Abschlussprüfung Sommer 2021

Fachinformatiker für Anwendungsentwicklung

Dokumentation zur betrieblichen Projektarbeit

Entwicklung eines Wizards für den Import von Auswertungsdaten im IT-System „Wirkungsdaten“

Prüfungsbezirk

Frankfurt fi 10 (AP T2V1)

Prüfungsbewerber:

Tim Rhein

Raimundstraße 8

60431 Frankfurt am Main

Identnummer: 1236827

E-Mail: tim.rhein@giz.de

Ausbildungsbetrieb:

Deutsche Gesellschaft für Internationale Zusammenarbeit (GIZ) GmbH

Projektbetreuer: Peter Eichenauer

E-Mail: peter.eichenauer@giz.de

giz

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

BMZ	Bundesministerium für wirtschaftliche Zusammenarbeit und Entwicklung
CSS	Cascading Style Sheets
DOM	Document Object Model
ERM	Entity-Relationship-Modell
GIZ	Gesellschaft für Internationale Zusammenarbeit
GUI	Graphical User Interface
HTML	Hypertext Markup Language
JPA	Jakarta Persistence API (Hibernate)
JS	JavaScript
JSP	Jakarta Server Pages
MVC	Model View Controller
POJO	Plain Old Java Object
SQL	Structured Query Language
UI	User Interface
URL	Uniform Resource Locator

1 Einleitung

Die folgende Projektdokumentation ist Teil der IHK-Abschlussprüfung und befasst sich mit dem Ablauf der betrieblichen Projektarbeit, welches der Autor im Rahmen seiner Ausbildung zum Fachinformatiker mit Fachrichtung Anwendungsentwicklung durchgeführt hat. Dieses Projekt wird in der Firma Deutsche Gesellschaft für Internationale Zusammenarbeit (GIZ) GmbH durchgeführt, welche der Ausbildungsbetrieb ist. Die GIZ ist eine Organisation der Entwicklungszusammenarbeit und agiert im Auftrag mehrerer Ministerien der Bundesrepublik Deutschland. Zu ihrem Hauptauftraggeber gehört das Bundesministerium für wirtschaftliche Zusammenarbeit und Entwicklung (BMZ). Die GIZ hat weltweit 90 Standorte, davon sind 6 in Deutschland. Insgesamt beschäftigt sie über 22.000 Mitarbeiterinnen und Mitarbeiter in 120 Ländern.

1.1 Projektbeschreibung

Im Rahmen des Projektes sollte ein Wizard für den Import von Auswertungsdaten im IT-System „Wirkungsdaten“ entwickelt werden. Es werden die Daten von laufenden und abgeschlossenen Vorhaben ab einem bestimmten Auftragswert, die von BMZ sowie im Drittgeschäften beauftragt wurden, zur jährlichen Plausibilisierung in der Web Anwendung zur Erhebung der GIZ Wirkungsdaten eingetragen. Der Fachbereich, Stabsstelle Evaluierung, erhebt die GIZ Wirkungsdaten, um die Wirkungen der Arbeit der GIZ Vorhabens- und Landesebene zu messen. Die Auswertungsdaten werden automatisiert in einer CSV-Datei gespeichert und werden von mehreren Bereichen plausibilisiert. Danach aggregiert der Fachbereich die plausibilisierten Daten und disaggregiert nach Regionen und Ländern. Anschließend müssen die Daten wieder importiert werden, dies können momentan nur die technischen Systemverantwortlichen durchführen. Die existierende Importfunktion ist zudem für Administratorinnen und Administratoren nicht im Hauptmenü verlinkt, sondern nur über eine manuelle Eingabe der URL erreichbar.

1.2 Projektziel

Ziel des Projektes war es, eine browserbasierte, benutzerfreundliche grafische Oberfläche für den Import der Wirkungsdaten zu entwickeln. Die Import-Historie sollte in der bereits vorhandenen Datenbank abgelegt werden. Die Stabstelle Evaluierung soll zukünftig die technischen Systemverantwortlichen ablösen und als administrativer Benutzerkreis die Import Funktion übernehmen. Zudem sollte bei der Bearbeitung des Imports ein Fortschritts-Symbol angezeigt werden, um den Benutzerinnen und Benutzern Rückmeldung zu dem Verarbeitungsprozess zu geben.

1.3 Projektbegründung

Die Importfunktionalität wird jährlich von den technischen Systemverantwortlichen bedient, welches zusätzlich ein hoher Zeitaufwand ist. Der Fachbereich soll die Aufgabe des Imports zukünftig übernehmen, da dies zu dem Prozess der Erhebung der GIZ Wirkungsdaten gehört.

1.4 Projektschnittstellen

Da die Wirkungsdaten Webanwendung ein bestehendes System ist, besitzt sie bereits Schnittstellen. Eine Verbindung ist Microsoft Active Directory, welche das Benutzerverzeichnis darstellt. Des Weiteren besteht eine Verbindung zu SAP PBS, welches ein System zur Projekt Bearbeitung ist.

1.5 Projektabgrenzung

Der Projektumfang ist begrenzt, aus diesem Grund wurde nur eine Erweiterung der bestehenden Anwendung in der Projektarbeit umgesetzt. Des Weiteren wurde die Webanwendung Wirkungsdaten in dem Zeitraum der Projektarbeit des Autors nicht weiterentwickelt, weshalb die Abgrenzung der bestehenden Anwendung und der Erweiterung transparent sind.

1.6 Projektumfeld

Das Projekt findet in der Gruppe D250 - Web- und Individualanwendungen statt, welche momentan 14 Mitarbeiterinnen und Mitarbeiter hat. Im Rahmen einer jährlichen Datenerfassung ist eine Plausibilisierung durch circa 20 Mitarbeiterinnen und Mitarbeiter der Stabstelle Evaluierung und des Fach- und Methodenbereichs nötig.

2 Projektplanung

2.1 Projektphasen

Für die Umsetzung des Projektes hat die IHK Frankfurt 70 Stunden vorgeschrieben. Zu Anfang des Projekts, wurde eine Aufteilung der verschiedenen Phasen des Prozesses der gesamten Entwicklung niedergeschrieben. Eine grobe Zeitplanung mit den Hauptphasen und eine detaillierte Zeitplanung mit jeweiligen Prozessschritten befindet sich unter *A.1: Grobe Zeitplanung* auf Seite ii.

2.2 Ressourcenplanung

Für die Durchführung des Projektes war der Auszubildende Tim Rhein zuständig. Der Auftraggeber beziehungsweise Projektbetreuer Peter Eichenauer stand bezüglich Fragen und Problemen zur Verfügung. Für das Projekt Wirkungsdaten wurden innerbetriebliche Coding-Richtlinien zugrunde gelegt. Dazu gehören unter anderem die Entwicklungsumgebung. Aufgrund der strategischen Ausrichtung im Unternehmen wurde die Anwendung in Java programmiert. Die Ressourcenplanung umfasst neben allen sachlichen Ressourcen, die Softwareressourcen und personellen Ressourcen, welche im Anhang *A.3: Verwendete Ressourcen* auf Seite iii vorliegt.

2.3 Entwicklungsprozess

Grundsätzlich hat sich die Umsetzung des Projektes an den agilen Prozessmethoden von Scrum orientiert. Da im Rahmen des Projektes lediglich 70 Stunden vorgegeben sind, ist diese zeitliche Basis zu gering, um die Bestandteile des Scrum-Frameworks im vollen Umfang einzusetzen. In abgewandelter Form sind die Ereignisse Sprint Planning, Daily Scrum und Sprint Review zum Einsatz gekommen. Von den Artefakten ist das vorhandene Product Backlog durch die neuen Anforderungen als User Stories erweitert worden. Durch den agilen Prozess war es möglich vor Beginn der Bearbeitung von Aufgaben, wie auch am Ende eines Bearbeitungsschrittes, regelmäßig Rücksprache zu halten. Die Zyklen haben in der Regel eine Woche beansprucht. Allerdings sind bei Problemen oder vorzeitigem Abschluss von Aufgaben, zusätzliche Meetings erfolgt. Wegen der anhaltenden Pandemie Situation wurden diese via Teams abgehalten.

3 Analysephase

3.1 Ist-Analyse

Wie bereits in *1.1 Projektbeschreibung* erwähnt setzt sich das Projekt GIZ Wirkungsdaten aus mehreren Teilbereichen zusammen. In der Ist-Analyse soll nun herausgearbeitet werden, wie die Daten erhoben werden und welche Verarbeitungsschritte diese durchlaufen.

Verantwortlich für die Koordination der Erhebung der Wirkungsdaten ist die Stabsstelle Evaluierung. Sie stellt sicher, dass die Daten nach den verschiedenen Bedürfnissen aggregiert werden können, um Aussagen zu ausgewählten Wirkungen der Arbeit der GIZ zu treffen und sie für die Kommunikation mit der Öffentlichkeit und unserer Auftraggeber einzusetzen. Zur jährlichen Plausibilisierung muss jedes Projekt seine Daten in der Webanwendung „Wirkungsdaten“ zur Erhebung der GIZ Wirkungsdaten eintragen. In der Anwendung werden Aggregationsindikatoren abgefragt, welche so formuliert sind, möglichst viele Vorhaben eines Sektors zuliefern zu können, um eine optimale Erfassung, der Wirkungen der Projekte in ihren jeweiligen Ländern, zu erzielen. Zu den Aggregationsindikatoren gehören beispielsweise Beschäftigung, Ländliche Entwicklung, Ernährungssicherung und Gesundheit. Mit den Aggregationsindikatoren wird keine Zielerreichung gemessen, sondern der Ist-Zustand zum Zeitpunkt der Datenerhebung abgebildet. Sobald die Erhebungsfrist abgeschlossen ist, werden alle Daten automatisiert in einer CSV-Datei zusammengefasst und gespeichert. Diese Funktion ist implementiert und wird von der Stabsstelle Evaluierung durch das Exportieren in der Anwendung erzielt. Unter *A.4: Testdaten CSV-Datei* auf Seite iv ist ein Ausschnitt einer CSV-Datei mit Testdaten dargestellt. Eine CSV-Datei aus der Produktion ist aus Datenschutzrechtlichen Gründen nicht möglich zu demonstrieren. Zudem umfasst diese zu viele Daten, um sie in einem Screenshot festzuhalten. Insgesamt sind es 669 Spalten zu unterschiedlichen Indikatoren und über 1000 Zeilen mit Einträgen der Projekte zu den jeweiligen Indikatoren.

Nach der Datenerhebung werden die Rohdaten auf Basis der CSV-Datei von mehreren Bereichen plausibilisiert. Anschließend aggregiert die Stabstelle Evaluierung die plausibilisierten Daten und disaggregiert diese nach Regionen und Ländern. In dem aktuellen Zustand hatten lediglich die technischen Systemverantwortlichen die

Möglichkeit die CSV-Datei mit hohem Zeitaufwand zu importieren. Die benötigten Admin-Berechtigungen besitzt der Fachbereich bereits, doch der Zugriff auf die Importfunktionalität wird lediglich von den technischen Systemverantwortlichen durchgeführt. Zudem ist die Importfunktion für Administratorinnen und Administratoren nicht im Hauptmenü verlinkt, sondern nur über die manuelle Eingabe der URL zu erreichen. Auch zeigt die Importfunktionalität die importierte CSV-Datei sehr spartanisch an, was für den Fachbereich in der Benutzung zu Problemen führen würde. In einem vorhandenen Stylesheet ist vorgeschrieben, welche Grundlagen eingehalten werden müssen.

3.2 Soll-Konzept

Ziel des Projektes ist es, dass der Fachbereich Stabsstelle Evaluierung die Importfunktionalität übernehmen soll. Zur Führung von der Startseite zu der Importfunktion wird eine Navigation gebraucht. Die neue Oberfläche der Importfunktion, wie die Navigation, sollen sich nahtlos in das vorhandene Design der Anwendung einfügen. Bei der Ausführung der Importfunktion soll ein Element, beziehungsweise ein Fortschritts-Symbol angezeigt werden, welches den Fortschritt des Importvorgangs darstellt. Für jeden Importvorgang soll ein Datenbankeintrag mit Datum, Dateiname, User und Fehler als Historie in der Datenbank gespeichert werden. Anzumerken ist, dass die vorhandene Datenbankstruktur zu nutzen und zu erweitern ist und keine neue entworfen wird.

