Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Glossar

2 Einleitung

!!!!! labels kontrollieren !!!!! ÜBERARBEITEN
Im Rahmen der Abschlussarbeit für den Ausbildungberuft Fachinformatiker für Anwendungsentwicklung wurde diese Projektdokumentation angefertigt. Sie dokumentiert den Ablauf und die Herangehensweise welche zur Lösung, der im Vorfeld von dem zuständigen Ausbilder definierten, Aufgabe beigetragen haben. Der Ausbildungsbetrieb OktoPOS Solutions GmbH ist ein mittelständiges Unternehmen mit Hauptsitz in Hamburg.

2.1 Projektbeschreibung

ÄNDERUNGEN HIER
Das von der OktoPOS Solutions enwickelte Produkt OkotPOS Cash ist ein, im international Raum, genutztes POS System. Für eine anwenderfreundliche Nutzung ist die gesamte Textausgabe des Front-End in diversen Sprachen konfigurierbar. Zur Realisierung der Textausgabe in den geforderten Sprachen, werden im Quellcode Platzhalter (Tokens) statt konkreter Texte verwendet. Für jede Sprache gibt genau eine Property Datei, in der die Texte der jeweiligen Sprache als Key-Value Paar hinterlegt sind. Die Erweiterung und Wartbarkeit dieser Property Dateien sind nach aktuellem Stand optimierungsbedürftig. Es gibt keine Garantie dafür, dass jede Datei die selbe Anzahl an Tokens beinhalten bzw. es gibt keinen direkten Überblick über den Übersetzungsstand der Dateien. In den meisten Fällen werden die Übersetzungen von unternehmensfremden Personal angefertigt, die mit der Struktur solcher Dateien nicht vertraut sind und daher mehr Zeit in Anspruch nehmen als notwendig.
Der unternehmensinternen Übsersetzungsdienst TranslationService, bietet neben den grundlegenden Funktionen zum Importieren/Exportieren von Tokens und Übersetzung auch ein benutzerfreundlichen Userinterface.

2.2 Projektziel

ÄNDERUNGEN HIER
Ziel des Projektes ist es, durch die Anbindung der Kassensoftware an den unternehmensinternen Translationservice, den Prozess der Übersetzung von dem Releaseprozess zu entkoppeln und Versionsupdates der Übersetzungen während des Livebetriebes zu ermöglichen. Im Rahmen des Projektes soll die Integrität des Kassencodes erhalten bleiben. Deshalb ist es nötig den Updateprozess in eine weitere Anwendung zu überführen. Die Anforderungen der Anwendung sind aus dem Soll-Konzept zu entnehmen. Der Transaltionservice ist zu so zu erweitern, dass Tokens und Übersetzungen in Abhängigkeit ihrer Version an den Translationservice gesendet bzw. geladen werden können. Nach aktuellem Stand, ist der Translationservice nicht in der Lagen die Tokens und Übersetzungen zu Versionieren. Das Anelgen einer neuen Version erfolgt von außen über eine zu schaffende Schnittstelle.Im Front-End des Translationservice, soll die Möglichkeit gegeben werden die Tokens und den Stand der Übersetzung nach ihrer Version anzuzeigen. Da das Erstellen der Schnittstellen, die Implementierung der neuen Anwendung, das Testen der Komponenten usw. bereits die veranschlagten 70 Stunden benötigt, werden das Deployment der Minianwendung sowie die Anpasssung des Deploymentprozesses der Kassesoftware und die Anpassungen am Frontend als Fremdleistung an eine andere Abteilung weitergegeben.

2.3 Projektumfeld

ÄNDERUNGEN HIER
Primär ist das Projekt ein Auftrag der Entwicklungsabteilung des Kassensystems. Das Java gestützte POS System nutzt Übersetzungsdateien um Textstellen im Front-End in verschiedenen Sprachen darzustellen. Während der Entwicklung an der Kasse, speziell beim implementieren von Features, werden in machen Fällen neue Übersetzungstokens eingeführt. Der Entwickler muss dabei den Token in jede einzelne Datei schreiben. Die Übersetzer des Tochterunternehmens OktoCareer nutzen den Translationserivce für die Übersetzungen an dem Personalmanagementsystem OktoCareer. Der TranslationSerivce bietet eine Datenstruktur, welche mit geringen Aufwand auf die neuen Anforderungen angepasst werden kann, und die grundlegende Struktur für die RESTFull API.

2.4 Projektabgrenzung

Die grundlegende Struktur des Translationservice ist bereits implementieren und wird produktiv im Betrieb genutzt. Das Projekt beschränkt sich hinsichtlich der Arbeiten an dem Transaltionservice nur auf das Hinzufügen der neuen Schnittstellen, dem Erweiteren der Datenbank und den daraus resultierenden Änderungen an den vorhandenen Entitäten und Front-End Anpassungen. Durch die beschränkte Dauer die für dieses Projekt zur Verfügung gestellt wurde, hat zur Folge, dass Teile zur Realisierung des Gesamtprojektes an andere Abteilungen ausgelagert werden müssen. Dazu gehört der GitWebhook, welcher die Übersetzungsdateien der Kasse an den Übersetzungsdienst übergibt.

3 Projektplanung

3.1 Projektphasen

Im Vorfeld des Projektes wurden die zu Verfügung stehenden 70 Entwicklerstunden auf verschiende Projektphasen aufgeteilt. Die Zeiteinteilung sowie die einezlene Projektphasen wurden in einer groben Tabelle (Tabelle 1: Grobe Zeiplanung) zusammengefasst. Eine genaue Zeitplanung inklusive der Aufgaben jeder einzelnen Phase können aus dem Anhang 3.1. Detaillierte Zeitplanung entnommen werden. Um die zeitliche Abfolge der einzelnen Projektphase grafisch darzustellen, wurden die einzelnen Aufgaben aus den Projektphasen in sinngemäße Aufgaben zusammengefasst und ein Gantt-Chart überführt Anhang 3.1. Gantt-Chart.

