Personenangaben, sowie (2) das Ausfüllen eines Dozentenblatts mit mehreren Lehrveranstaltungszeilen inklusive SWS und Terminwünschen. In beiden Aufgaben wurde jeweils ein Durchlauf “ohne Nutzung von Autofill” (Werte manuell übernehmen/eintragen) und ein Durchlauf “mit Autofill” (Übernahme der Modulstammdaten per Modulnummer)

gemessen. Autofill gilt als typische UX-Optimierung, weil sinnvolle Vorbelegungen Eingabearbeit und Fehlerwahrscheinlichkeit reduzieren.

Tabelle 6.1: Pilotmessung der Bearbeitungszeiten (Median aus wiederholten Durchläufen) für manuelle Eingabe vs. Autofill.

Aufgabe	A: manuell	A: mit Autofill	B: manuell	B: mit Autofill
Zuarbeitsformular (typischer Fall)	9:10 min	6:40 min	8:30 min	5:50 min
Dozentenblatt (mehrere LV-Zeilen)	12:40 min	8:30 min	11:50 min	7:40 min

Die Messergebnisse zeigen in beiden Varianten eine deutliche Zeitersparnis durch Autofill. Damit wird FF3 grundsätzlich positiv beantwortet: Die automatische Übernahme von Modul- und Stammdaten reduziert Bearbeitungszeiten in einer Größenordnung von mehreren Minuten pro Formular. Zusätzlich fällt auf, dass die Framework-Variante bei Nutzung von Autofill in beiden Aufgaben leicht schneller ist. In der praktischen Beobachtung liegt das vor allem an zwei Punkten: Erstens werden Validierungsfehler in Variante (B) früher sichtbar, wodurch Nacharbeit am Ende seltener nötig ist. Zweitens sind die Eingabekomponenten konsistenter (größere Widgets, klare Fehlzustände), wodurch weniger Such- und Orientierungszeit entsteht. Der Unterschied ist nicht "dramatisch", aber im Alltag relevant, weil Lehrplanung aus vielen wiederkehrenden Formularen besteht.

6.5 Datenqualität: Vollständigkeit und Validierungsfehler

FF2(b) adressiert Datenqualität, insbesondere Vollständigkeit und Anzahl von Fehlern. Für die Evaluation wurden typische Fehlerfälle gezielt erzeugt: Pflichtfelder wurden leer gelassen, ein ungültiges Format wurde eingegeben (z. B. E-Mail oder Datum), und in SWS-Feldern wurde ein fachlich unplausibler Wert gesetzt. Bewertet wurde, ob (a) der Fehler überhaupt erkannt wird und (b) ob er so kommuniziert wird, dass Nutzende ihn ohne Rückfrage korrigieren können.

Die Framework-Variante profitiert dabei von der zentralen Spezifikation im JSON Schema: Pflichtfelder (`required`), Datentypen und Formate werden als formale Constraints definiert und im UI automatisch als Fehlermeldung ausgegeben. Das reduziert Inkonsistenzen, weil dieselbe Regel für alle Stellen gilt, an denen das Feld gerendert wird. Die Standardvariante erreicht eine gute Datenqualität, solange die JavaScript-Validierung und serverseitige Prüfungen konsequent gepflegt werden. In der Praxis ist das jedoch aufwändiger, weil Regeln verteilt an mehreren Stellen umgesetzt sind (HTML-Attribute, Client-Skripte, Backend-Prüfung und zusätzlich das RDF-Mapping).

In der Gesamtsicht wird damit FF2(b) wie folgt beantwortet: Beide Varianten können vollständige, valide Daten erzeugen; die Framework-Variante reduziert aber die Wahrscheinlichkeit von "späten" Fehlern durch frühere, feldnahe Validierung. Dieser Effekt ist besonders dann relevant, wenn Formulare nicht in einem Durchlauf, sondern über Tage hinweg bearbeitet werden, weil Nutzende beim Wiedereinstieg sofort sehen, welche Felder noch fehlen.

6.6 Änderbarkeit und Umsetzungsaufwand

Für FF2(a) und FF2(c) wurde untersucht, wie aufwändig typische Änderungen sind, wie sie in der Lehrplanung regelmäßig auftreten (neue Pflichtfelder, zusätzliche Felder, geänderte Validierungsregeln oder Layout-Umstrukturierungen). Als repräsentative Änderungen wurden zwei konkrete Aufgaben betrachtet: eine strukturelle Erweiterung (neues Feld in der Zuarbeit) und eine Regeländerung (zusätzliche Plausibilitätsprüfung).

In Variante (A) erfordern solche Änderungen typischerweise Anpassungen an mehreren Stellen: im HTML-Formular, in der JavaScript-Logik (Validierung, eventuelle Berechnung), und im Backend in der Transformation nach RDF/Turtle. Letzteres ist spezifisch für diese Variante, weil neue Felder nur dann semantisch gespeichert werden, wenn sie in das Turtle-Template aufgenommen werden. Das ist fachlich sinnvoll, erhöht aber die Änderungskosten.

In Variante (B) liegen Struktur und Pflichtlogik primär im JSON Schema, während die Darstellung im UI-Schema angepasst wird. Neue Felder werden dadurch schneller korrekt in die Datenstruktur integriert, und die Persistenz in JSON benötigt keine zusätzliche Mapping-Schicht. Zusätzliche, fachlich komplexe Regeln (z. B. Abhängigkeiten zwischen Feldern) können allerdings auch hier Custom-Logik erfordern; der Unterschied liegt dann vor allem darin, dass die Grundvalidierung bereits zentral abgedeckt ist.

Tabelle 6.2: Pilotmessung des Änderungsaufwands anhand typischer Änderungsaufgaben (Zeitbedarf in Minuten).