3.3 Wirtschaftlichkeitsanalyse

3.3.1 „Make-or-Buy“ - Entscheidung

Die Anwendung zum Verarbeiten der Daten existiert bereits, wodurch die Erweiterung intern oder an einen externen Entwickler abgegeben wird. Dies würde bedeuten, dass Mitarbeiterinnen und Mitarbeiter involviert werden, um dem Externen Lizenzen zu geben für Jira, wie auch das zur Verfügung stellen eines Laptops und das Einrichten, damit der Externe ordnungsgemäß nach GIZ Richtlinien das Projekt entwickeln kann. Der daraus resultierende Aufwand und die Kosten wären zu hoch. Eine grobe Auflistung befindet sich in *A.5: „Make-or-Buy“ Entscheidung* auf Seite iv. Aus diesem Grund wurde sich bei diesem Projekt für eine Eigenentwicklung entschieden.

3.3.2 Projektkosten

Im Folgenden sollen die Kosten, welche während des Projektes anfallen kalkuliert werden. Die zu kalkulierenden Kosten setzten sich aus Ressourcen und dem Personalaufwand zusammen. Ressourcen, die vor der Projektdurchführung bereits in Verwendung waren, sind unabhängig von der Durchführung des Projektes angefallen und wurden dementsprechend nicht in die Kostenrechnung mit aufgenommen. In Hinblick auf die Kosten für die Umsetzung des Projektes wurde nur Open Source Software verwendet, so dass keine zusätzlichen Lizenzkosten angefallen sind. Aus diesem Grund haben die Ressourcen in der nachfolgenden Kalkulation keine weiteren Kosten verursacht. Der Stundenverrechnungssatz für die Personalkosten eines Mitarbeiters wurde einem internen Dokument entnommen. Die Durchführungszeit des Projektes betrug 70 Stunden. In der Kostenaufstellung unter *A.6: Projektkosten,* wurden die zu kalkulierenden Kosten des Personalaufwandes nach Projektvorgängen aufgelistet, sowie summiert dargestellt.

3.3.3 Amortisierungsdauer

Bei der Amortisierungsdauer, soll betrachtet werden, ab welchen Zeitpunkt die Entwicklung des Projektes sich amortisiert. Die Kosten für die Durchführung des Projektes wurden in *3.3.2 Projektkosten* kalkuliert. Anhand von Erfahrungen aus den letzten Durchläufen der Datenerhebung wurde kalkuliert, dass die technischen Systemverantwortlichen für den Import der CSV-Datei 1 bis 2 Tage, abhängig von Fehlimporten, benötigen. Zudem findet die Datenerhebung 1-mal pro Jahr statt.
Mit der neuen Erweiterung in der Anwendung, kann nun die Stabsstelle Evaluierung den Import anstoßen. Durch die übersichtlichere Oberflächengestaltung lässt sich Zeit bei der Überprüfung einsparen und die Kommunikation zwischen Fachseite und technischen Systemverantwortlichen entfällt, wodurch die Summe der eingesparten Zeit 4 Stunden entspricht. Daraus ergibt sich folgende Gleichung zur Amortisationszeit:

$$\text{Amortisationsdauer} = \frac{985\ €}{4\ \frac{h}{Jahr} * 92 \frac{€}{h}} \approx 2{,}676\ Jahre\ \approx\ 32\ Monate$$

Nach nahezu 32 Monaten sind die Kosten zur Entwicklung der Erweiterung durch die Zeiteinsparung gedeckt.

3.4 Anwendungsfälle

Für die Übersicht, wie die Anwenderinnen und Anwender mit der Anwendung arbeiten und welche Fälle aus Endanwendersicht abgedeckt werden müssen, wurden zwei Use-Case-Diagramme erstellt. Im Fall des Projektes gehören lediglich die Administratorinnen und Administratoren zu den Akteuren, welche die Daten bei der Erhebung einpflegen beziehungsweise importieren. Um zu verdeutlichen, welche Akteure es gibt, befinden sich die Use-Case-Diagramme in vereinfachter Form in *A.7: Use-Case-Diagramme* auf Seite v. Da die Anwendung GIZ Wirkungsdaten enorm viele Funktionen bereitstellt, wurden dem Akteur Mitarbeiterinnen und Mitarbeiter nur wenige Funktionen zur Darstellung hinzugefügt.

3.5 Lastenheft

Aufbauend auf der Analysephase wurde das Lastenheft in Form von User Stories und zusätzlicher Beschreibung in Jira gemeinsam mit dem Projektbetreuer Peter Eichenauer erstellt. Das Lastenheft befindet sich im Anhang unter *A.8 Lastenheft* auf Seite vii.

4 Entwurfsphase

4.1 Zielplattform

Aufgrund der strategischen Ausrichtung im Unternehmen wird die Anwendung in Java programmiert. Als Spezifikation zur Entwicklung wird die Anwendung in Java 11 entwickelt, da diese schon in dem Projekt zum Einsatz kommt. Für das Projekt Wirkungsdaten wurden innerbetriebliche Coding-Richtlinien zugrunde gelegt. Dazu gehören unter anderem Entwicklungsumgebung, Plugins, Bibliotheken, Formatierungs- und Benennungsvorgaben. Daher wird als Entwicklungsumgebung Eclipse IDE 2019-06 genutzt. Des Weiteren muss diese Webanwendung auf dem Browser Microsoft Edge (derzeit Version 88.0.705.53) lauffähig sein.

4.2 Architekturdesign

Als Basis des Architekturdesigns für das Projekt, soll das Model View Controller (MVC) Konzept dienen. Dieses Muster stellt eine Unterteilung der Anwendung in das Model (Datenmodell), View (Darstellung der Daten) und den Controller (Steuerung der

Anwendung) dar. Das Model, welches die Daten hält, wird durch die Datenbank und dessen JPA Entitäten abgebildet. Zu der Modelschicht gehören auch das Repository und der Service, welche vom Controller aufgerufen werden, um Daten in der Datenbank zu persistieren. Die View übernimmt das Frontend der Webanwendung und ist lediglich zur Annahme und Ausgabe von Daten, die jeweils an oder aus dem Backend übergeben werden. Der Controller reagiert auf die Anfragen des Frontend und diese werden im Backend durch eine Java Klasse verarbeitet. Durch die Nutzung des MVC Architekturmusters erfolgt eine Teilung des Quellcodes, wodurch die einzelnen Komponenten unabhängig voneinander angepasst werden können und die Übersichtlichkeit verbessert wird. Ein Schaubild der MVC Architektur ist im Anhang unter *A.9: Architekturmuster des MVC* auf Seite viii dargestellt.

4.3 Benutzeroberflächenentwurf

Wie unter *3.2 Soll-Konzept* beschrieben, soll sich das Design nahtlos in das der bestehenden Anwendung einfügen, damit der Fachbereich ohne weiteres mit der Import-Funktion arbeiten kann. Hierfür wurden zuerst Mockups entworfen, welche sich im Anhang unter *A.10: Mockups* auf Seite viii befinden.
Die Webanwendung soll wie oben beschrieben, einen einheitlichen Aufbau haben, bei welchem der Header aus dem Logo der GIZ, dem Begriff Intranet und der Auswahl der Sprachen Deutsch oder Englisch besteht. Um den Header zum Body abzugrenzen, soll der Header abschließend einen grauen Balken besitzen, in dem die Navigation ist. Der Bereich, in dem sich die Benutzerin oder der Benutzer befindet, wird hervorgehoben. Im Hauptmenü werden der Benutzerin oder dem Benutzer nur bestimmte Funktionalitäten angezeigt. Für Administratorinnen und Administratoren mit den entsprechenden Berechtigungen werden alle Funktionen freigegeben. Der Abschluss des Inhaltes beziehungsweise der Footer, soll wie im Header den grauen Balken besitzen, welcher die Informationen der Anwendung wiederspiegelt.

4.4 Datenbankmodell

Das Datenbankmodell besteht aus vielen Entitäten, um diese nicht alle aufzuzählen, befindet sich im Anhang unter *A.11: ER Modell* unter der Seite xi ein ERM, welches die gesamte Datenbank der Anwendung abbildet. Für das Projekt soll für das Speichern der Historie des Imports eine Erweiterung durchgeführt werden. Die Entität *ImportHistory*

speichert dabei die Attribute Datum, Dateiname, User und Fehler um Auskunft über die Importe, welche stattgefunden haben, wiederzugeben. Bei der Erweiterung muss eine *Many-To-One* Beziehung (1:n-Beziehung) zwischen der Entität *ApplicationUser* und *ImportHistory* bestehen. Da eine Administratorin oder ein Administrator mehrere Importvorgänge durchführen kann, aber ein Import immer nur zu einer Administratorin oder einem Administrator gehören kann. Diese Beziehung ist in einem vereinfachten Soll-Zustand ERM dargestellt (siehe *Abbildung 14: Vereinfachtes Soll-Zustand ERM*).

4.5 Geschäftslogikentwurf

Die Geschäftslogik setzt sich hauptsächlich aus der Klasse *ImportController* zusammen, welcher für das Annehmen, Weiterverarbeiten und Zurückgeben der Daten zuständig ist. Es muss eine Schnittstelle ermöglicht werden, um die Eingabedaten aus dem Frontend, in die Datenbank zu persistieren. Dieser Weg muss auch umgekehrt möglich sein, um die Daten aus der Datenstruktur, im Frontend anzuzeigen. Zusätzlich muss die Operation des Fortschritts-Symbol berechnet werden und über einen Asynchronen Aufruf dem Frontend der erreichte Fortschritt kommuniziert werden.

4.6 Qualitätssicherung

Die Funktionalität der Geschäftslogik soll durch automatisierte JUnit Tests mit einer Abdeckung von 70% gewährleistet werden. Fehler, die durch den neu implementierten Code auftreten, kann so vorgebeugt, beziehungsweise schnell behoben werden. Bei der Funktionalität der Benutzeroberfläche soll der UI-Test rudimentär erfolgen, sprich die Anwendung muss vom Programmierer ausführlich getestet werden.

4.7 Pflichtenheft

Das Pflichtenheft wurde in Form von User Stories festgehalten. Die User Stories wurden am Ende der Entwurfsphase in Jira aufgenommen und beschreiben die konkreten fachlichen und technischen Details der Umsetzung der Anforderungen. Zusätzlich wird Auskunft über abgeschlossene Kriterien der Durchführung des Projektes gegeben. Unter *A.12: Pflichtenheft* auf Seite xii sind die Akzeptanzkriterien der User Stories abgebildet.