Projektphase	Stunden
Analysephase	7h
Entwurfsphase	11h
Implementierungsphase	35h
Abnahme und Testphase	3h
Dokumentation	14h
Gesamtstunden	70h

Tabelle 1: Grobe Zeitplanung

3.2 Vorgehensmodel

ÄNDERUNGEN HIER

3.3 Resourcenplanung

Das Projekt wurde auf den von dem Ausbildungsbetrieb zur Verfügung gestellten Windows Surface Book und Windwos 10 geplant, bearbeitet und getestet. Dabei wurde als Entwicklungsumgebung für den TransaltionService sowie für die Java App die für den Betrieb linzensierte Software IntelliJ Ultimate von der Firma JetBrains verwendet. Die Grundlage der Daten für den TranslationSerivce sind in einer MySql Datenbank gespeichert. Zur Überprüfung der korrekten Anwendung des ORM Frameworks ”Doctrine” hinsichtlich DDL und DQL, wurden die Ergebnisse über die Software MySQL Workbench mittels SQL Abfragen validiert. Neben den automatisierten Tests der Schnittstellen mit PHPUnit, wurden regelmäßig mit der Software Postman Anfragen an die REST Schnittstellen geschickt. Für die, in Latex erstellte, Dokumentation notwendigen Diagramme wurden über den Onlinedienst Lucidchart angefertigt. Eine genaue Übersicht aller verwendeten Resoucren ist in der Anhang 3.3. Resourcen und Technologien zu finden.

4 Analysephase

4.1 Ist-Analyse

In der nachfolgende Analyse wird das Kassensystem OktoPOS Cash und der Übersetzungsdienst TranslationService in sein ursprünglichen Verfassung beschreiben.

4.1.1 OkotoPOS Cash

Wie in der Projektbeschreibung und dem Projektumfeld beschrieben untertützt das Kassensystem OktoPOS Cash ein multilinguales Front-End. Um das zu gewährleisten, werden im Quellcode der Kasse s.g. Übersetzungstokens verwendent. Für jede unterstützte Sprache, ist eine Übersetzungs-datei im Porpertyformat in der Ressourcen der Kasse hinterlegt. Jede Datei beinhaltet das gleiche Set an Tokens mit der jeweiligen Übersetzung als Key-Value Paar. Bei der Einführung eines neue Tokens kopiert der Entwickler den neuen Token in eine der Dateien mit einer sinngemäßen Übersetzung. Da es im Unternehmen gängig ist, die Tokens nur in eine bzw. maximal zwei Datein zu überführen, wird die Aufgabe der Wartung und Pflege an die Übersetzungsabteilung deligiert. Neben der trivialen und zeitaufwändigen Arbeit die einzelenen Dateien auf den gleich Stand zu halten kann es, auf Grund von diversen Faktoren, passieren das der Stand der Dateien divergiert.
Bisher gibt es keine konkrete Oberfläche um Übersetzungen für die einzelnen Tokens anzufertigen bzw. auch keine direkte Refferenz auf die Textstelle auf die der Token zeigen soll. Der Übersetzer hat die Aufgabe aus geeigneten Quellen (Ticketsystem, andere Übersetzungsdatei, Entwickler fragen) sich eine exemplarische Übersetzung zu beschaffen. Es gibt allerdings auch Fälle bei denen bisher keine Übersetzung für einen Token erstellt wurde. Dann hat der Übersetzer die Aufgabe, über das Lesen von Quellcode die richtige Textstelle im Front-End zu finden.
Im aktuellen Workflow werden die Übersetzungsdatein im Betazustand einer neuen Minorversion an die Abteilung für Übersetzung übergeben. Nach Fertigstellung der Übersetzung werden überarbeiteten Dateien an Projectowner der Kassesoftware übergeben, welcher er dann mit den alten Übersetzungsdateien zusammenführt. Dadurch kann nur noch indirekt über einen Releasebranch auf die verwendeten Übersetzungen geschlossen werden und nicht über einen Versionsstand der Übersetzungsdateien. Außerdem hat der workflow die Folge, dass Änderungen an den Übersetzungsdateien (Rechtschreibfehler) ein Versionsupdate der Kasse mit sich bringt.

4.1.2 TranslationService

Für die Übersetzungen in einem anderen System des Unternehmens wurde der webbasierte Transaltionservice entwicklet. Dieser hat die Aufgabe über Schnittstellen, Tokens zu importieren. In einer benutzerfreundlicher Übersicht hat der Übersetzer die Möglichkeit sich nicht übersetzte Tokens für seine Sprache anzeigen zu lassen. Neben der Übersicht hat jeder Token, falls vorhanden, eine exemplarische Üebrsetzung für jede Sprache in der Token bereits übersetzt wurde. Der Translationservice biete verschiedene Möglichkeiten die Informationen für Tokens anzeigen zu lassen. Bisher ist es allerdings nicht möglich eine Übersetzung zu versionieren. Das bedeutet, das die Änderungen an den Übersetzungen final sind und nicht mehr auf alte Übersetzungen zurückgeschlossen werden kann. Die Daten des Übersetzungsdienst sind in einer MySql Datenbank gespeichert. In der Abbildung 4.1. Datenbankstruktur Urfassung ist eine grobe Übersicht über das Datenbankmodel aufgezeigt. Das vollständige ERD ist im Anhang 4.1. Datenbankstruktur Urfassung zu finden.