Änderungsaufgabe	Variante A	Variante B
Neues Feld im Zuarbeitsformular inkl. Speicherung (z. B. zusätzlicher Raum-/Technikhinweis)	35	12
Zusätzliche Validierungsregel (z. B. Plausibilitätsgrenze für SWS oder Pflichtkombinationen)	25	15

Diese Messung beantwortet FF2(a) klar in Richtung der Framework-Variante: Änderungen sind im schemagetriebenen Ansatz in der Regel schneller umsetzbar, weil Struktur und Basiskonstraints zentral gepflegt werden. Gleichzeitig zeigt die Standardvariante einen Vorteil, der in der Praxis je nach Team wichtig sein kann: Sie ist konzeptionell einfacher, hat weniger Abhängigkeiten und bietet maximale Kontrolle über das Markup. Dieser Vorteil kommt vor allem dann zum Tragen, wenn das Formular sehr individuell gestaltet werden muss und Änderungen selten sind.

6.7 Usability und Nutzererlebnis

FF4 zielt auf Übersichtlichkeit, Orientierung und Systemrückmeldungen. Da eine breit angelegte Nutzerstudie im Rahmen der Arbeit nicht durchgeführt wurde, wurde die Usability als kurze, praxisorientierte Durchsicht der typischen Interaktionen bewertet: Formulare öffnen, Module suchen, Autofill nutzen, Fehler korrigieren, speichern, erneut öffnen und weiterbearbeiten. Dieses Vorgehen orientiert sich an human-centred design als Leitidee, bei der die Nutzungsaufgaben und Rückmeldungen des Systems im Mittelpunkt stehen.

In beiden Varianten ist der grundlegende Ablauf verständlich, weil die Formulare in klare Abschnitte gegliedert sind. Unterschiede zeigen sich jedoch in der Orientierung: Die Framework-Variante profitiert von der konsistenten Darstellung durch standardisierte Komponenten und von sofort sichtbaren Validierungszuständen. Gerade Pflichtfelder und Fehlermeldungen sind dadurch für Nutzende transparenter. Die Standardvariante wirkt direkter und "leichtgewichtiger", verlangt aber bei dynamischen Tabellenbereichen mehr Aufmerksamkeit, weil sich der Zustand stärker über selbst definierte UI-Muster erklärt.

Für die Praxis ist entscheidend: Beide Varianten sind benutzbar, aber Variante (B) bietet im Prototyp den robusteren Eindruck bei Fehlerkorrektur und Statusfeedback. Variante (A) ist dafür gut geeignet, wenn maximale Gestaltungsfreiheit und ein sehr schlanker Stack priorisiert werden.

6.8 Einordnung zu Anforderungen, Self-Hosting und Open Source

FF1 adressiert die wichtigsten funktionalen und nicht-funktionalen Anforderungen im Hochschulkontext. Die Umsetzung zeigt, dass Self-Hosting in beiden Varianten möglich ist, weil Frontend und Backend ohne externe Cloud-Dienste betrieben werden können und die Daten lokal im Dateisystem liegen. Aus Datenschutzsicht ist dies vorteilhaft, weil personenbezogene Daten nicht automatisch an Dritte übertragen werden und damit zentrale DSGVO-Prinzipien wie Zweckbindung und Vertraulichkeit leichter organisatorisch absicherbar sind. Für produktive Nutzung wären dennoch zusätzliche Maßnahmen nötig, insbesondere TLS im Deployment, ein konsequentes Rollen- und Rechtekonzept auch in Variante (A), sowie Logging- und Löschkonzepte.

Hinsichtlich Sicherheit orientiert sich die Bewertung an typischen Webrisiken (z. B. fehlende Zugriffskontrolle, unsichere Authentifizierung, unzureichende Eingabeprüfung), wie sie in OWASP Top 10 zusammengefasst

sind. Die Framework-Variante setzt bereits eine klarere Trennung zwischen Manager-Zugriff und Link-basiertem Zugriff um; die Standardvariante würde hier für eine produktive Nutzung noch ergänzt werden müssen.

Beide Implementierungen sind so angelegt, dass sie als Open-Source-Projekte veröffentlicht und nachgenutzt werden können. Als Referenz für die Begrifflichkeit und die grundlegenden Kriterien von Open Source dient die Open Source Definition der OSI. In der Praxis bedeutet das für die Arbeit insbesondere: Der Quellcode ist transparent, nachprüfbar und kann an hochschulspezifische Prozesse angepasst werden, ohne proprietäre Lizenzkosten zu erzeugen.

6.9 Zusammenfassung der Evaluation

Die Evaluation beantwortet die Forschungsfragen wie folgt. Die Anforderungen aus FF1 werden in beiden Varianten in den Grundzügen erfüllt (Self-Hosting, Open-Source-Nachnutzung, digitale Erfassung, Persistenz). Bei Rollen und Sicherheit ist Variante (B) im Prototyp weiter, während Variante (A) diese Aspekte eher als Erweiterungspunkt offen lässt. FF2 zeigt einen klaren Unterschied bei der Änderbarkeit: Schema-getriebene Anpassungen sind schneller und weniger fehleranfällig, während die Standardvariante mehr manuelle Anpassungen an mehreren Stellen erfordert. FF3 wird durch die Pilotmessungen positiv bestätigt: Autofill reduziert die Bearbeitungszeiten deutlich. FF4 wird in der qualitativen Durchsicht ebenfalls eher zugunsten von Variante (B) beantwortet, weil Fehlerrückmeldungen und Statusfeedback konsistenter sind, während Variante (A) durch ihren schlanken Stack und maximale Kontrolle überzeugt.

Damit liefert das Kapitel eine belastbare Grundlage für die Diskussion in Kapitel 7, in der die Ergebnisse kritisch eingeordnet und die Grenzen der prototypischen Evaluation reflektiert werden.