5 Implementierung

5.1 Implementierung der Benutzeroberfläche

Die Benutzeroberfläche konnte auf der Grundlage von den Mockups, welche unter *4.3 Benutzeroberflächenentwurf* beschrieben sind, mit kleineren Abweichungen implementiert werden. Die Überschrift „Import Rohdaten“ wurde in „Datenimport“ geändert und in die bestehende Navigation übernommen, anstatt wie in den Mockups dargestellt, nur als Überschrift verwendet zu werden (siehe *A.10: Mockups*). Zudem wurde der Button „Import Historie anzeigen“ (siehe *Abbildung 8: UI der Import-Funktion*) nicht implementiert. Stattdessen wird die Tabelle der Import-Historie bei erstmaligen Aufrufen der Seite dargestellt. Das Umsetzen der Oberfläche ist in Jakarta Server Pages (JSP) erfolgt. Diese erzeugen Server-seitig dynamisches HTML. Der Java Code und JavaScript Code wurde zur besseren Übersicht in separierten Klassen implementiert, welcher im Detail unter *5.3 Implementierung der Geschäftslogik* behandelt wird. Zur einheitlichen Gestaltung der Seiten sind die JSP-Dateien ähnlich aufgebaut und Elemente, die öfters verwendet wurden, sind über einen HTML-Tag <*include*> eingebunden. Zusätzlich gibt es Cascading Style Sheets (CSS), welche ebenso in den JSP-Dateien eingebunden sind. Der JSP-Datei für das Import Wizard wurde eine weitere CSS-Datei für das Design des Fortschrittsbalken, der Ausgabe der importierten Rohdaten und der Import Historie hinzugefügt, um diese von den vorhanden Vorlagen zu separieren. Dieses Stylesheet ist im Anhang *A.13: CSS-Datei (Ausschnitt)* auf Seite xiv zu finden. Für das Implementieren der Navigation wurde ein vorhandener Teil der Navigation, welcher nicht mehr verwendet wurde recycelt. Dabei wurde die JSP-Datei, die für das Import Wizard notwendig war, in die Bereiche des vorherigen Navigationselement eingebunden. Des Weiteren wurde für die Überprüfung auf die Rolle von Administratorinnen und Administratoren eine Anpassung im Backend implementiert, welche im Abschnitt *5.3 Implementierung der Geschäftslogik* beschrieben wird. Zusätzlich musste der tote Code aus der gesamten Wirkungsdaten Anwendung entfernt werden. Zur Manipulation des *DOM,* um Eingabewerte aus Feldern auszulesen, wurde die JavaScript-Bibliothek jQuery genutzt. Dieses ermöglicht es viele Funktionen des klassischen JavaScript vereinfacht zu nutzen, wie beispielsweise das Abrufen oder Hinzufügen von Werten. Die Kommunikation mit dem Backend erfolgt

durch jQuery. Beim Aufrufen der Seite, wird zusätzlich zu dem grundlegenden Design, die Import Historie in Form von einer Tabelle angezeigt. Die Daten werden aus der Datenbank an das Frontend gesendet und über jQuery werden die jeweiligen Attribute dargestellt. Beim Upload wird die CSV-Datei über eine *POST*-Methode an den Sever zur Weiterverarbeitung gesendet. Der Importvorgang wird im Backend durchgeführt und in Form einer Tabelle an das Frontend zurückgegeben. Bei der Gestaltung der Oberfläche, wie auch in der gesamten Umsetzung wurden die Richtlinien beachtet. Dazu zählte im Bereich der Oberflächengestaltung, dass das Design nahtlos in das vorhandene übergeht. Im Anhang *A.14: JSP-Datei (Ausschnitt)* auf Seite xv ist die JSP-Datei zu sehen.

5.2 Implementierung der Datenstrukturen

Bei der Implementierung der Erweiterung der Datenbankstruktur, wurde sich an dem ERM (siehe *Abbildung 14: Vereinfachtes Soll-Zustand ERM*) orientiert. Dadurch, dass JPA den gesamten Zugriff auf die Datenbank kapselt, musste lediglich eine neue Klasse *ImportHistory* (siehe *Listings 3: Klasse ImportHistory.java*) in Java Code erstellt werden, um eine neue Tabelle in die vorhandene Datenbank zu integrieren. Beim Erstellen der Klasse war es notwendig, die Attribute mit ihren jeweiligen Datentypen, wie im ERM dargestellt, im Rumpf der Klasse als POJO Objekte zu deklarieren. Das beschreibt eine Klasse, die nur Daten hält, Getter und Setter besitzt, allerdings keine Logik abbildet. Zusätzlich mussten die Attribute durch die JPA-Annotation *@Column* gekennzeichnet werden und die Klasse als *@Entity*. Da die Klasse *ImportHistory* in einer 1:n Beziehung zur *ApplicationUser* Klasse steht, musste durch die Annotation *@JoinColumn(name=" ")* ein Fremdschlüssel zur Verknüpfung der Tabellen erstellt werden. Damit nun beim Starten der Anwendung die angelegte Klasse automatisiert von JPA in eine relationale Tabelle überführt wird, war es notwendig in den Einstellungen des Frameworks Hibernate die Konfiguration *hibernate.hbm2ddl.auto=update* umzustellen. Des Weiteren wurde eine Klasse *ImportHistoryRepository* (siehe *Listings 4: Interface ImportHistoryRepository.java*) angelegt, welche lediglich ein Interface beinhaltet, um die übergebenen Werte aus dem Frontend in die Datenbank zu persistieren. Zusätzlich musste ein Service (siehe *Listings 5: Klasse ImportHistoryService.java*) implementiert werden, welcher die

Datensätze bei einem angestoßenen Import vom Controller übergeben bekommt und diese an das eben erwähnte Repository weitergibt oder alle Einträge aus der Datenbank an den Controller liefert.

5.3 Implementierung der Geschäftslogik

Wie in unter *3.1 Ist-Analyse* erwähnt, existiert die Klasse *ImportController*, wie die dazugehörige Test-Klasse bereits. Damit nur Administratorinnen und Administratoren auf die Import-Funktion zugreifen können, musste die Rolle des *Admins* überprüft werden (siehe *Listings 8: Klasse ImportController.java*). Um die Rolle an das Frontend zu senden war ein *Model*-Objekt notwendig, welches Spring liefert. Dem *Model* kann jeder Typ von Daten zugewiesen werden. Am Ende der Verarbeitung vom *ImportController* wird das *Model* an das Frontend geliefert. Im Frontend kann durch jQuery auf die Objekte die im *Model* zugegriffen und ausgegeben werden. Für den Zugriff auf die Import-Historie, musste auf die im Service vorhandenen Methoden zugegriffen werden. Der Zugriff wurde durch die Annotation *@Autowired*, welche die automatische Injektion von Abhängigkeiten erzeugt, ermöglicht. Durch die Methode *getListOfImportHistory* im Service (siehe *Listings 5: Klasse ImportHistoryService.java)*, können alle Datenbank Einträge an den Controller übermittelt werden, welcher sie dem *Model* hinzufügt und an das Frontend zurückliefert. Der Grundstein für das Importieren der CSV-Datei war schon gelegt, aber in Kombination mit dem Fortschritts-Symbol musste dieser Aufruf asynchron verlaufen. Die Methode, welche für den Import zuständig ist, musste in die Klasse *ImportParserServiceImpl.java* ausgelagert werden. Durch die zusätzliche Annotation *@Async* kann die Methode asynchron zu dem Rest verlaufen (siehe *Listings 6: Auszug aus der Klasse ImportParserServiceImpl.java*). Zusätzlich war es notwendig die Klasse *WirkungsdatenConfiguration.java* um die Annotation *@EnableAsync* zu ergänzen. Diese Methode für den asynchronen Aufruf wird der Status einer statischen Variablen auf *true* gesetzt, wodurch das Frontend die Information bekommt, das Fortschritts-Symbol zu starten. In einer JavaScript Klasse wird die Berechnung durchgeführt (siehe *Listings 7: Datei dataImportPage.js*). Anhand der vorherigen Importvorgänge und einem eigenen Test mit Datensätzen aus der produktiven Anwendung, wurde der Durchschnitt einer bearbeiteten Zeile genommen, welcher mit den Zeilen der zu importierenden CSV-Datei multipliziert wird. Nach

erfolgreichem Importieren mussten die Attribute einer Import-Historie in die Datenbank persistiert werden. Dies erfolgt über das Aufrufen der Methode aus der Klasse *ImportHistoryService*. Dazu war es notwendig die Klasse *ImportController* anzupassen. Ein Ausschnitt der Klasse, welcher das Aufrufen der asynchronen Methode und das Persistieren der Import-Historie abbildet, befindet sich im Anhang (siehe *Listings 8: Klasse ImportController.java*). Sobald dies ausgeführt wird, übergibt das *Model* die importierten Daten an das Frontend. Im Anhang unter *A.16 Klassendiagramm (Ausscnitt)* auf Seite xxii ist ein Klassendiagramm, dass die Klassen in denen programmiert wurde, aufführt. Da die GIZ „Wirkungsdaten“ eine bestehende Anwendung ist, sind weitere Methoden in den Klassen, welche unabhängig von der Erweiterung sind, vorhanden.

5.4 Testen der Anwendung

Während der Implementierungsphase, wurden auch Testfälle geschrieben, um die implementierten Methoden auf Fehler zu untersuchen. Mit den JUnit-Tests wurden die Methoden der einzelnen Klassen auf Funktionalität überprüft (siehe *Listings 9: JUnit Test von ImportRepositoryTest.java*). Dieser überprüft das Persistieren durch eingespielte Testdaten. Um die produktive Datenbank nicht mit den Test-Daten zu manipulieren, wird auf der Test-Datenbank gespeichert. Des Weiteren wurde die Klasse *ImportController* getestet. In diesem Fall mussten die vorhandenen Testfälle an die veränderte Klasse angepasst werden.

6 Abnahme- und Deploymentphase

6.1 Code Review

Bei Abschluss des Projektes, konnte der Autor noch nicht wieder ins Büro, sodass das Code-Review über digitalem Weg erfolgte. Durch die iterativen wöchentlichen Zyklen hatte der Projektleiter Vorkenntnisse, wodurch das Code-Review schnell besprochen worden konnte. Bei der Überprüfung der Code Qualität wurde sich an der GIZ Java Entwicklungsrichtlinie orientiert. Dabei wurde auf fachliche und technische Fehler, wie auch die Benennung der Variablen geachtet. Durch das Recycling des Codes wurde auch darauf geachtet, dass kein „toten Code“ mehr vorhanden war. Die Anmerkungen wurden aufgenommen und sollen nach der Projektarbeit noch umgesetzt werden, um

die Qualität des Codes zu verbessern. Unter dem Punkt *8.3 Ausblick* wurden die Anmerkungen mit aufgeführt.

6.2 Abnahme

Wie im Code Review erwähnt, ist auch die Abnahme online erfolgt, welche im Anschluss des Code Reviews durchgeführt wurde. Durch den Wissensstand des Projektleiters konnte dieses zügig erfolgen. Per Mail wurde die Abnahme des Projektleiters schriftlich bestätigt.

6.3 Deployment

Aufgrund der anhaltenden Pandemie konnte ein finales Deployment nicht erfolgen. Sobald eine Rückkehr der Mitarbeiterinnen und Mitarbeiter in die Geschäftsstelle möglich ist, kann die Erweiterung mit wenig Aufwand deployed werden.

7 Dokumentation

Die Dokumentation wurde während des gesamten Verlaufes der Durchführung erstellt, da viele verwendete Anlagen und Inhalte, in der Planung entstanden sind. Sie gibt Aufschluss über den Aufbau und Ablauf des Projektes. Neben der Dokumentation, wurde auch eine Entwicklerdokumentation in Form von JavaDocs verfasst. Diese gibt einen detaillierten Einblick in alle implementierten Java Klassen. Darunter fallen Attribute, Methoden und deren Kommentare.

8 Fazit

8.1 Soll-Ist-Vergleich

Rückblickend konnten alle Anforderungen des Projekts in der von der IHK Frankfurt vorgegebenen Zeit umgesetzt werden. Die geringfügigen Abweichungen konnten in der Planung kompensiert werden. Eine grobe Abbildung des Zeitmanagements befindet sich im Anhang unter *A.17: Soll-Ist-Vergleich der groben Zeitplanung* auf Seite xxii. Die Implementierung nahm mehr Zeit ein als ursprünglich geplant, die durch die Analysephase und Entwurfsphase, welche weniger Zeit in Anspruch nahmen, kompensiert wurde. Auch die Tests der Oberfläche sind rudimentär ausgeführt worden und dies konnte während der Implementierung immer wieder getestet werden. Daraus

resultierend, hat das Testen der Oberfläche nicht die Zeit, die geplant worden war, in Anspruch genommen und kam der Implementierung zusätzlich zu gute.
Wie schon unter *6.1 Code Review* erwähnt hatte der Projektleiter durch die iterativen Zyklen einen gewissen Wissenstand über den Code, wodurch das Code Review schneller besprochen werden konnte. Die Zeitersparnis des Code Reviews und das ausgefallene Deployment wegen der Pandemie konnten der Dokumentation zugutekommen.

8.2 Lesson Learned

Die Umsetzung des Abschlussprojektes war für den Autor eine bereichernde Erfahrung. Durch die intensive Auseinandersetzung mit vielen komplexen technischen Mitteln konnten die bereits bestehenden Kenntnisse der Softwareentwicklung eingesetzt, aber auch erweitert werden. Besonders im Bereich der eigenständigen Planung und Herangehensweise wurden viele neue Kenntnisse erworben. In diesem Projekt ist dem Autor einmal mehr deutlich geworden, wie wichtig die Kommunikation mit dem Projektleiter ist, um die Akzeptanzkriterien der Kundenansprüche ordnungsgemäß zu erfüllen. Nach der Ansicht des Autors hatte das Projekt in vielerlei Hinsicht einen großen Mehrwert in Bezug auf den Ausbau seiner praktischen Kenntnisse der Softwareentwicklung.