4.2 Soll-Analyse

Druch die Anbindung des Kassensystems an den Transaltionservice soll die Pflege und Wartung nahezu automatisiert werden. Die Übesetzungen für die einzelnen Tokens sollen über eine neue Schnittstelle am Translationservice in Abhängigkeit zu ihrer Version geladen werden und über eine Prozess in die verwendete Version des Kasse eingespielt werden. Änderungen an Übersetzungen auf Ebene des Translationserivce werden dann durch den Neustart der Kasse übernommen. Dadurch soll der Übersetzungsprozess vollständig von dem Deployment der Kasse entkoppelt werden. Damit die Übersetzungen in Abhängigkeit ihrer Version erstellt bzw. geladen werden können, ist es notwendig den Translationservice dahin

gehend zu erweitern das Tokens und Übersetzungen versioniert werden können. Die dafür notwendigen Änderungen sind auf Datenbankebene sowie Schnittsellen- und Fron-End-Ebene auszuführen.

4.3 Wirtschaftsanalyse

4.3.1 Monetare Vorteile

Nach Abschluss des Projektes werden die Übersetzungsdateien durch den Übersetzungsdienst generiert, dadruch entfällt die zeitaufwändigen Arbeit der Wartung und Pflege. Durch die benutzerfreundliche Oberfläche des Übersetzungsdienst müssen externe Übersetzer nicht mehr in die Struktur und Funktionsweise der Übersetzungsdateien eingebarietet werden. Die dadruch entstandene Zeitersparniss kann als zusätzlicher Gewinn für das Unternehmen gerechnet werden.

4.3.2 Nicht-monetare Vorteile

Als Softwarehersteller ist der Kundensupport und die Kundenzufriedenheit ein wichtiger Bestandteil der wettbewerbsfähigkeit. Auch wenn mit dem Liveupdate der Übersetzungen im Kassensystem und die Versionierung der Tokens und Übersetzungen im TranslationService kann keine konkreten Gewinne berechnen lassen, ergeben sich dennoch Voreile, die sich indirekt positiv auf den Unternehmensgwinn auswirken.

4.3.3 Kostenaufstellung

MATHE HIER

4.3.4 Aromatisierungsdauer

Das Prohjekt beinhaltet nicht-monetare Vorteile (siehte Nicht-monetare Vorteile). Deswegen hat der Autor eine Aromatisierungsrechnung ohne den Faktor der N

4.4 Anwendungsfälle

Für das Projekt wurden unterschiedliche Stakeholder bestimmt. Da es sich bei dem Projekt um ein überwiedend internes Feature handelt, wird der Stakeholder ”Kunde” nicht mit einbezogen.

4.5 Komponenten

Im Projekt werden zwei bereits bestehende Komponenten durch ein weitere Komponente verbunden. Im Anhang 4.5. Komponentendiagramm wird veranschaulicht wie diese Komponenten nach Fertigstellung des Projektes miteinander kommunizieren werden. Für die Kommunikation der beiden Hauptsysteme, OktoPOS Cash und dem Translationservice, werden Hilfkomponentnen verwendent. Die neu zu entwickelnde Komponente Translationupdater hat die Aufgabe, die Übersetzungen für Tokens in Abhängigkeit von Sprache und Version von dem TranslationSerivce zu laden, in das Kassenlesbare Format zu konvertieren und die bestehenden Übersetzungen zu updaten. Das bestehende Versionverwaltungssystem besitzt die Möglichkeit, WebHooks einzurichten. Der Zweck eines WebHooks kann optimal als genutzt werden, um die Änderungen an den Übersetzungsdateien der Kasse an den Translationserivce zu übertragen. Der Translationservice bietet bisher schon die Möglichkeit zum Importieren und Exportieren von Tokens und Übersetzungen, allerdings nicht im gefordertetn Maße. Daher stellt der TranslationSerivce neue Schnittstellen bereit mit denen die Hilfskomponenten kommunizieren können. Für eine bessere Übersicht wurden die Anpassungen an den bestehenden System farblich im Komponentendiagramm (Anhang 4.5. Komponentendiagramm) hervorgehoben.

4.6 Lastenheft

FOLGT IRGENDWANN

5 Entwurfsphase

5.1 Zielplattform

Ziel des Projektes ist, wie im Punkt Projektbeschreibung beschrieben, die Kommunikation zwischen zwei bestehenden System zu ermöglichen. Die Integrität des Kassencodes soll nach Angaben des Auftraggebers erhalten bleiben. Dafür ist es notwendig ein Modul zu implementieren, welches die nötigen Schritte zum Aktualisieren der Übersetzungsdateien ermöglicht.
Nach Angaben des Auftraggebers soll die Möglichkeit bestehen, zu einem späteren Zeitpunkt das Modul in den Kassencode zu integrieren. Daraus entsteht Vorgabe, das Modul auf dem gleichen Languagelevel des Kassencodes zu programmieren.
Zum aktuellen Zeitpunkt wird die Kasse über den Buildprozess Ant deployed. Basierend auf der Anforderung des Auftraggebers (LASTENHEFT PUNKT XX), wird in naher Zukunft der Buildprozess auf Gradle umgestellt. Für eine erleichterte Integration des neuen Modules, soll das neue Projekt bereits den Gradle Buildprozess unterstützen.
Der Translationservice ist in der Programmiersprache PHP mit dem Languagelevel 7.3 erstellt. Es wurde bei der Entwicklung des Translationservice darauf geachtete, dass der Serivce auf allen gängigen Webbrowser ”FOOTNODE” funktioniert. Daher wurde für die Darstellung der Daten HTML 5.2 verwendet. Für eine saubere und ansprechende Darstellung der Daten wurde das Stylesheet von Twitter Bootstrap verwendet.
Da es sich bei den vorgegeben System um produktive Systeme handelt, die bereits aktiv genutzt werden, sind die Programmierparadigmen und Programmiersprachen vorgegeben. Das Aufstellen einer Nutzwertanalyse ist daher obsolet.