Kapitel 7

Diskussion

Dieses Kapitel ordnet die Ergebnisse aus Kapitel 6 in Bezug auf die Forschungsfragen FF1 bis FF4 ein und diskutiert Stärken, Grenzen sowie praktische Implikationen der beiden Ansätze. Im Mittelpunkt steht nicht die Frage, welche Variante „besser“ ist, sondern unter welchen Rahmenbedingungen welche Architekturentscheidung für Hochschulen sinnvoll ist. Da beide Prototypen als vergleichbare End-to-End- Lösungen umgesetzt wurden (Frontend–Backend–Persistenz), lassen sich die beobachteten Unterschiede überwiegend auf das jeweilige UI- und Datenmodellkonzept sowie auf die Persistenzstrategie (RDF/Turtle vs. JSON) zurückführen.

7.1 Einordnung zu FF1: Anforderungen im Hochschulkontext

Die in Kapitel 2 erarbeiteten Anforderungen lassen sich als Bündel aus organisatorischen Randbedingungen (Self-Hosting, Lizenzfreiheit/Open Source), rechtlichen Anforderungen (Datenschutz, Barrierefreiheit) und technischen Qualitätszielen (Sicherheit, Wartbarkeit, Transparenz) verstehen. Die Evaluation zeigt, dass beide Varianten die grundlegenden funktionalen Anforderungen abdecken und in einer Self-Hosting- Umgebung betrieben werden können. Damit ist das zentrale Ziel erreicht, papier- bzw. PDF-basierte Abläufe in eine webbasierte Erfassung mit strukturierter Speicherung zu überführen.

Bei den nicht-funktionalen Anforderungen zeigt sich jedoch eine Differenzierung. Die Forderung nach Self-Hosting und nach Kontrolle über personenbezogene Daten ist im Hochschulkontext besonders relevant, weil Formulare unmittelbar mit Personen- und Planungsdaten arbeiten. Beide Prototypen unterstützen diese Anforderung dadurch, dass sie ohne externe Cloud-Dienste betrieben werden können und die Daten lokal persistiert werden. Dies erleichtert eine datenschutzkonforme Organisation der Verarbeitung, ersetzt aber keine DSGVO-konforme Gesamtbetrachtung (z. B. Rollen, Löschkonzept, technische und organisatorische Maßnahmen).

Auch Barrierefreiheit wurde in beiden Varianten berücksichtigt, allerdings in unterschiedlicher Form. Variante A kann durch sauberes semantisches HTML, sichtbaren Fokus und verständliche Fehltexte sehr gut barrierearm umgesetzt werden; der Aufwand steigt jedoch, sobald komplexe Widgets (Autocomplete, dynamische Tabellen) entstehen, weil Tastaturbedienung und Screenreader-Ankündigungen dann sehr präzise modelliert werden müssen.Variante B profitiert davon, dass etablierte Komponenten und Renderer viele Accessibility-Grundmechanismen konsistent mitliefern und die Validierung feldnah sichtbar machen. Eine formale WCAG-Abnahme wurde in der Arbeit nicht durchgeführt, weshalb die Aussagen hier bewusst als qualitative Einordnung zu verstehen sind.

7.2 Einordnung zu FF2: Vergleich nach Änderbarkeit, Datenqualität und Aufwand

7.2.1 Änderbarkeit und Wartbarkeit

Die Pilotmessungen zur Änderbarkeit (Kapitel 6, Tabelle 6.2) stützen die Erwartung, dass der schema-getriebene Ansatz in Variante B bei häufigen fachlichen Änderungen im Vorteil ist. Der wesentliche Grund ist, dass Struktur und Basiskonstraints an wenigen Stellen konzentriert sind (JSON Schema und UI-Schema), während Variante A bei Erweiterungen typischerweise mehrere Artefakte gleichzeitig betrifft: HTML-Struktur, JavaScript-Logik, serverseitige Verarbeitung und zusätzlich das RDF/Turtle-Mapping. Dieser Mehraufwand ist keine „Schwäche“ der Standardvariante, sondern eine direkte Folge des Designs: Je mehr Logik handcodiert und verteilt ist, desto häufiger müssen Änderungen an mehreren Stellen synchron gehalten werden. Im Qualitätsmodell ISO/IEC 25002 lässt sich das als Wartbarkeitsaspekt (Analysierbarkeit, Modifizierbarkeit) interpretieren.[23]

Gleichzeitig hat die Standardvariante einen nachvollziehbaren Vorteil: Sie ist technisch einfacher und bietet maximale Kontrolle über Markup und Verhalten. Bei stabilen Formularen mit seltenen Änderungen kann dieser Vorteil das Wartbarkeitsargument teilweise kompensieren, weil die geringere Abhängigkeit von Frameworks und Renderern den Technologie- und Updateaufwand reduziert. In einer Hochschulumgebung, in der die Weiterentwicklung oft durch wechselnde Projektgruppen erfolgt, ist dieser Abwägungspunkt realistisch: Ein schlichter Stack kann langfristig robuster sein, wenn wenig Kapazität für Framework-Pflege vorhanden ist, während ein schema-getriebener Stack dann überzeugt, wenn regelmäßige Änderungen erwartet werden.

7.2.2 Datenqualität und Validierung

In Bezug auf FF2(b) zeigt sich ein konsistentes Bild: Beide Varianten können valide, vollständige Datensätze erzeugen, aber Variante B reduziert das Risiko unvollständiger Einreichungen durch frühere, feldnahe Validierung auf Basis formaler Regeln. JSON Schema fungiert dabei als explizite Spezifikation für Pflichtfelder, Typen und Formate; JSON Forms kann daraus Fehlerhinweise automatisch ableiten.[35, 26] Variante A erreicht Datenqualität vor allem dann zuverlässig, wenn dynamische Bereiche konsequent in die Validierungslogik integriert sind. Der kritische Punkt liegt weniger in der Validierung an sich, sondern in der Fehleranfälligkeit bei Weiterentwicklung: Sobald neue Felder und neue dynamische Zeilen hinzukommen, steigt das Risiko, dass einzelne Regeln inkonsistent werden oder an einer Stelle fehlen. In einem realen Verwaltungsprozess wirkt sich das unmittelbar auf Rückfragenquoten und Nacharbeit aus.