8.3 Ausblick

In der vorgegebenen Zeit konnte eine lauffähige Erweiterung der Anwendung implementiert werden und alle gestellten Anforderungen an das Projekt wurden umgesetzt. Allerdings ist geplant, den Datenimport zusätzlich auf Englisch umzusetzen. Dies war kein Akzeptanzkriterium, da der Import nur von Administratorinnen und Administratoren ausgeführt wird, welche ausschließlich in den Geschäftsstellen in Deutschland arbeiten. Zudem wurde das Benutzerhandbuch nicht erweitert. Um die Administratorinnen und Administratoren mit der Import-Funktion vertraut zu machen, sollte ein Eintrag im Benutzerhandbuch und eine Videoanleitung via Adobe Captivate erstellt werden. Des Weiteren kann die Berechnung der Importdauer spezifischer implementiert werden, z.B. durch einen weiteren *GET* Request, welcher kontinuierlich den Staus der zu bearbeiteten CSV-Datei abfragt. Alle Erweiterungen können durch das verwendete MVC Konzept problemlos integriert werden.

Anhangsverzeichnis

Anhang

A.1 Grobe Zeitplanung

Projektphase	Geplante Zeit in Stunden (h)
Analyse	6h
Entwurf	8h
Implementierung	41h
Abnahme und Deployment	6h
Dokumentation	9h
Gesamt	**70h**

Tabelle 1: Grobe Zeitplanung

A.2 Detaillierte Zeitplanung

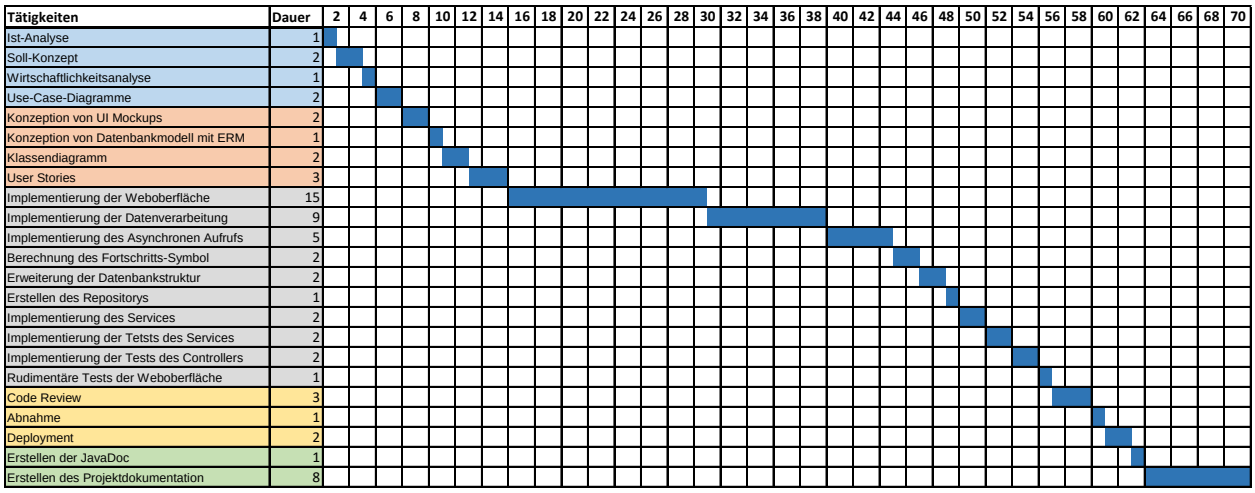


Abbildung 1: Detaillierte Zeitplanung in Form eines Gantt-Diagramms

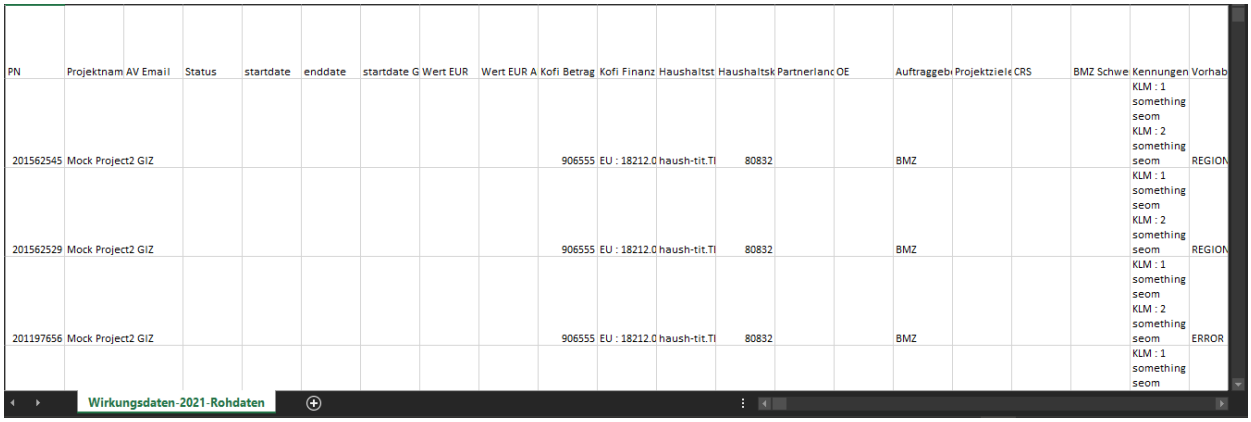
A.3 Verwendete Ressourcen

Ressourcenplanung
Hardware
Büroarbeitsplatz (zu Hause)
Laptop Fujitsu LIFEBOOK E Series
Software
Betriebssystem: Windows 10 Enterprise
Entwicklungsumgebung: Eclipse
Versionsverwaltung: SVN
Build Tool: Maven
Continous Integration: Jenkins
Test Framework: JUnit
Datenbank: Microsoft SQL
Anwendungsumgebung: Microsoft Edge
Entwicklungsserver: Tomcat 9
Diagrammerstellung: Draw.io (in Jira)
Programmiersprachen
Java
JavaScript
HTML (textbasierte Auszeichnungssprache)
CSS (Stylesheet-Sprache)
Bibliotheken
jQuery
JUnit 4
Mockito
Spring
Personal
Projektleiter: Auftraggeber des Projektes, Technischer Ansprechpartner, Abnahme und Code Review

Auszubildender: Durchführung und Entwicklung des Projektes

Tabelle 2:Ressourcenplanung

A.4 Testdaten CSV-Datei



PN	Projektnam	AV Email	Status	startdate	enddate	startdate G	Wert EUR	Wert EUR A	Kofi Betrag	Kofi Finanz	Haushaltst	Haushaltsk	Partnerlanc	OE	Auftraggeb	Projektziele	CRS	BMZ Schwe	Kennungen	Vorhab
201562545	Mock Project2	GIZ							906555	EU : 18212.0	haush-tit.TI	80832			BMZ				KLM : 1 something seom KLM : 2 something seom	REGION
201562529	Mock Project2	GIZ							906555	EU : 18212.0	haush-tit.TI	80832			BMZ				KLM : 1 something seom KLM : 2 something seom	REGION
201197656	Mock Project2	GIZ							906555	EU : 18212.0	haush-tit.TI	80832			BMZ				KLM : 1 something seom KLM : 2 something seom	ERROR
																			KLM : 1 something seom	

Wirkungsdaten-2021-Rohdaten

Abbildung 2: Ausschnitt von Testdaten in Form einer CSV-Datei

A.5 „Make-or-Buy“ Entscheidung

Intern	**Stundensatz**	**Zeitbedarf**	**Extern**	**Stundensatz**	**Zeitbedarf**
Herr Rhein	7,5 €	70 Stunden	Herr Eichenauer	92 €	16 Stunden
Herr Eichenauer	92 €	5 Stunden	Externer	87,5 €	50 Stunden
Summe in €	**985**		**Summe in €**	**5847**	

Tabelle 3: „Make-or-Buy“ - Entscheidung

A.6 Projektkosten

Mitarbeiter	**Stundensatz**	**Aufgaben**	**Zeitbedarf**	**Kosten in €**
Herr Rhein	7,5 €	Projektdurchführung	70 Stunden	525 €
Herr Eichenauer	92 €	Code-Review, Abnahme und Unterstützung des Projektes und	5 Stunden	460 €
Summe in €				**985 €**