5.2 Architektur

Der Transaltionserivce basiert auf dem Mode-View-Controller (MVC)-Architekturmuster. Demnach werden die zugrunde liegenden Daten in einer Datenbank gespeichert, während die View für die Darstellung der Datenverantwortlich ist. Über die Controller bildet eine bidirketionale Kommunikationschicht um Daten zu manipulieren bzw. Daten anzuzeigen. Durch die Entkopplung, welche durch das MVC Pattern erreicht wird, wird die Erweiter- und Wartbarkeit signifikant erleichtert.
Die Darstellung der Daten wird im Translationservice durch die Template-Engine Twig realisiert. Dabei werden die Daten über Action Klassen der Engine zu verfügung gestellt, welche sie dann an die, als variabel definierten, Felder übergibt. Die Action Klassen representieren, im MVC Design Pattern, die Controller. Sie werden üblicherweise über vordefinierte Routen aufgerufen und verarbeitetn die Daten auf Grundlage der gesendeten Requests. In webbasierten Anwendungen werden üblicher weise die genutzten Daten in einer Datenbank gespeichert. Die Modelle der Daten sind als Entitäten in einer Datenbank definiert. Das Object-Relation Mapping (ORM)-Framework automatisiert dabei die Aufgabe, aus definierten Objekten die korrekte DDL zu genieren und auf der Datenbank auszuführen. Die aktuell verwendeten Schnittstellen des Translationserivce sind bereits nach der RESTful Architektur designed. RESTful steht für eine zustandslose Kommunikation zwischen Client und Server. Durch Cachingfähige Daten können Client-Server-Interaktionen optimiert werden. Die neuen Schnittstellen, sollen ebenfalls nach der RESTful Architektur implementiert werden. Der Translationupdater ist ein Modul welches das Kassensystem um eine Subroutine erweitern soll. Die Subroutine beschafft Daten von einem autarkem System, verarbeitet diese und injeziert die verarbeiten Daten in ein bestehendes System. Für eine saubere Implementierung wird eine Variation des Pipes und Filter Architekturmustern angewendet. Dabei wird der Filterteil des Architekturmusters weggelassen.

5.3 Benutzeroberfläche

Änderungen an der Benutzeroberfläche des Translationserivce werden über die Template-Engine Twig durchgeführt. Grundsätzlich wird die Oberfläche durch die Auszeichnungssprache HTML beschrieben. Twig bietet die Möglichkeit HTML Formulare durch algorithmische Grundstrukturen zu erweiteren. Dabei ist es möglich Iterationen und bedingte Anweisungen direkt im HTML Formular einzubinden.

5.4 Datengrundlage

Wie in der Architektur beschrieben, speichert der Translationservice seine Daten in einer MySql Datenbank. Um eine organisierte Übersicht über die bestehenden Daten wurde da Datenbankschema in ein ERD übertragen. Datengrundlage zeigt einen groben Überblick über das bisherige Datenbankschema des Translationservice.

Abbildung 1: Ursprüngliches ERD ohne Attribute

Eine Detaillierte Übersicht mit Attributen ist in der Abbildung 5.4. Datenbankstruktur Urfassung zu finden. Die REST Schnittstellen im Translaionservice verwenden das für REST typische Format Json um Daten mit Clients auszutauschen. Die neu zu entwicklenden Schnittstellen sollen sich der bestehenden Struktur anpassen und das selbe Format für Requests und Responses verwenden. Als Datengrundlage für die Übersetzungsdateien nutzt das Kassensystem Dateien im Propertyformat. Der Translationupdater hat somit auch die Aufgabe, die empfangenen Daten in das korrekte Format zu konvertieren.

5.5 RESTFul APIs

Im Punkt TranslationService wurde bereits beschrieben, dass die Schnittstellen nach der REST Architektur desigend wurden. Neue Schnittstellen werden ebenfalls nach der RESTful Architektur desigend.
Im Gegensatz zu den ursprünglichen undokumentierten Schnittstellen, hat der Auftraggeber darum gebten die neuen Schnittstellen zu dokumentieren.
Da der Transaltionservice nach vollendung dieses Projektes von mehreren Abteilungen genutzt wird ist eine vollständige Dokumentation der neuen Schnittstellen notwendig. Als Dokumentationswerkzeug für RESTful APIs wird in diesem Projekt Swagger verwendet. Ein Ausschnitt der, im Planungsmeeting entstanden, Schnittstellen ist im Anhang 5.5. Schnittstelledokumentation zu finden. Die neuen Schnittstellen wurden in einem Planungsmeeting mit zuständigen Projectowner des Translationsercive geplant.

5.6 Testcases

Das gesamte Projekt wurde nach dem Prinzip Test-driven entwickelt. Das bedeutet, dass vor der Implementierung Tests geschrieben werden. Die Tests werden auf Grund der fehlenden Funktionalität zu beginn fehlschlagen. Test-Driven hat den Vorteil, dass der Entwickler während der Implementierung von Funktionalitäten ein direktes Feedback bekommt, ob das gewünschte Ergebnis erreicht wurde bzw. diverse Feherfälle abgedeckt wurden.
Damit Tests die Datenbankzugriffe beinhalten, nicht beeinflusst werden können, werden entweder Mockups für Datenbanken erstellt oder es wird für den Test auf einer leeren Testdatenbank ausgeführt. Um eine leere Testdatenbank zu garantieren, werden vor und nach jedem Test die gesamte Datenbank bereinigt. Für den Test werden ausschließlich im Test definierte Daten verwendet. Eine Liste von getesteten Funktionen ist im Anhang 5.6. Testcases Translationservice und Anhang 5.6. Testcases Translationservice zu finden. Die Tests für den Translationsercive werden mit dem PHP Framework PHPUnit erstellt und regelmäßig ausgeführt. Der Translationupdater nutzt das Testframework JUnit und führt seine Tests über das Gradle Buildscript während des Buildprozesses aus.

5.7 Pflichtenheft

ANHANG

6 Implementierungsphase

6.1 Iterationsphase

Für eine strukturierte Implementierung, wurde ein Iterationsplan erstellt. Dieser gibt dem Entwickler eine Überblick über die erfoderlichen Schritte zur Fertiggestellung der neuen Software. Der Itarationplan steht im Anhang 6.1. Iternationsplan zur Verfügung. Das Erstellen der Tests ist kein Teil der Implementierungsphase sondern ist noch Teil des Entwurfes.