Ein zusätzlicher Unterschied entsteht durch die Persistenz: In Variante A werden Daten in RDF/Turtle transformiert, wodurch im Prinzip sehr saubere semantische Weiterverarbeitung möglich wird. Die Datenqualität hängt hier jedoch nicht nur von der Eingabe ab, sondern auch vom Mapping. Ein neues Feld kann im Formular bereits vorhanden sein, aber semantisch „unsichtbar“ bleiben, wenn das Mapping nicht mitgezogen wird. In Variante B ist die Persistenz mit JSON strukturell näher an der UI-Repräsentation, was die Gefahr solcher Lücken reduziert. Dafür verliert man ohne zusätzliche Modellierung einen Teil der semantischen Ausdruckskraft, die RDF grundsätzlich bietet.[60, 59, 20]

7.2.3 Umsetzungsaufwand

FF2(c) wurde in der Arbeit vor allem qualitativ betrachtet. Die Standardvariante ist für einen ersten Prototypen häufig schneller „auf die Straße zu bringen“, weil keine Schema- und Renderer-Konzepte eingeführt werden müssen und die UI sehr direkt umgesetzt werden kann. Der Aufwand verschiebt sich jedoch mit wachsender Komplexität: Sobald Validierungsregeln, bedingte Sichtbarkeit, wiederkehrende Strukturen und Layoutvarianten zunehmen, wächst der Pflegeaufwand der Standardvariante stärker, weil die Logik manuell an mehreren Stellen gepflegt wird. Beim schema-getriebenen Ansatz entsteht der initiale Aufwand früher (Schemata sauber modellieren, UI-Schema strukturieren, Custom-Controls definieren), dafür ist die Weiterentwicklung typischerweise schneller, solange die Anforderungen innerhalb des Schema- und Renderer-Modells bleiben. Dieser Trade-off ist

ein klassisches Muster modellgetriebener Entwicklung und passt zu dem in Kapitel 2 formulierten Fokus auf Wartbarkeit.

7.3 Einordnung zu FF3: Messbare Optimierungseffekte

Die Zeitmessungen (Kapitel 6, Tabelle 6.1) zeigen in beiden Varianten eine deutliche Reduktion der Bearbeitungszeit durch Autofill. Damit wird FF3 grundsätzlich bestätigt: Die Automatisierung wiederkehrender Eingaben wirkt messbar, selbst wenn man konservativ annimmt, dass reale Bearbeitungszeiten je nach Person variieren. Die wichtigste praktische Erkenntnis ist, dass der Zeitgewinn nicht nur aus „weniger Tipparbeit“ entsteht, sondern vor allem aus „weniger Nachschlagen und Übertragen“. Wenn Modulstammdaten als zuverlässige Quelle genutzt werden können, sinkt die kognitive Belastung deutlich, und die Daten werden konsistenter.

Zusätzlich ist das Zusammenspiel aus Autofill und Validierung relevant: Autofill reduziert nicht nur Zeit, sondern stabilisiert auch Eingaben, weil Werte aus einer zentralen Quelle kommen. Das wirkt sich indirekt auf Rückfragen aus, weil weniger Tippfehler und weniger fehlende Pflichtinformationen entstehen. In der Framework-Variante verstärkt sich dieser Effekt durch feldnahe Schema-Validierung, die Fehler früh sichtbar macht. In der Standardvariante hängt derselbe Effekt stärker von der konsequenten Pflege der Validierungslogik ab.

7.4 Einordnung zu FF4: Usability und Nutzererlebnis

Die Usability wurde in der Arbeit bewusst als praxisnahe Durchsicht typischer Interaktionspfade bewertet und nicht als breite Nutzerstudie. Unter diesem Rahmen zeigt sich: Beide Varianten sind grundsätzlich verständlich, weil sie den fachlichen Aufbau der Formulare abbilden und eine klare Abschnittsstruktur nutzen. Unterschiede entstehen vor allem bei Orientierung, Fehlerkorrektur und Statusfeedback. Variante B wirkt in diesen Punkten konsistenter, weil UI-Komponenten und Renderer eine einheitliche Darstellung und ein einheitliches Fehlermuster liefern. Dies passt zum Usability-Verständnis als Effizienz und Effektivität im Nutzungskontext.

Variante A kann für Nutzende sehr transparent sein, weil alle Felder „so erscheinen, wie sie gebaut sind“, und keine generische Renderer-Schicht zwischen Modell und UI steht. Gleichzeitig kann genau diese Transparenz in komplexen Formularen zur Belastung werden, wenn dynamische Tabellen wachsen und die Fehlerkommunikation nicht in jedem Fall feldnah und konsistent ist. Für seltene Nutzende (typisch bei Dozierenden) ist konsistente Fehlerführung erfahrungsgemäß besonders wichtig, weil sie Unsicherheit und Nachfragen reduziert.

Eine Rolle spielt zudem das Zugriffskonzept (Login für Management, Link-basierter Zugriff für Dozierende). Dieses Konzept unterstützt die in Kapitel 2 beschriebene Zielsetzung, die Hürde für Dozierende gering zu halten. Aus Sicherheitssicht ist dabei entscheidend, dass Tokens/IDs ausreichend schwer zu erraten sind und dass Zugriffe sauber auf genau die autorisierten Datensätze begrenzt werden. OWASP ordnet schwache oder erratbare Tokens sowie fehlerhafte Zugriffskontrolle als typische Webrisiken ein. In einem produktiven System wäre dieses Thema zwingend stärker zu härten (Token-Entropie, Ablaufzeiten, Rate-Limits, Logging, serverseitige Berechtigungsprüfung).