Tabelle 4: Berechnung der Projektkosten

A.7 Use-Case-Diagramme

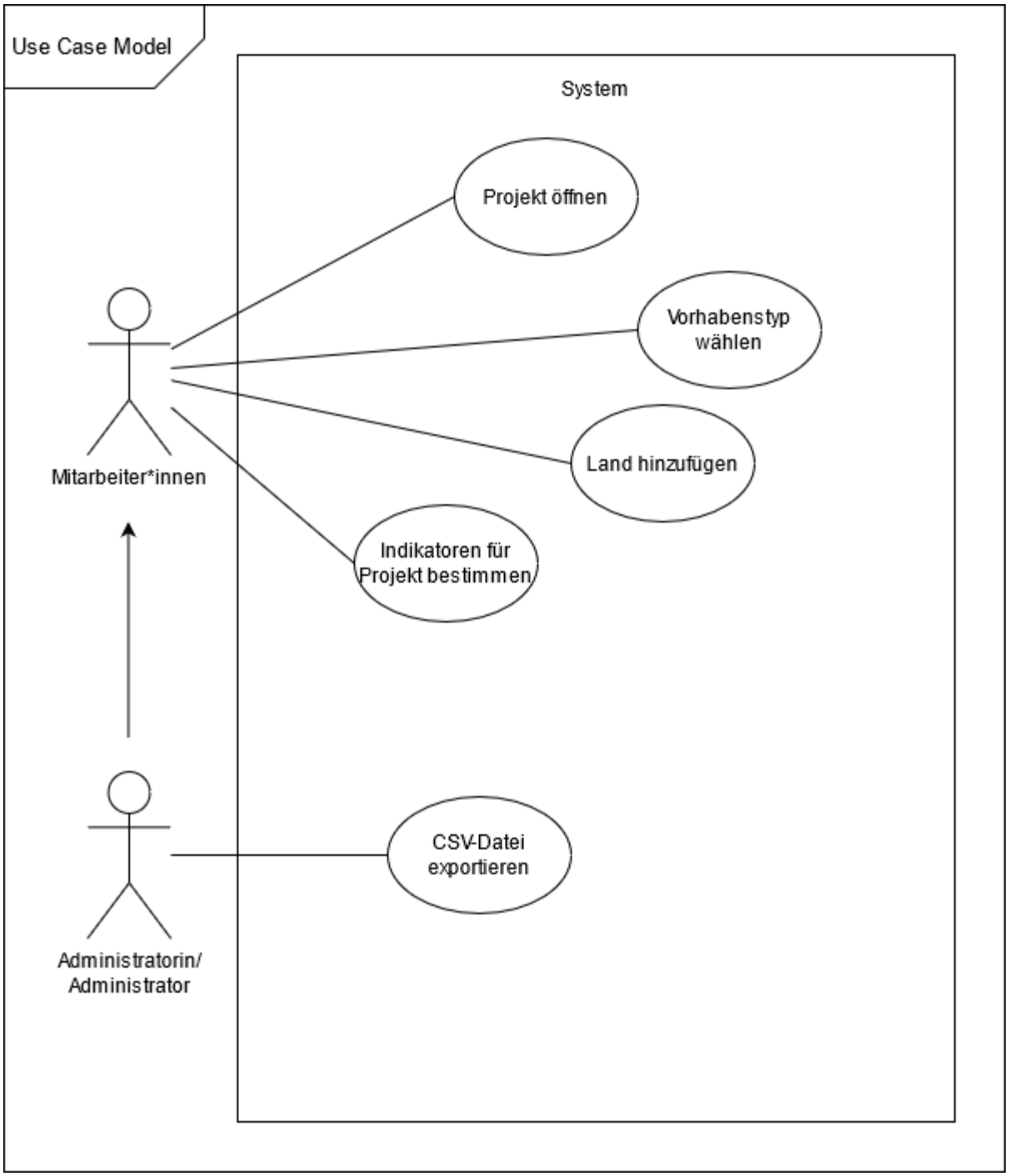


Abbildung 3: Vereinfachtes Anwendungsfalldiagramm im Ist-Zustand

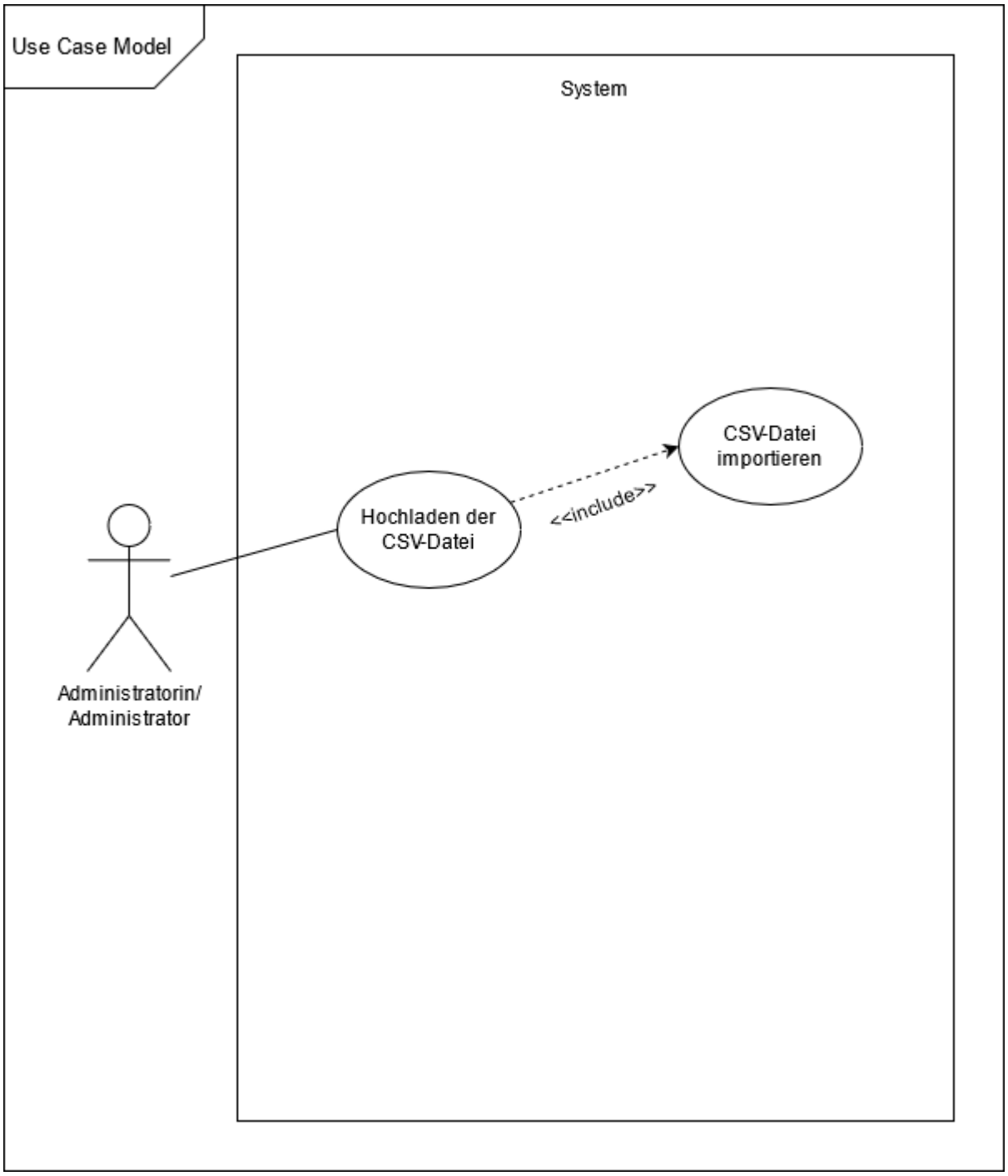


Abbildung 4: Anwendungsfalldiagramm im Soll-Zustand

A.8 Lastenheft

User story: Als Admin-User kann über eine Navigationspunkt "Datenimport", welche bei der vorhandenen Navigation verknüpft ist, selbstständig zu der Importfunktion gelangen um den jährlichen Import-Prozess durchzuführen zu können.

Description

Der Navigationspunkt wird in der vorhanden Navigation eingebunden mit identischen Designgrundstätzen. Der Navigationspunkt soll "Datenimport" benamt werden und nur für Administrartor*innnen sichtbar sein. Über den Navigationspunkt sollen Administrartor*innnen zu der Importfunktion gelangen um den jährlichen Plausibilierungsprozess durchzuführen.

Abbildung 5: User Story - Navigation zur Importfunktion

User story: Als Admin-User kann ich eine CSV-Datei mit korrigierten Rohdaten über die Weboberfläche importieren und erhalte verständliche Meldungen über Erfolg bzw. Fehler des Importvorgangs, um selbständig im jährliche Plausibilierungsprozess Daten korrigieren zu können

Description

Das Rohdatenset (CSV-Datei im Format des Export Rohdaten) kann von Admin-Usern importiert werden und korrigierte Daten werden in der Datenbank für die jeweilige PN/Land gespeichert und werden somit über die normale Projektansicht für alle User sichtbar.

Der Vorgang kann bei mehrere Stunden dauern, somit soll bei der Bearbeitung des Imports ein Fortschritts-Symbol angezeigt werden, um die User Rückmeldung zu dem Verarbeitungsprozess zu geben.

Die Importfunktion ~~WIDA-656~~ wird wiederverwendet,

Abbildung 6: User Story - Importfunktion für Korrekturen (Oberflächen Wizard)