6.2 Translationservice

6.2.1 Erweiterung der bestehenden Datenstruktur

Als Grundlage für die weiterführende Programmierung wurde zu Beginn der Implementierung mit der Erweitung der Datenstruktur begonnen. Um die geforderte Versionierung im Translationservice abbilden zu können, musste eine neue Entität in das bestehende Datenbankschema eingebunden werden. Das ORM Mapping Tool Doctrine kann über Annotations Klassen in ein Datenbankschema überführen.
Dabei werden für Klassen, die in der Datenbank eine Entität representieren, mit der Annotation

```
@ORM\Table (name="version")
```

Listing 1: Annotation für eine Entität

wird die Klasse Version.php für Doctrine als Entität markiert.
Für ein Attribut einer Tabelle wird ein Attribut der PHP Klasse mit der Annotation

```
@ORM\Column (type="string", nullable=false)
```

Listing 2: Annotation für ein Attribut

für Doctrine gekennzeichnet. Dabei können neben dem Datentyp auch weitere Paramter wie zum Beispiel `nullable=false` gesetzt werden. `nullable` sagt aus, das der Wert in der Spalte nicht null sein darf.
Aus dem finalen ERD (A.3.2) sind neben der neuen Basisentität noch Relationstabellen Token_Version und Translation_Version zu erkennen. Relationstabellen werden immer dann von Doctrine generiert, wenn zwei Tabellen in einer Many-To-Many Beziehung stehen. Relationstabellen werden verwendet um das Datenbankschema in der dirtten Normalform zu ”bringen”. In der objeketorientierten Porgrammierung werden Many-To-Many Beziehungen über Listen abgeblidet. Dabei haben die beiden Objekte eine Liste von Instanzen des jeweiligen anderen Objektes als Attribut.
Damit das verwendente ORM Tool die Relationstabellen für die betroffenen Tabellen erstellen kann, werden weitere Annotations benötigt. Listing B.1 zeigt an dem Beispiel Token_Version die Umsetzung einer M:N Beziehung über Doctrine Annotations.

6.2.2 Erweitern der Schnittstellen

Anhand der Schnittstellendefinition A.6 wurden die neuen Schnittstellen im Translationservice implementiert. Der Translationservice wurde auf Basis des Slim Frameworks aufgebaut. Über das PHP-File `Bootstrap.php` werden die Routen der Klasse `TranslationServiceAppBuilder.php` werden die Routen in der `TranslationServiceApp` registriert. Neue Routen werden anhand eine simplen Schemas als Route für die App definiert. Die Definition der Routen innerhalb des Slim Frameworks ist wie folgt aufgebaut:

```
$app->post('/api/v1/{foo}/list',Foo::class)
```

Listing 3: Beispielroute

Die Tabelle 2 erklärt die Bestandteile der Routenregistrierung aus 3.

Befehl	Erläuterung
$app	Wrapper Klasse um die verwendeten Container der Applikation.
post	Definition des HTTP Request für die Route. (GET, POST, PUT usw.)
´/api/{foo}/list´	Definition der Route. Variable Parameter werden in {} dargestellt.
Foo::class	Die genutzte Action Klasse die aufgerufen werden soll.

Tabelle 2: Übersicht Routenregistrierung in Slim

Im Listing B.2 wird die Umsetzung einer Routenregistrierung anhand von zwei, im Rahmen des Projektes erstellten, Routen deutlich gemacht.
Für die neuen Routen wurden dementsprechen neue Actionklassen entwickelt die die jeweiligen Anforderung an die Routen erfüllt. Über das verwendente PHP Framework

```
"php-di/slim-bridge": "^1.1.2"
```

können Abhängigkeiten in Container abgelegt werden und per Dependency Injection in den verschiedenen Klassen genutzt werden. In den meisten Fällen haben die Actionklassen Abhängigkeiten zu den verschiedenen Repositoires. Die Abhängigkeiten werden in den Konstruktoren der Actionklasse definiert. Durch die Dependency Injection werden bei Aufruf die richtigen Abhängigkeit geladen. Dependency Injection hat den weiteren Vorteil, dass zur Laufzeit auf der gleichen Instanz eines Objektes gearbeitet wird.
Bei dem Aufruf einer Klasse wird die Funktion `__invoke` mit den Parameter `Slim/Request` und `Slim/Response` aufgerufen. Innerhalb der `__invoke` wird die eigentlich Logik der Actionklasse ausgeführt. Über den Parameter `$request` ist eine Refferenz auf das mitgesendete Requestobjekt. Neben den grundlegenden Informationen eines HTTP-Requests kann auch weitere Routenparameter (siehe Beispielroute Listing 3 {foo}) und einen Requestbody zugegriffen werden. Der Requestbody muss generell dann gesetzt werden, wenn es sich um einen POST-Request handelt.
Der Parameter `Response` ist eine Refferenz auf das Responseobjekt des Routenaufrufes. Dieser wird so manipuliert, dass die Applikation die korrekten Informationen an den aufrufenden Klienten zurückgibt. Für den Response stehen mehrere Möglichkeiten zur Verfügung um die Informationen in ein eindeutig lesbares und serealisierbares Ergebnis zu ”verpacken”. Basierend auf der RESTful-Architektur werden Informationen bezüglich des Responses in einem Json an den Klienten zurückgegeben. Neben der Information ob die angeforderte Operation erfolgreich (success) werden auch eine Message und optional auch noch weitere Daten an den Klienten zurückgegeben. Da es sich um eine Refferenz handelt, muss das Responseobjekt nicht neu erzeugt werden, es werden neue Attribute hinzugefügt oder geändert.
Der Ablauf von `__invoke` Funktionen der Actionklasse wird in dem Kapitel Geschäftslogik 6.3 erklärt. Während des Projektes wurde alle im Anhang A.6 definierten Routen nach dem in diesem Kapitel beschriebenen Prinzip implementiert.