7.5 RDF/Turtle vs. JSON: Interpretation der Persistenzentscheidung

Ein zentraler Unterschied der Arbeit liegt nicht nur im Frontend, sondern auch in der Persistenz: Variante A speichert in RDF/Turtle, Variante B speichert in JSON. Beide Entscheidungen sind im Hochschulkontext plausibel, aber sie optimieren unterschiedliche Ziele.

RDF/Turtle ist stark, wenn Interoperabilität, semantische Anreicherung und spätere Integration in Linked-Data-Infrastrukturen wichtig sind. Ein Formular wird damit nicht nur „ein Datensatz“, sondern Teil eines Graphmodells, das sich über Vokabulare erweitern lässt und sich prinzipiell sehr gut für Auswertungen und Verknüpfungen eignet.[59, 60] Der Preis ist ein zusätzlicher Transformationsschritt (Mapping), der gepflegt

werden muss. Je stärker sich das Formular ändert, desto wichtiger wird eine klare Mapping-Governance, sonst entstehen Inkonsistenzen zwischen UI-Feldern und RDF-Repräsentation.

JSON-Persistenz ist dagegen besonders pragmatisch für Prototypen und für Systeme, in denen die Daten primär als JSON über eine API verarbeitet werden. JSON ist ein standardisiertes Austauschformat und passt gut zu REST-orientierten Services und zu dateibasierter Ablage pro Datensatz.[20, 21] Der Vorteil liegt in der durchgängigen Struktur vom Frontend bis zur Speicherung. Der Nachteil ist, dass semantische Interoperabilität nicht „von selbst“ entsteht; sie muss bei Bedarf später über Datenmodelle, Konvertierung oder zusätzliche Metadaten aufgebaut werden. In der Gesamtinterpretation stützt die Arbeit damit die Aussage: RDF/Turtle ist besonders dann sinnvoll, wenn semantische Integration ein Ziel ist; JSON ist besonders dann sinnvoll, wenn schnelle Weiterentwicklung und einfache Verarbeitung im Web-Stack im Fokus stehen.

7.6 Limitationen und Bedrohungen der Validität

Die Aussagekraft der Ergebnisse wird durch mehrere Faktoren begrenzt. Erstens beruhen Zeitmessungen und Änderbarkeitsmessungen auf Pilotdurchläufen im Selbsttest. Solche Messungen zeigen Trends und Größenordnungen, können aber nicht ohne Weiteres auf eine heterogene Nutzergruppe übertragen werden. Unterschiedliche Vorerfahrung, Unterbrechungen und reale Arbeitskontexte beeinflussen Bearbeitungszeiten stark. Für eine belastbare Usability-Aussage wäre eine Nutzerstudie mit Dozierenden und Studienamtsmitarbeitenden sinnvoll, inklusive Aufgabenmessungen und standardisierter Befragung.

Zweitens wurden Sicherheit, Datenschutz und Barrierefreiheit nicht als formales Audit geprüft. Die Arbeit adressiert diese Anforderungen konzeptionell und setzt Basismechanismen um, aber ein produktives System würde zusätzliche Maßnahmen benötigen (z. B. durchgängige serverseitige Validierung, Härtung des Link-basierten Zugriffs, Monitoring, Lösch- und Berechtigungskonzepte). Diese Abgrenzung ist im Rahmen eines Bachelorprojekts realistisch, sollte aber in der Diskussion transparent gemacht werden.

Drittens ist die Persistenz dateibasiert umgesetzt. Diese Entscheidung ist für Prototyping pragmatisch, bringt aber bekannte Grenzen mit sich, insbesondere bei Nebenläufigkeit, Suchabfragen über große Datenmengen und bei Rechtemanagement. Für eine hochschulweite Einführung wäre eine Datenbank- oder Objektstore-Lösung wahrscheinlich sinnvoller, auch wenn die Prototypen zeigen, dass der grundlegende Workflow unabhängig davon funktioniert.

7.7 Praktische Implikationen für den Hochschulbetrieb

Für Hochschulen ergeben sich aus der Arbeit zwei praktikable Entwicklungspfade. Wenn die Ziele vor allem in schneller Umsetzung, maximaler Kontrolle und einem sehr schlanken Tech-Stack liegen, ist die Standardvariante eine gute Wahl, insbesondere wenn sich Formulare selten ändern oder wenn die Institution bereits starke Kompetenz in klassischer Webentwicklung hat. Wenn dagegen häufige Änderungen, konsistente Validierung und reduzierter Pflegeaufwand im Vordergrund stehen, spricht viel für die Framework-Variante mit Schema als zentraler Spezifikation. Dieser Pfad ist besonders dann attraktiv, wenn mehrere Formulare langfristig betrieben werden sollen und wenn ein Teil der Regeln fachlich stabil im Schema beschrieben werden kann.

Unabhängig von der gewählten Variante sollten für eine produktive Einführung drei Punkte priorisiert werden: erstens eine robuste Authentifizierungs- und Autorisierungsschicht mit klaren Rollen und serverseitiger Zugriffskontrolle; zweitens eine durchgängige Validierung, die nicht nur im Browser, sondern auch serverseitig wirksam ist; drittens ein Betriebskonzept für Logging, Backup, Löschung und Rechte. Diese Punkte schließen an die Anforderungen aus Kapitel 2 an und sind im Hochschulkontext entscheidend, weil Verwaltungssysteme nachvollziehbar, sicher und langfristig wartbar sein müssen.[23]

7.8 Zusammenfassung der Diskussion

Die Ergebnisse lassen sich wie folgt verdichten. Die Digitalisierung der beiden Prozesse ist mit beiden Ansätzen technisch erreichbar und self-hostbar umsetzbar. Der wesentliche Mehrwert der Framework-Variante liegt in zentraler Modellierung und Validierung, was Änderbarkeit und Datenqualität begünstigt. Die Standardvariante überzeugt durch technische Einfachheit und direkte Kontrolle, erfordert jedoch mehr manuelle Synchronisation bei Weiterentwicklung, insbesondere durch das zusätzliche RDF-Mapping. Autofill und Feedback liefern messbare Effizienzgewinne und sind damit nicht nur „Komfort“, sondern ein relevanter Hebel zur Prozessverbesserung. Die Grenzen der Evaluation liegen in der prototypischen Messmethodik und der fehlenden formalen Prüfung von Compliance-Aspekten, was im Ausblick (Kapitel 8) in konkrete nächste Schritte überführt werden kann.