A.9 Architekturmuster des MVC

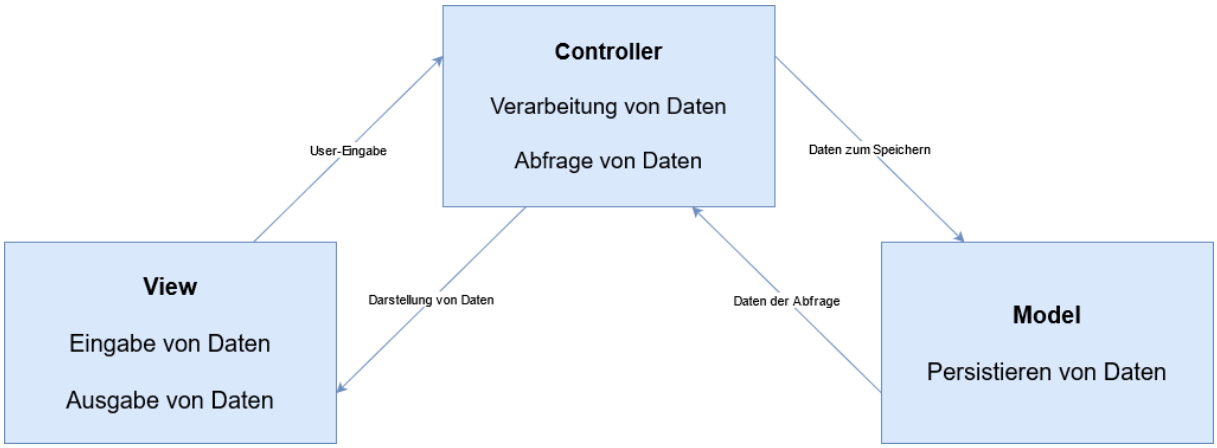


Abbildung 7: Architekturmuster des Model View Controller

A.10 Mockups

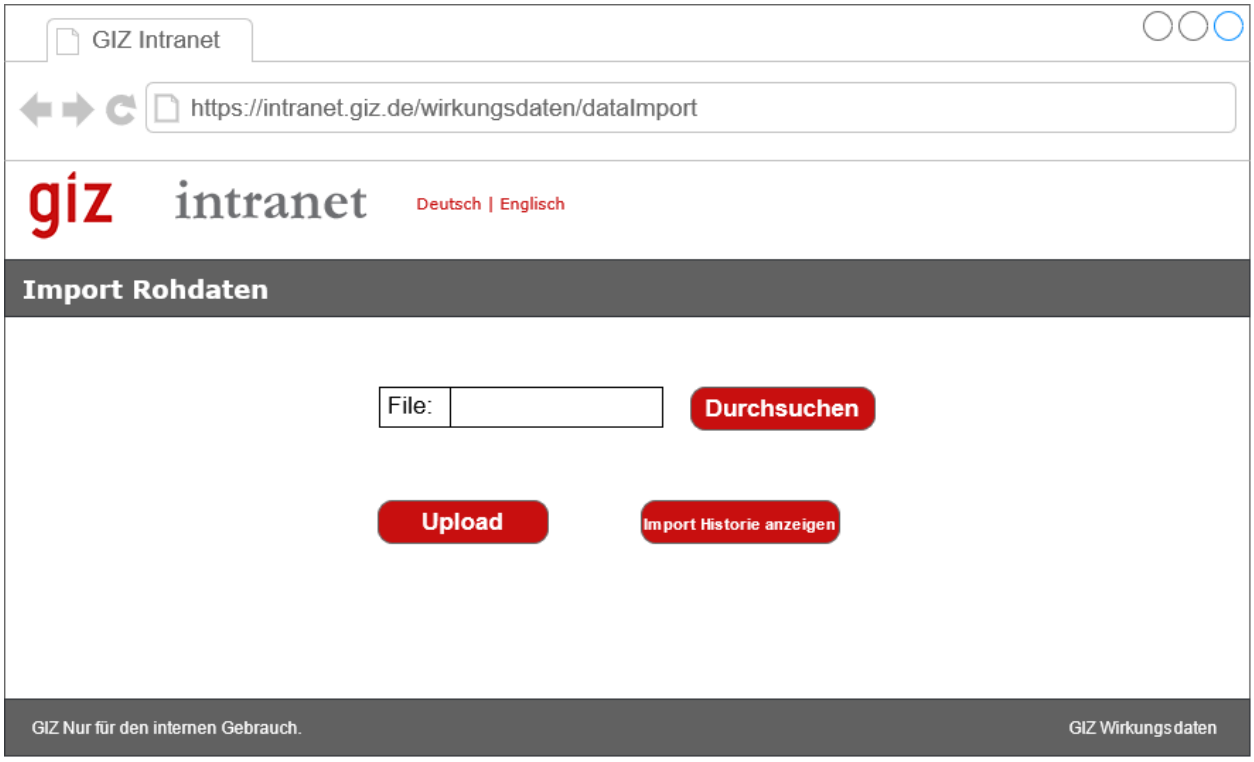


Abbildung 8: UI der Import-Funktion

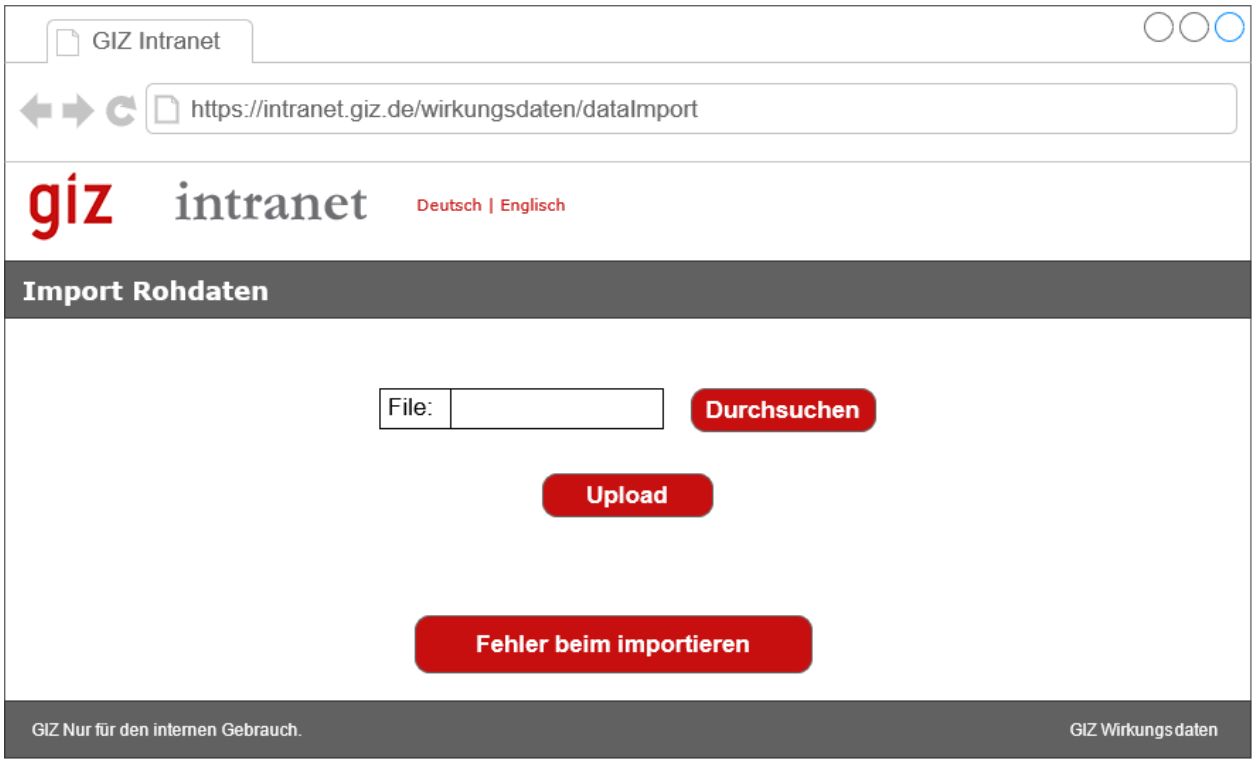


Abbildung 9: UI eines fehlgeschlagenen Imports

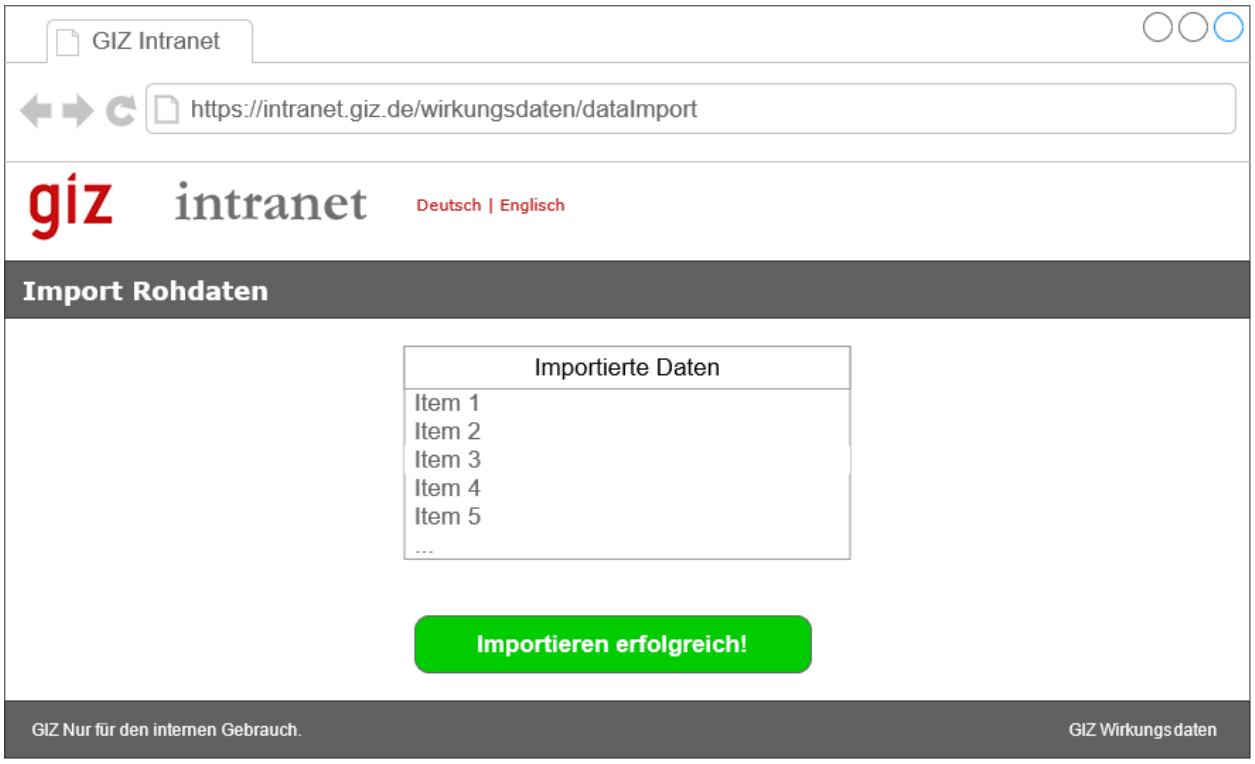


Abbildung 10: UI eines erfolgreichen Imports

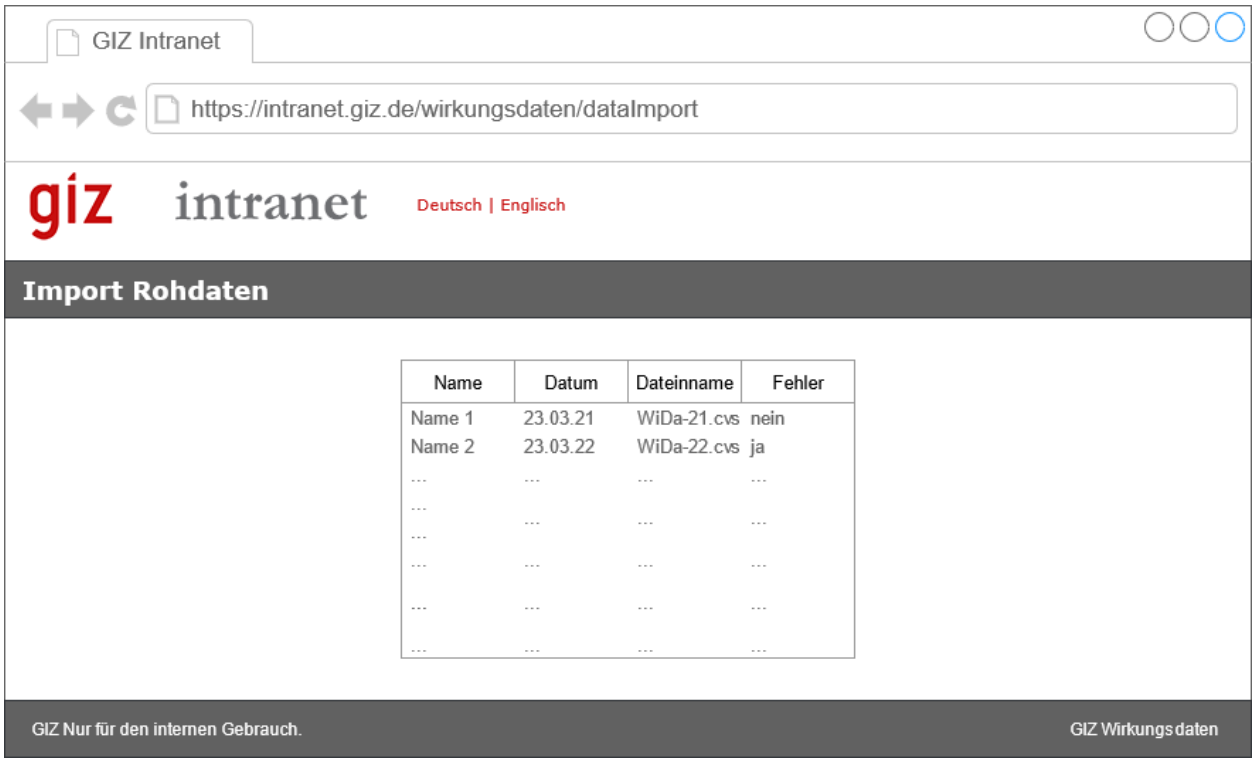


Abbildung 11: UI zum Anzeigen der Import-Historie

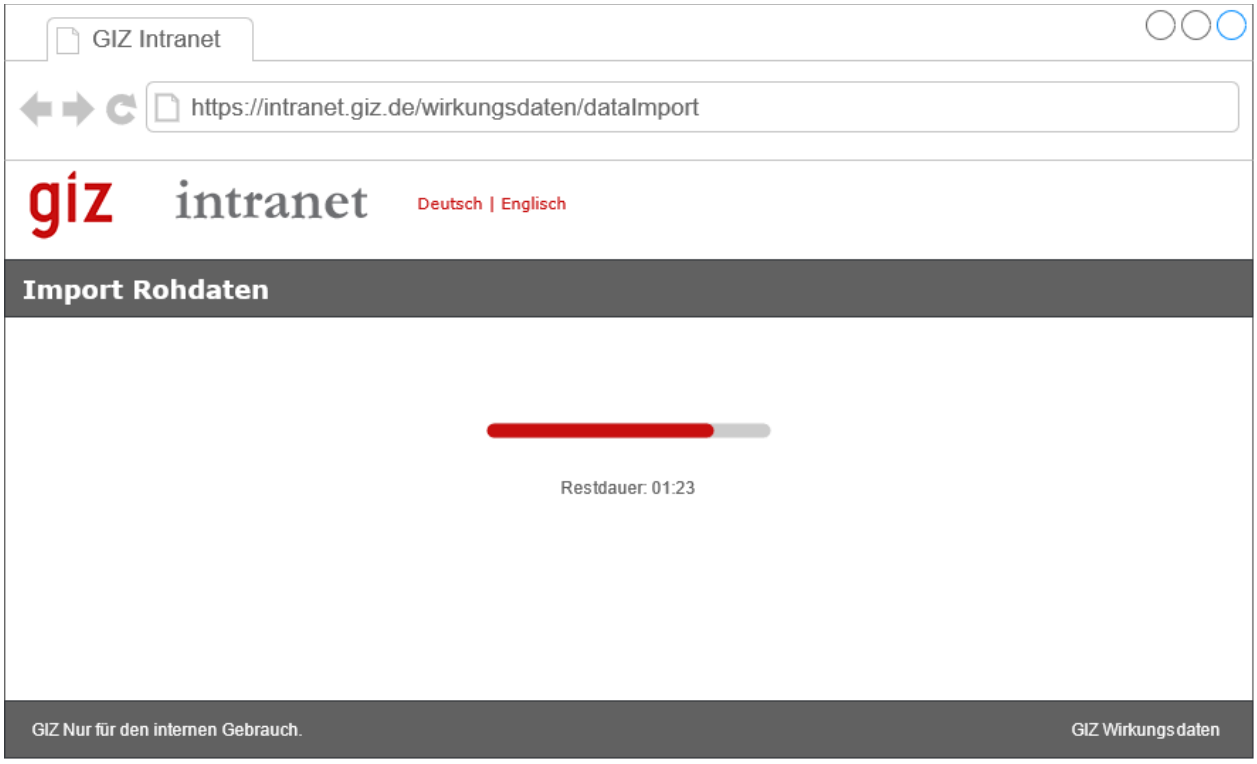


Abbildung 12: UI des Fortschritts-Symbol

A.11 ER Modell

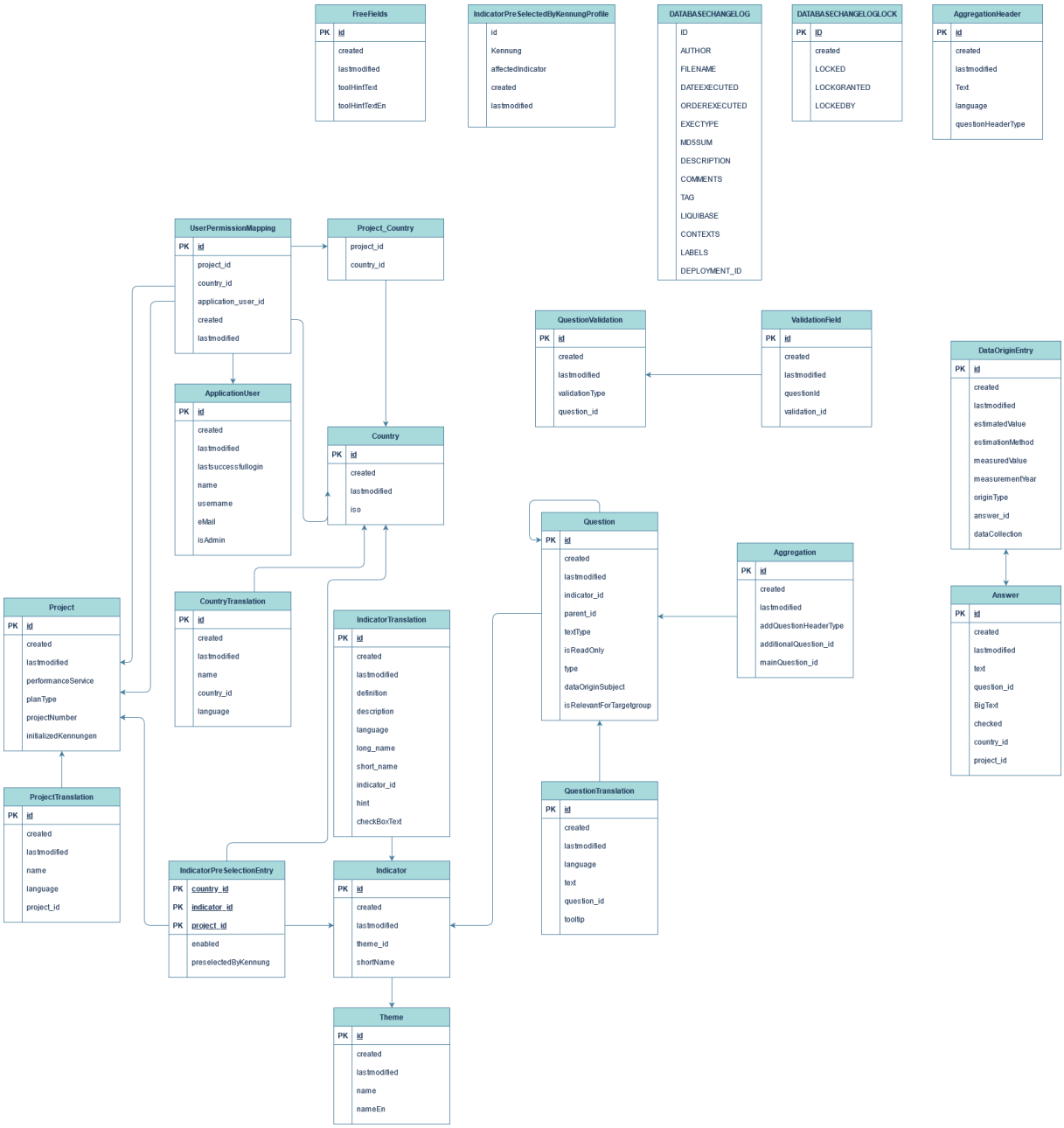


Abbildung 13: Ist-Zustand des ER Modells

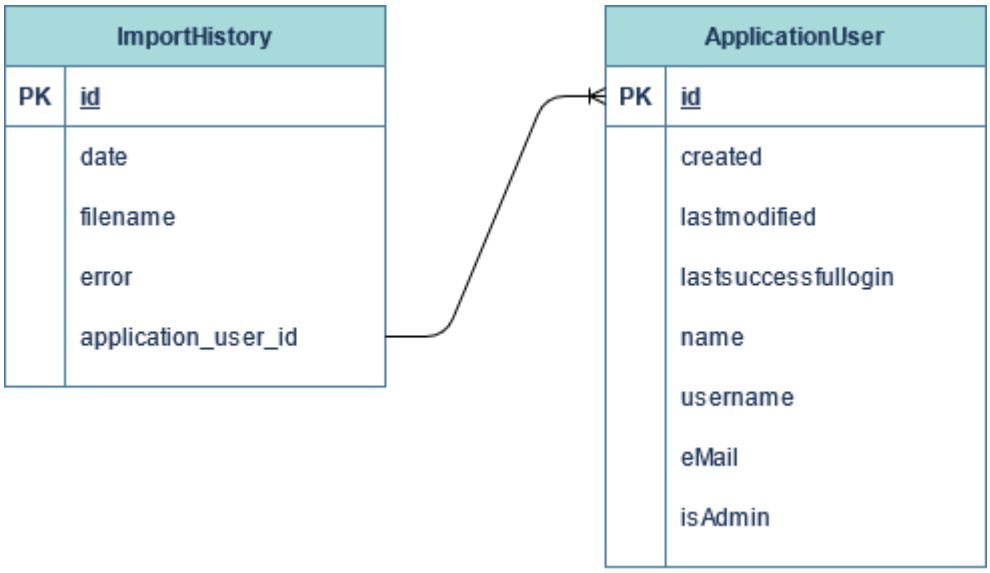


Abbildung 14: Vereinfachtes Soll-Zustand ERM

A.12 Pflichtenheft

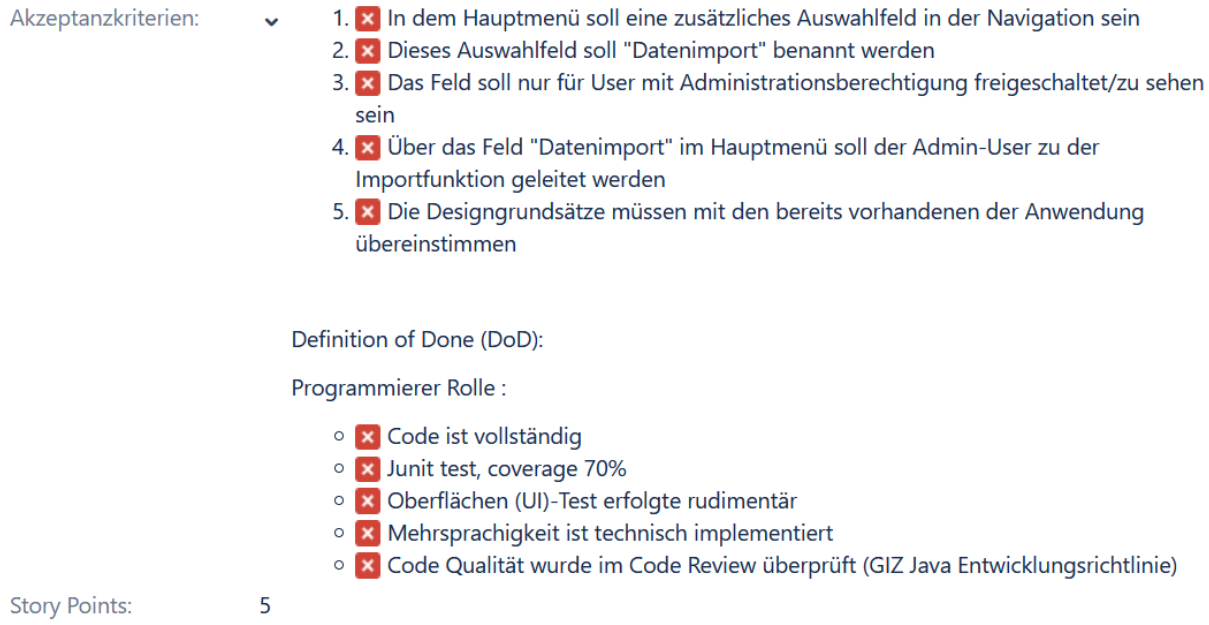


Abbildung 15: Akzeptanzkriterien der User Story - Navigation zur Importfunktion

Akzeptanzkriterien:

1. ✗ Bei Aufruf der Seite dataImport erscheint eine Seite im allgemeinen Layout der Anwendung
2. ✗ Es erscheint eine Überschrift "Datenimport"
3. ✗ Es erscheint ein Element zur Auswahl der Datei vom Gerät des Users
4. ✗ Es erscheint ein Element zum Hochladen der Datei
5. ✗ Bei Ausführung des Importvorgangs wird ein Fortschrittssymbol angezeigt
6. ✗ Bei Erfolg erscheint eine Meldung "Importvorgang wurde erfolgreich durchgeführt"
7. ✗ Bei Misserfolg erscheint eine Meldung "Importvorgang konnte nicht durchgeführt werden" und zusätzlich
   - ✗ wenn Fehler von Syntax (z.B. Delimiter oder Encoding), dann wird "Syntaxfehler" angezeigt
   - ✗ wenn Fehler der Struktur (z.B. Anzahl der Spalten stimmt nicht), dann wird "Strukturfehler" angezeigt
   - ✗ wenn Validierungsfehler, dann wird Anzahl Validierungsfehler und Anzahl der betroffenen Projekte angezeigt
8. ✗ Importvorgang (Historie (Datum, User u.a.) wird in Datenbank gespeichert
9. ✗ Alle durchgeführten Importvorgänge werden tabellarisch angezeigt

Definition of Done (DoD):

Programmierer Rolle :

- ✗ Code ist vollständig
- ✗ Junit test, coverage 70%
- ✗ Oberflächen (UI)-Test erfolgte rudimentär
- ✗ Mehrsprachigkeit ist technisch implementiert
- ✗ Code Qualität wurde im Code Review überprüft (GIZ Java Entwicklungsrichtlinie)

Story Points: 40

Abbildung 16: Akzeptanzkriterien der User Story - Importfunktion für Korrekturen (Oberflächen Wizard)

A.13 CSS-Datei (Ausschnitt)

```css
/* Layout helpers for progessBar
-----------------------------------*/

#myProgress {
  width: 100%;
  background-color: #ddd;
}

#myBar {
  width: 0%;
  height: 30px;
  background-color: #c80f0f;
  text-align: center;
  line-height: 30px;
  color: white;
}

#importTable {
  border-collapse: collapse;
  width: 100%;
}

#importTable td, importTable th {
  border: 1px solid #ddd;
  padding: 8px;
}

#importTable tr:nth-child(even){background-color: #f2f2f2;}

#importTable th {
  padding-top: 12px;
  padding-bottom: 12px;
  text-align: middle;
  background-color: #c80f0f;
  color: white;
}
```

Listings 1: CSS-Dateiausschnitt der Tabellen und des Fortschritts-Symbols

A.14 JSP-Datei (Ausschnitt)

```
<div class="mainFrame" id="accessContent">