6.3 Geschäftslogik

In dem Kapitel 6.2.2 Erweiterung der Schnittstellen wurde beschrieben, wie die einzelnen Routen mit dem Slim-Framework angelegt und mit einer Geschäftslogik verknüpft werden. Diese Kapitel beschreibt den Umgang mit den mitgesendeten Daten und der generellen Routine die der Bearbeiter nutzt um vorhersehbare Fehler abzufangen und das geforderte Ergbniss erreicht.
Wie in Kapitel 6.2.2 beschrieben, hat jede Route genau eine Actionklasse, alternativ auch Controller gennant, die wiederrum genau eine `__invoke`-Funktion besitzt. Listing B.3 und Listing B.4 zeigen zwei unterschiedlich Implementierungen der `__invoke`-Funktionen aus dem Kapitel 6.2.2 die sich nach den Vorgaben der Schnittstellendokumentation A.6 und Testvorgaben aus dem Kapitel A.8 verhalten. Eine `__invoke`-Funktion beginnt damit die mitgesendeten Daten und Routenparameter in geeigneten variablen zu überführen. Um zu verhindern, dass mit fehlenden, falschen oder kaputten Daten gearbeitet wird, werden vor zuerst die, für die Schnittstelle definierten, Fehlerfälle abgefangen und mit dem, in der Schnittstellendokumentation A.6 definierten, Response an die Klienten zurückgegeben.
Nach Ausschluss aller möglichen Fehlerfälle, können die validen Daten verarbeitet werden. Die Erweiterung des Translationservice beinhaltet drei Arten von Schnittstellen - Hinzufügen, Ändern und Abfragen von Daten.

Parameter	Erläuterung
baseUrl	Bestimmt die URL über die der Translationserivce erreichbar ist
languages	Eine Liste von Sprachen die aktualisiert werden sollen
verseion	Gibt die Version der Übersetzungen an

Tabelle 3: Konfigurationsparameter

6.3.1 Anpassen der Benutzeroberfläche

In Kapitel 5.3 wird der grundlegende Aufbau der Benutzeroberfläche des Translationservice erläutert. Die Anforderungen des Auftraggebers hinsichtlich der Erweiterungen der Benutzeroberfläche beschränken sich auf einen weiteren Filter, der die Translation und Tokenübersicht hinsichtlich ihrer Version sortiert. Dabei hat sich der Projektbearbeiter an den bestehende Style der Benutzeroberfläche gehalten. Im Anhang A.12 ist ein Vorher-Nachher-Vergleich zu sehen. Für den neuen Filter wurde ein neuer Dropdown-Button an das bestehende Filtermenu angefügt. Das Dropdown besteht aus den einzelenen Versionen die bereits im System angelegt wurden. Die Daten werden dem Front-End über die Twig-Engine bereitgestellt. Der Klick auf ein Dropdown-Item löst die

6.4 Translationupdater

Der zweite Tei des Projektes ist der Translationupdater. Der Translationupdater ist ein Dienst zum aktualisieren der Übersetzungsdateien der Kasse. Die Routine für den Translationupdater besteht aus drei Teilen - Beschaffen, verarbeiten, injezieren. Da es sich um einen Dienst für das Kassensystem handelt, sollen die in der Tabelle 3 aufgezählten Teile des Dienstes konfigurierbar sein. Die Konfiguration des Translationupdater wird über die Klasse `SystemConfig.java` abgeblidet. Diese lädt die einzelen Properties aus der `configuration.ini` (siehe Listing B.5) und überführt die Werte der Parameter in Attribute der Klasse. Über eine Instanz der `SystemConfig.java` können via `Getter`-Methoden auf die einzelnen Werte zugegriffen werden. Da es sich bei der Konfiguration um eine eindeutige, zur Laufzeit unverändere Klasse handelt, wurde die `SystemConfig.java` als Singelton-Pattern implementiert. Die Singeltonimplementation stellt sicher, dass von einer Klasse genau ein Objekt erzeugt wird. Da eine Konfiguration normalerweise an verschiedene Stellen im Code verwendent wird, kann auch der Vorteil der globale Erreichbarkeit eines Singelton genutzt werden. Listing B.6 zeigt die notwendigen Elemente damit es sich bei der `SystemConfig.java` um eine Singleton-Implementierung handelt.
Als weitere Utility wurde das `Printer.java` Interface geschaffen. In der Tabelle 4 werden die bereitgestellten Methoden beschrieben. Die unterschiedlichen Methoden sollen die Möglichkeit bieten, die Dringlichkeit der Message zu definieren.
Eine konkrete Implementierung des Interfaces `Printer.java` ist die Klasse `ConsolePrinter.java`. Der

Signatur	Beschreibung
printProgressMessage(int progress, int max, String toWrite)	Gibt den aktuellen Progress inklusive einer Nachricht aus
print(String message)	Gibt Nachricht aus.
printError(String error)	Gibt einen Error aus.
printWarning(String)	Gibt eine Warnung aus.

Tabelle 4: Printer Interface

`ConsolePriner` soll die Nachrichten über die Kommandozeile an den Nutzer weitergeben. Durch das Interface `Printer.java` besteht die Möglichkeiten zu einem anderen Zeitpunkt weitere Ausgabeformen zuzulassen.