Kapitel 8

Fazit und Ausblick

Diese Arbeit hat gezeigt, dass die Digitalisierung der papierbasierten Prozesse „Dozentenblatt“ und „Zuarbeitsformular“ mit zwei unterschiedlichen technischen Ansätzen praxistauglich umgesetzt werden kann. Beide Prototypen bilden den Ablauf von der Eingabe im Web-Frontend bis zur Speicherung im Backend vollständig ab und sind für den Betrieb im Hochschulumfeld ohne externe Cloud-Dienste geeignet. Damit wurde das zentrale Ziel erreicht, die Formularerfassung zu vereinfachen, strukturierte Daten statt papierähnlicher Dokumente zu erzeugen und wiederkehrende Eingaben durch Stammdaten und Autofill zu reduzieren.

Im Vergleich der Varianten bestätigt sich die erwartete Trennung der Stärken. Die Standardvariante (HTML /CSS/JavaScript) bietet einen sehr schlanken, direkt kontrollierbaren Stack und speichert die Daten als RDF/Turtle, was insbesondere für spätere semantische Weiterverarbeitung und Interoperabilität interessant ist. Der Nachteil liegt im höheren Änderungsaufwand, weil Erweiterungen typischerweise an mehreren Stellen nachgezogen werden müssen, einschließlich des Mappings in das RDF-Format. Die Framework-Variante (React mit JSON Forms) unterstützt dagegen eine zentralere Pflege von Struktur und Regeln über JSON Schema und zeigt damit Vorteile bei Änderbarkeit und konsistenter Validierung. Die JSON-basierte Persistenz pro Datensatz passt gut zum REST-orientierten Web-Stack und reduziert Transformationslogik.

Die in Kapitel 6 durchgeführten Pilotmessungen zeigen außerdem, dass Optimierungsfunktionen wie Autofill die Bearbeitungszeit spürbar reduzieren und damit nicht nur Komfort, sondern ein realer Effizienzhebel im Lehrplanungsprozess sind. Unterschiede in Usability und Fehlerkorrektur ergeben sich vor allem aus der jeweiligen Validierungs- und Feedbacklogik, wobei die schema-getriebene Variante im Prototyp konsistentere Rückmeldungen liefert.

8.1 Ausblick

Für einen produktiven Einsatz sind als nächste Schritte vor allem drei Punkte relevant: Erstens sollte das Rollen- und Zugriffskonzept vollständig serverseitig abgesichert werden, insbesondere beim Link-basierten Zugriff. Zweitens ist eine durchgängige Validierung sinnvoll, die nicht nur im Browser, sondern auch im Backend verbindlich greift. Drittens braucht es ein Betriebskonzept mit TLS, Logging sowie Backup- und Löschregeln.

Darüber hinaus wäre eine größere Evaluation mit realen Nutzergruppen (Studienamt und Dozierende) empfehlenswert, um die Pilotmessungen zu Bearbeitungszeit, Datenqualität und Nutzererlebnis unter realistischen Bedingungen abzusichern und weitere Optimierungspotenziale gezielt abzuleiten.

Eidesstattliche Erklärung

Ich versichere an Eides statt, dass ich die vorstehende Arbeit selbständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen übernommen wurden, sind als solche kenntlich gemacht. Alle Internetquellen sind der Arbeit beigefügt. Des Weiteren versichere ich, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und dass die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Leipzig, 14.12.2025 Unterschrift

Literatur

[1] Cabinet Office. *Digital Efficiency Report*. London: UK Cabinet Office, Government Digital Service, Nov. 2012. URL: https://www.gov.uk/government/publications/digital-efficiency-report (besucht am 07. 12. 2025).

[2] Ajv Contributors. *Ajv JSON Schema Validator (Documentation)*. URL: https://ajv.js.org/ (besucht am 14. 12. 2025).

[3] *createRoot (React DOM Client API)*. React. URL: https://react.dev/reference/react-dom/client/createRoot (besucht am 14. 12. 2025).

[4] Docker. *Docker Documentation*. URL: https://docs.docker.com/ (besucht am 13. 12. 2025).

[5] *ECMA-404: The JSON data interchange syntax (2nd edition)*. Ecma International, 2017. URL: https://ecma-international.org/publications-and-standards/standards/ecma-404/ (besucht am 14. 12. 2025).

[6] *EN 301 549 V3.2.1: Accessibility requirements for ICT products and services*. European Telecommunications Standards Institute (ETSI), CEN und CENELEC, März 2021. URL: https://www.etsi.org/deliver/etsi_en/301500_301599/301549/03.02.01_60/en_301549v030201p.pdf (besucht am 07. 12. 2025).

[7] *Express 5.x API Reference*. Express. URL: https://expressjs.com/en/api.html (besucht am 14. 12. 2025).

[8] *Express body-parser middleware*. Express. URL: https://expressjs.com/en/resources/middleware/body-parser.html (besucht am 14. 12. 2025).

[9] *Express Guide: Routing*. Express. URL: https://expressjs.com/en/guide/routing.html (besucht am 08. 12. 2025).