              <c:if test="${importStatus == false}">

              form method="POST" enctype="multipart/form-data" action="dataImport">
              File: <input type="file" name="file"/>
              <div class="formButtonFrame" style="margin-bottom: 10px;">
                     <input type="submit" value="Upload" class="btn btnsubmit btnlarge"/>
              </div>
              form>
              </c:if>

              <c:if test="${importStatus}">
                     <h2>Importvorgang wurde gestartet: </h2>
                     <div id="myProgress">
                            <div id="myBar"></div>
                     </div>
                     script type="text/javascript">move(${importFilesize});</script>
              <br>
              </c:if>



              <br>
              <c:if test="${importStatus}">
              <c:if test="${importHistory.size() > 0 }">
                     <table id="importTable">
                            <tr>
                                   <th>User</th>
                                   <th>Dateiname</th>
                                   <th>Datum</th>
                                   <th>Fehler</th>
                            </tr>

                            <c:forEach items="${importHistory}" var="file">
                                   <c:if test="${file != null }">
                                          <tr>
                                                 <td><c:out value="${file.getUser().getUser-
Name()}" /></td>
                                                 <td><c:out value="${file.getFilename()}"
/></td>
                                                 <td><c:out value="${file.getDate()}" /></td>
                                                 <td><c:out value="${file.getError()}"
/></td>
                                          </tr>
                                   </c:if>
                            </c:forEach>
                     </table>
              </c:if>
              </c:if>

              <br>
              ${importLog}

              <br>
              ${importResult}
</div>
```

Listings 2: JSP-Datei dataImport.jsp zur Erstellung der Oberfläche

A.15 Listings des Java Codes

```java
@Entity
public class ImportHistory extends de.giz.model.Entity implements Serializable {

    private static final long serialVersionUID = 1L;

    @Column
    private Date date;

    @Column
    private String filename;

    @Column
    private String error;

    @ManyToOne
    @JoinColumn(name="application_user_id")
    private ApplicationUser user;

    public Date getDate() {
        return date;
    }

    public void setDate(Date date) {
        this.date = date;
    }

    public String getFilename() {
        return filename;
    }

    public void setFilename(String filename) {
        this.filename = filename;
    }

    public String getError() {
        return error;
    }

    public void setError(String error) {
        this.error = error;
    }

    public ApplicationUser getUser() {
        return user;
    }

    public void setUser(ApplicationUser user) {
        this.user = user;
    }
}
```