Der erste Schritt der eigentlichen Routine, ist die Beschaffung der geforderten Daten. Der Translationservice bietet dafür die neue Schnittstelle (siehe Listing 4)

```
'/api/v1/version/{version}/translations/{language}
```

Listing 4: Export Translations

Die Schnittstelle erwartet die Parameter `version` und `language`. Für den Parameter `version` ist der Wert aus der `SystemConfig.java` zu verwenden. Die Konfiguration ermöglicht es, Übersetzungsdateien in mehreren innerhalb eines Programmaufrufes zu aktualisieren. Das Nassi-Shneiderman-Diagramm im Anhang A.11.1 beschreibt die Ablauf zur Beschaffung der Daten in Abhängigkeit der Version und der verwendenten Sprachen. Der notwendige HTTP-GET-Request wird über die externe Java Library `org.apache.httpcomponents` erstellt und ausgeführt. Der Methodenaufruf `CloseableHttpClient::execute` gibt ein Responseobjekt zurück, mit dem die Antwort des Translationservice ausgelesen werden kann. Die Schnittstellendokumentation A.6 gibt an, beschreibt den Response bei einer erfolgreichen Anfrage. Mit der externen Library `com.google.gson` werden die Arrayelmente des json in ein Array mit dem Elmenttyp `TransferObject` überführt. Zur weitern Verwendung der Daten, wird das Array in eine ArryList umgewandelt und mit dem Sprachenkürzel als Key in einer HashMap hinterlegt.
In der nächsten Instanz der Routine, werden die bereitgestellten Daten in das property-schema konvertiert und temporär als Datei im Dateisystem abgelegt. Dabei fällt die Informationen über die Sprache aus dem TransferObject weg. In der letzten Instanz werden die neuen Übersetzungsdatein in die .jar der Kassensoftware injeziert. Der Schematische Aufbau der Kassensoftware unterliegt laut der Betriebsordnung der OktoPOS Solutions GmbH der Geheimhaltung. Daher ist es dem Autor dieses Dokumentation nicht gestattet konkrete Informationen über den Zielpfad der Injektion anzugeben. Das Updaten eine .jar-Datei ist innerhalb einer Java-Anwendung nur über externe Hilfsmittel möglich. Der `CashDeskInjector.java` sucht innerhalb seines Dateiverzeichnis nach der konkreten .jar der Kassensoftware. Wird diese nicht gefunden, wird eine Warnung an den Benutzer ausgegeben und das Programm beendet. In dem anderen Fall wird ein CMD-Befehl generiert, der die Übersetzungsdatei im Klassenpfad der Kasse durch die neue Übersetzungsdatei ersetzt. Der CMD-Befehl wird über `Runtime.getRuntime().ex` ausgeführt. Das Nassi-Shneiderman-Diagramm A.11.2beschreibt den genauen Aufbau der Injection in einer allgemeingültigen Notation. Die erfolgreiche Injektion wird eine Meldung an den Nutzer weitergegeben und das Programm beendet seine Routine.

7 Abnahme- und Einführungsphase

7.1 Abnahme durch den Fachbereich

Die Änderungen am Translationserivce wurden nach Fertigstellung dem zuständigen Projekteigentümer vorgelegt. Da die Anforderunen und Lösungsansetze in den Meetings regelmäßig besprochen wurden, ist der Großteil er Änderungen bereits bekannt gewesen. Dadurch konnte die Abnahme innerhalb eines kurzen Zeitraumes durchgeführt werden. Der neu entwicklete Dienst Translaionupdater wurde durch den Projekteingentümer der Kassensoftware abgenommen. Dabei wurden Verbesserungsvorschläge bezüglich des Iterativen Ablaufes der Routine angemerkt und für den Ausblick 8.2 notiert.

7.2 Einführung

7.3 Deployment

8 Retroanalyse

8.1 IST-SOLL-Vergleich

Im Rahmen des Projektes wurde 2 von 3 Kernzielen erreicht.

8.2 Ausblick

8.3 Lesson learnd

A Anhang

A.1 Detaillierte Zeitplanung

A.2 Gantt-Chart

Gantt Chart will follow!