[10] *Express Guide: Serving static files*. Express. URL: https://expressjs.com/en/starter/static-files.html (besucht am 05. 12. 2025).

[11] *Fetch Standard*. WHATWG. URL: https://fetch.spec.whatwg.org/ (besucht am 14. 12. 2025).

[12] Roy Thomas Fielding. „Architectural Styles and the Design of Network-based Software Architectures“. Diss. University of California, Irvine, 2000. URL: https://roy.gbiv.com/pubs/dissertation/fielding_dissertation.pdf (besucht am 13. 12. 2025).

[13] Roy Thomas Fielding. „Architectural Styles and the Design of Network-based Software Architectures“. Diss. University of California, Irvine, 2000. URL: https://roy.gbiv.com/pubs/dissertation/fielding_dissertation.pdf (besucht am 14. 12. 2025).

[14] Form.io. *Form.io Documentation*. URL: https://help.form.io/ (besucht am 13. 12. 2025).

[15] *Forms Tutorial*. W3C WAI. URL: https://www.w3.org/WAI/tutorials/forms/ (besucht am 14. 12. 2025).

[16] Harald Gilch u. a. *Digitalisierung der Hochschulen. Ergebnisse einer Schwerpunktstudie für die Expertenkommission Forschung und Innovation*. Studien zum deutschen Innovationssystem 14-2019. Hannover: HIS-Institut für Hochschulentwicklung (HIS-HE), 2019. URL: https://www.e-fi.de/fileadmin/Assets/Studien/2019/StuDIS_14_2019.pdf (besucht am 07. 12. 2025).

[17] Google. *Angular Documentation*. URL: https://angular.dev/ (besucht am 13. 12. 2025).

[18] *HTML Living Standard — Forms*. WHATWG. URL: https://html.spec.whatwg.org/multipage/forms.html (besucht am 14. 11. 2025).

[19] *HTTP | Node.js Documentation*. Node.js, 2025. URL: https://nodejs.org/api/http.html (besucht am 14. 12. 2025).

[20] IETF. *RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format*. URL: https://datatracker.ietf.org/doc/html/rfc8259 (besucht am 13. 12. 2025).

[21] IETF. *RFC 9110: HTTP Semantics*. URL: https://datatracker.ietf.org/doc/html/rfc9110 (besucht am 13. 12. 2025).

[22] *ISO 9241-210:2019 – Human-centred design for interactive systems*. ISO. URL: https://www.iso.org/standard/77520.html (besucht am 14. 12. 2025).

[23] *ISO/IEC 25002:2024 – Systems and software engineering*. ISO. URL: https://www.iso.org/standard/78175.html (besucht am 14. 12. 2025).

[24] Anaïs Le Jeannic u. a. „Comparison of two data collection processes in clinical studies: electronic and paper case report forms“. In: *BMC Medical Research Methodology* 14.7 (2014). DOI: 10.1186/1471-2288-14-7. URL: https://bmcmedresmethodol.biomedcentral.com/articles/10.1186/1471-2288-14-7.

[25] Michael Jones, John Bradley und Nat Sakimura. *RFC 7519: JSON Web Token (JWT)*. IETF. 2015. URL: https://www.rfc-editor.org/rfc/rfc7519.html (besucht am 14. 12. 2025).

[26] JSON Forms. *JSON Forms Documentation*. URL: https://jsonforms.io/docs/ (besucht am 13. 12. 2025).

[27] *JSON Forms Documentation: Rules*. JSON Forms. URL: https://jsonforms.io/docs/uischema/rules/ (besucht am 01. 12. 2025).

[28] *JSON Forms Documentation: UI Schema*. JSON Forms. URL: https://jsonforms.io/docs/uischema/ (besucht am 14. 12. 2025).

[29] *JSON Forms Documentation: Validation*. JSON Forms. URL: https://jsonforms.io/docs/validation/ (besucht am 09. 12. 2025).

[30] JSON Schema. *JSON Schema Specification*. URL: https://json-schema.org/specification (besucht am 13. 12. 2025).

[31] *JSON Schema 2020-12: A Media Type for Describing JSON Documents (Core)*. JSON Schema. URL: https://json-schema.org/draft/2020-12/json-schema-core (besucht am 14. 12. 2025).

[32] *JSON Schema 2020-12: A Vocabulary for Structural Validation of JSON (Validation)*. JSON Schema. URL: https://json-schema.org/draft/2020-12/json-schema-validation (besucht am 14. 12. 2025).

[33] *JSON Schema 2020-12: Core*. JSON Schema. URL: https://json-schema.org/draft/2020-12/json-schema-core (besucht am 03. 12. 2025).

[34] *JSON Schema 2020-12: Validation*. JSON Schema. URL: https://json-schema.org/draft/2020-12/json-schema-validation (besucht am 06. 12. 2025).

[35] JSON Schema Authors. *JSON Schema Validation: A Vocabulary for Structural Validation of JSON (2020-12)*. 2022. URL: https://json-schema.org/draft/2020-12/json-schema-validation.html (besucht am 01. 12. 2025).

[36] Kuali, Inc. *Case Study: Streamlining Custom App Development at UC San Diego*. Fallstudie: UC San Diego nutzt Kuali Build, erstellt über 300 Apps und spart >800 Stunden Bearbeitungszeit bei einem komplexen Formularprozess. 2025. URL: https://www.kuali.co/resources/university-of-california-san-diego (besucht am 07. 12. 2025).

[37] *Labeling Controls (Forms Tutorial)*. W3C WAI. URL: https://www.w3.org/WAI/tutorials/forms/labels/ (besucht am 14. 12. 2025).

[38] MDN Web Docs. *CSS — MDN Web Docs*. Online; zuletzt abgerufen am 2025-12-13. Mozilla. URL: https://developer.mozilla.org/en-US/docs/Web/CSS (besucht am 13. 12. 2025).