Listings 3: Klasse ImportHistory.java

```java
import org.springframework.data.jpa.repository.JpaRepository;

import de.giz.model.ImportHistory;

public interface ImportHistoryRepository extends JpaRepository<ImportHistory, Long> {
}
```

Listings 4: Interface ImportHistoryRepository.java

```java
/** A Service for import history */

@Service
public class ImportHistoryService {
	@Autowired
	private ImportHistoryRepository importHistoryRepository;

	/**
	 * Persist import history data
	 * @param currentDate
	 * @param filename
	 * @param error
	 * @param user
	 */
	public void saveImportHistoryEntry(Date currentDate, String filename, String
error, ApplicationUser user) {
	  ImportHistory importHistory = new ImportHistory();
	  importHistory.setCreated(currentDate);
	  importHistory.setFilename(filename);
	  importHistory.setError(error);
	  importHistory.setUser(user);

	  importHistoryRepository.save(importHistory);
	}

	/**
	 * Gets all historys of imports
	 * @return List of ImportHistorys
	 */
	public List<ImportHistory> getListOfImportHistory() {
	    return importHistoryRepository.findAll();
	}
}
```

Listings 5: Klasse ImportHistoryService.java

```
    @Override
    @Async
    public void startAsyncImport(StringBuilder consoleOutput, List<Map<String, String>>
rawData) {
        int errorCount = 0;
        if (rawData != null && !rawData.isEmpty()) {
            rawData.get(0).forEach((k, v) -> consoleOutput.append("map[0]." + k + " = " + v +
"\r\n"));
            consoleOutput.append("\r\n");
            int failedProjects = 0;
            //Building the table header for ImportData
            htmlTableBuilder.addTableHeader("Projekt", "Fehler");

            String projectNumber;

            for (int i = 0; i < rawData.size(); i++) {
                int excelLineNumber = i + 2;
                Map<String, String> element = rawData.get(i);
                ParseResult parseResult = parse(element, excelLineNumber, true);
                if(parseResult.haveValidationErrors()) {
                    String projectNumberForLambaExpression =
Long.toString(parseResult.getProjectNumber());
                    parseResult.getValidationErrors().forEach(err ->
htmlTableBuilder.addRowValues("Simulation failed for project number " +
projectNumberForLambaExpression, "" + err));
                    failedProjects ++;
                    errorCount += parseResult.getValidationErrors().size();
                } else {
                    projectNumber = Long.toString(parseResult.getProjectNumber());
                    htmlTableBuilder.addRowValues("Simulation successful for project number ",
projectNumber);
                    parseResult = parse(element, excelLineNumber, false);

                    // this should not happen but you never know ...
                    if(!parseResult.haveValidationErrors()) {
                        projectNumber = Long.toString(parseResult.getProjectNumber());
                        htmlTableBuilder.addRowValues("Import successful for project number ",
projectNumber);
                    } else {
                        String projectNumberForLambaExpression =
Long.toString(parseResult.getProjectNumber());
                        parseResult.getValidationErrors().forEach(err ->
htmlTableBuilder.addRowValues("Import failed for project number " +
projectNumberForLambaExpression, "" + err));
                    }
                }
            }
            if(errorCount > 0) {
                OUTPUT.append("\r\n\r\nresult: validation failed with " + errorCount + "
errors in " + failedProjects + " failed projects.\r\n");
                ERROR_MESSAGE = "Ja";
            } else {
                OUTPUT.append("\r\n\r\nresult: import succeeded without errors.\r\n");
                ERROR_MESSAGE = "Nein";
            }
        } else {
            OUTPUT.append("Import failed because no project was chosen!\r\n");
            ERROR_MESSAGE = "Kein Projekt gewählt";
        }
        STATUS = false;
    }
```

Listings 6: Auszug aus der Klasse ImportParserServiceImpl.java

```
var i = 0;
function move(filesize) {
  if (i == 0) {
    i = 1;
    var elem = document.getElementById("myBar");
    var width = 1;
    filesize = filesize*30000;
    var id = setInterval(frame, filesize);
    function frame() {
      if (width >= 100) {
        clearInterval(id);
        i = 0;
      } else {
        width++;
        elem.style.width = width + "%";
        elem.innerHTML = width  + "%";
      }
    }
  }

}
```

Listings 7: Datei dataImportPage.js

```
    /**
     * Process the file upload of a CSV-Data
     */
    @PostMapping("/dataImport")
    public String processFileUpload(@RequestParam("file") MultipartFile file, Model
model) throws Exception {
        Role role = permissonService.getContext().getRoleForProjects();
        if (role != Role.ADMIN) {
            return homepageRedirect;
        }
        model.addAttribute("role", role);

        StringBuilder consoleOutput = new StringBuilder();

        importParserService.setStatus(true);

        List<Map<String, String>> rawData = null;
        try {
            rawData = rawImportService.rawImport(new InputStreamReader(
                    file.getInputStream(), DEFAULT_ENCODING_CSV_UPLOAD));
        } catch (IOException e) {
            consoleOutput.append("Errors in line : IOException\r\n");
        } catch (ArrayIndexOutOfBoundsException e) {
            consoleOutput.append("Errors in line :
ArrayIndexOutOfBoundsException\r\n");
        }

        //Zeilen der Rohdaten
        int filesize = rawData.size();

        // Kick of asynchronous lookup
        importParserService.startAsyncImport(consoleOutput, rawData);

        //Save new import in database importHistory
        if (!(importParserService.getStatus())) {
            Date date = new Date();
            String error = importParserService.getErrorMessage();
            String filename = file.getName();
            ApplicationUser user =
applicationUserService.createOrGetCurrentApplicationUser();
            importHistoryService.saveImportHistoryEntry(date, filename, error, user);
        }

        model.addAttribute("importStatus", importParserService.getStatus());
        model.addAttribute("importFilesize", filesize);

        if (!(importParserService.getStatus())) {
            //Info in console for developer
            logger.debug("output=" + consoleOutput.toString());
            logger.debug("result= " + importParserService.getOutput());
            model.addAttribute("importResult", importParserService.getOutput());
        }
        return "dataImport";
    }
```

Listings 8: Klasse ImportController.java

```java
/**
 * ImportHistory Repository Tests
 */
@RunWith(SpringRunner.class)
@WebAppConfiguration
@ContextConfiguration(classes = { WirkungsdatenConfiguration.class })
@Transactional(transactionManager = "transactionManager")
@Rollback
public class ImportHistoryRepositoryTest {

    private static final String TEST_FILENAME =
"C:\\Users\\rhein_tim\\Desktop\\Abschlussarbeit\\Projektantrag -Tim";
    static Date currentDate = new Date();
    private static final Date TEST_DATE = currentDate;
    private static final String TEST_ERROR = "No Error";
    static ApplicationUser user = new ApplicationUser();
    private static final ApplicationUser TEST_USER = user;

    @Autowired
    private ImportHistoryRepository importHistoryRepository;
    private ImportHistory testImportHistory;

    @Before
    public void init() {
        testImportHistory = createImportHistory(TEST_DATE, TEST_FILENAME, TEST_ERROR,
TEST_USER);
    }

    /**
     * Create a importHistory with all params
     */
    private ImportHistory createImportHistory(Date testDate, String testFilename, String
testError, ApplicationUser testUser) {
        ImportHistory testImport = new ImportHistory();
        testImport.setDate(testDate);
        testImport.setFilename(testFilename);
        testImport.setError(testError);
        testImport.setUser(testUser);
        return testImport;
    }

    /**
     * Test to save ImportHistory with all required data
     */
    @Test
    public void testSaveImportHistory() {
        try {
            ImportHistory importHistory = importHistoryRepository.save(testImportHistory);
            assertNotNull(importHistory);
            assertNotNull(importHistory.getId());
            assertNotNull(importHistory.getDate());
            assertNotNull(importHistory.getFilename());
            assertNotNull(importHistory.getError());
            assertNotNull(importHistory.getUser());
        } catch (Exception e) {
            fail("Name can not be null");
        }
    }
}
```

Listings 9: JUnit Test von ImportRepositoryTest.java

A.16 Klassendiagramm (Ausscnitt)

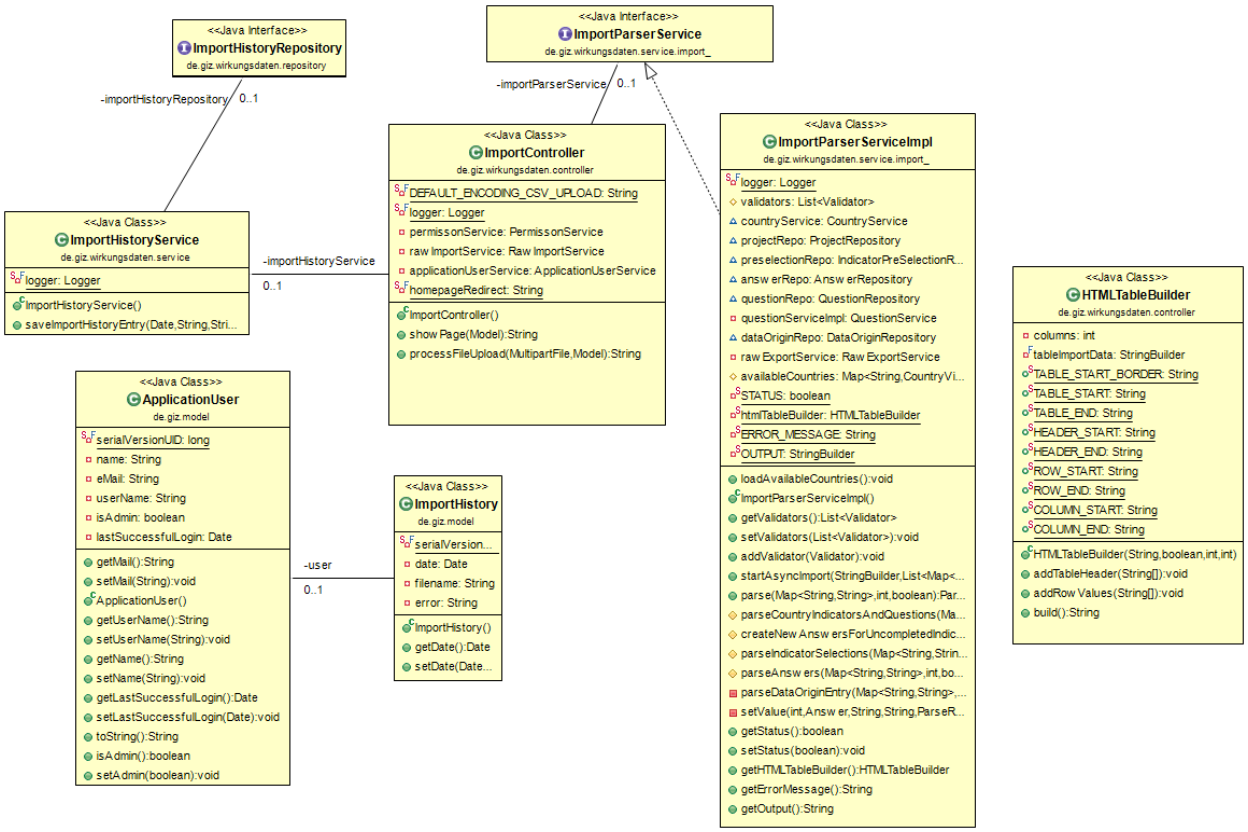


Abbildung 17: Klassendiagramm (Ausschnitt)

A.17 Soll-Ist-Vergleich der groben Zeitplanung

Projektphase	**Soll in Stunden**	**Ist in Stunden**	**Abweichung**
Analyse	6	5	-1
Entwurf	8	7	-1
Implementierung	41	43	+2
Abnahme und Deployment	6	3	-3
Dokumentation	9	12	+3
Gesamt	**70**	**70**	**0**

Tabelle 5: Soll-Ist-Vergleich der groben Zeitplanung