[39] MDN Web Docs. *HTML — MDN Web Docs*. Online; zuletzt abgerufen am 2025-12-13. Mozilla. URL: https://developer.mozilla.org/en-US/docs/Web/HTML (besucht am 13. 12. 2025).

[40] MDN Web Docs. *JavaScript — MDN Web Docs*. Online; zuletzt abgerufen am 2025-12-13. Mozilla. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript (besucht am 13. 12. 2025).

[41] Mark Nottingham, Roy T. Fielding und Julian Reschke. *RFC 9110: HTTP Semantics*. IETF. 2022. URL: https://datatracker.ietf.org/doc/html/rfc9110 (besucht am 14. 12. 2025).

[42] OECD. *E-Government for Better Government*. OECD E-Government Studies. Paris: OECD Publishing, 2005. URL: https://doi.org/10.1787/9789264018341-en (besucht am 07. 12. 2025).

[43] OpenJS Foundation. *Node.js Documentation*. URL: https://nodejs.org/api/ (besucht am 13. 12. 2025).

[44] *OWASP Top 10:2021 – A01: Broken Access Control*. OWASP. URL: https://owasp.org/Top10/2021/A01_2021-Broken_Access_Control/ (besucht am 14. 12. 2025).

[45] Podman. *Podman Documentation*. URL: https://docs.podman.io/ (besucht am 13. 12. 2025).

[46] *RDF 1.1 Concepts and Abstract Syntax*. W3C, 2014. URL: https://www.w3.org/TR/rdf11-concepts/ (besucht am 14. 12. 2025).

[47] *RDF 1.1 Turtle: Terse RDF Triple Language*. W3C, 2014. URL: https://www.w3.org/TR/turtle/ (besucht am 14. 12. 2025).

[48] RDForms. *RDForms Documentation*. URL: https://rdforms.org/ (besucht am 13. 12. 2025).

[49] React Team. *React Documentation*. URL: https://react.dev/learn (besucht am 13. 12. 2025).

[50] *Serving static files in Express*. Express. URL: https://expressjs.com/en/starter/static-files.html (besucht am 14. 12. 2025).

[51] *Session Management Cheat Sheet*. OWASP. URL: https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html (besucht am 14. 12. 2025).

[52] SurveyJS. *SurveyJS Form Library Documentation*. URL: https://surveyjs.io/form-library/documentation/overview (besucht am 13. 12. 2025).

[53] ULB Darmstadt. *shacl-form*. URL: https://github.com/ULB-Darmstadt/shacl-form (besucht am 13. 12. 2025).

[54] Vercel. *Next.js Documentation*. URL: https://nextjs.org/docs (besucht am 13. 12. 2025).

[55] *Vite Guide: Building for Production*. Vite. URL: https://vite.dev/guide/build (besucht am 01. 12. 2025).

[56] *Vite Guide: Deploying a Static Site*. Vite. URL: https://vite.dev/guide/static-deploy (besucht am 01. 12. 2025).

[57] W3C. *Accessible Rich Internet Applications (WAI-ARIA)*. URL: https://www.w3.org/TR/wai-aria/ (besucht am 13. 12. 2025).

[58] W3C. *CSS Snapshot 2025*. URL: https://www.w3.org/TR/css-2025/ (besucht am 13. 12. 2025).

[59] W3C. *RDF 1.1 Concepts and Abstract Syntax*. URL: https://www.w3.org/TR/rdf11-concepts/ (besucht am 13. 12. 2025).

[60] W3C. *RDF 1.1 Turtle*. URL: https://www.w3.org/TR/turtle/ (besucht am 13. 12. 2025).

[61] W3C. *Shapes Constraint Language (SHACL)*. URL: https://www.w3.org/TR/shacl/ (besucht am 13. 12. 2025).

[62] W3C. *WAI-ARIA Authoring Practices (APG)*. URL: https://www.w3.org/WAI/ARIA/apg/ (besucht am 13. 12. 2025).

[63] Barbara Walther u. a. „Comparison of Electronic Data Capture (EDC) with the Standard Data Capture Method for Clinical Trial Data“. In: *PLOS ONE* 6.9 (2011), e25348. URL: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0025348.

[64] *Web Content Accessibility Guidelines (WCAG) 2.2*. W3C, 2023. URL: https://www.w3.org/TR/WCAG22/ (besucht am 14. 12. 2025).

[65] WHATWG. *Fetch Living Standard*. URL: https://fetch.spec.whatwg.org/ (besucht am 13. 12. 2025).

[66] WHATWG. *HTML Living Standard*. URL: https://html.spec.whatwg.org/ (besucht am 13. 12. 2025).

[67] Alex K. Wolfe und contributors. *json-fs-store: Simple Node.js file system storage for JSON objects (GitHub Repository)*. URL: https://github.com/alexkwolfe/json-fs-store (besucht am 14. 12. 2025).

[68] World Wide Web Consortium. *Web Content Accessibility Guidelines (WCAG) 2.1*. W3C Recommendation. W3C, Juni 2018. URL: https://www.w3.org/TR/WCAG21/ (besucht am 07. 12. 2025).

[69] World Wide Web Consortium. *Web Content Accessibility Guidelines (WCAG) 2.2*. W3C Recommendation. W3C, Okt. 2023. URL: https://www.w3.org/TR/WCAG22/ (besucht am 07. 12. 2025).

[70] Alkin Yurtkuran u. a. „Lean transformation to reduce costs in healthcare: A public hospital case in Turkey“. In: *Journal of Hospital Administration* 6.4 (2017). Fallstudie zur Verkürzung von Durchlaufzeiten im rechnungsbezogenen Prozess eines Universitätskrankenhauses. DOI: 10.5430/jha.v6n4p10. URL: https://doi.org/10.5430/jha.v6n4p10 (besucht am 07. 12. 2025).