A.3 Datenbankmodelle

A.3.1 Datenbankstruktur Urfassung

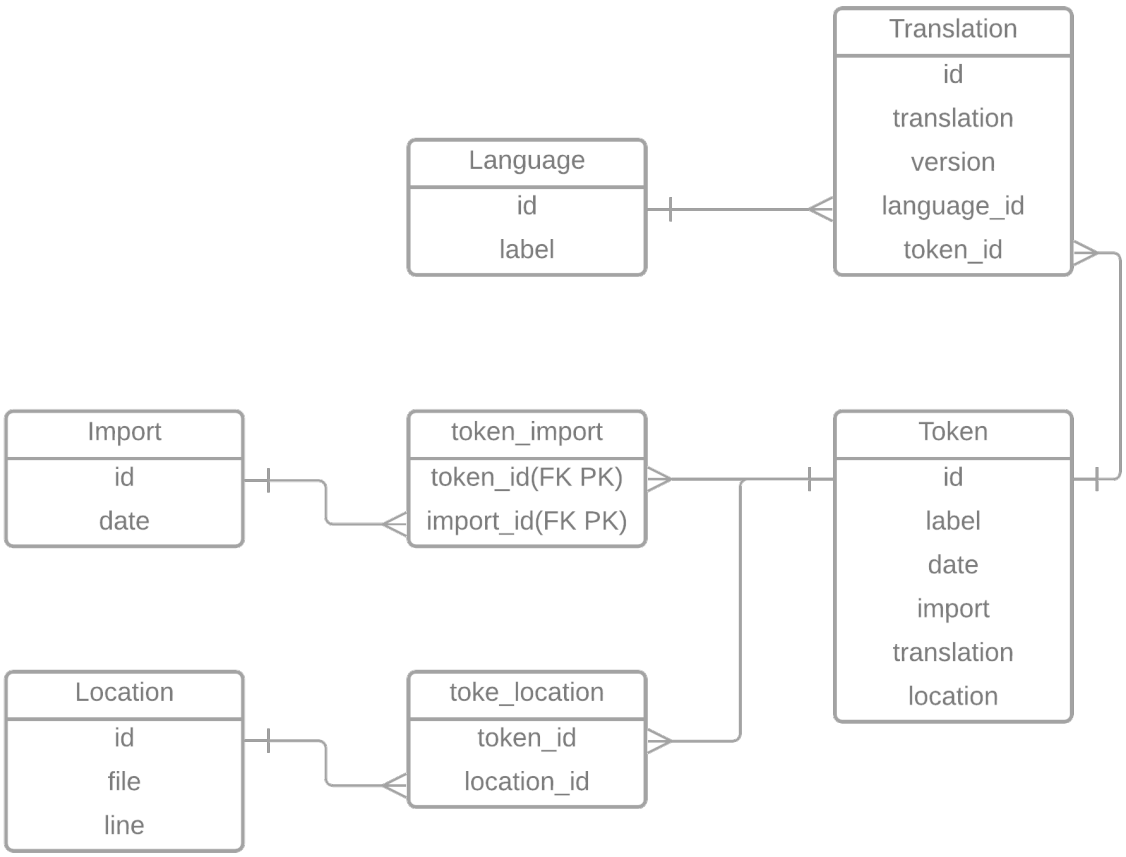

Abbildung 2: ERD im IST Zustand

A.3.2 Finale Datenbankstruktur

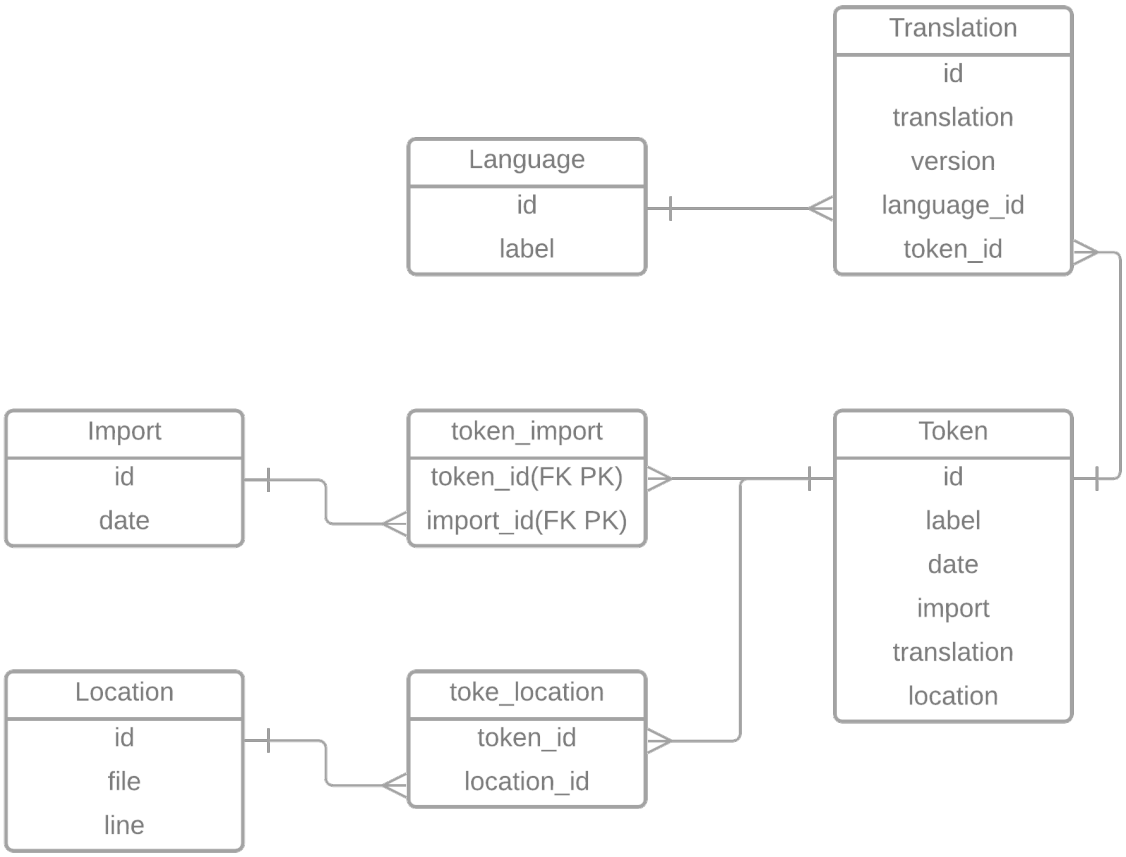

Abbildung 3: ERD im Soll Zustand

A.4 UML Anwendungsfalldiagramm

A.4.1 OkotoPOS Cash

A.4.2 Translationservice

A.5 Komponentendiagramm

A.6 Schnittstelledokumentation

Swagger

A.7 Resourcen und Technologien

1	2
3	4

Tabelle 5: Genutzte Resourcen

A.8 Testcases Translationservice

A.9 Testcases Translationservice

A.10 Iternationsplan

A.11 Nassi-Shneiderman-Diagramm

A.11.1 Datenbeschaffung

A.11.2 Cashdesk Injection

A.12 Translationservice Ui

B Listings

B.1 Many-To-Many Annotation

Token.php

```
/**
 * @var Version[]|ArrayCollection
 * @ORM\ManyToMany (targetEntity="OktoCareer\TranslationService\
    Version\Version", inversedBy="tokens")
 * @ORM\JoinTable (name="token_version",
 *      joinColumns = {
 *       @ORM\JoinColumn (name="token_id", referencedColumnName="id")
 * },
 *      inverseJoinColumns={
 *       @ORM\JoinColumn (name="version_id", referencedColumnName="id")
 * }
 *      )
 */
 private $version;
```

Version.php

```
/**
 * @var Token[]|ArrayCollection
 * @ORM\ManyToMany (targetEntity="OktoCareer\TranslationService\
    Token\Token",mappedBy="version")
 */
 private $tokens;
```

B.2 Routing

HTTP GET Route:

```
$app->get('/api/v1/version/{version}/translations/{language}'
  ,ExportTranslationAction::class);
```

HTTP POST Route:

```
$app->post('/api/v1/version/{version}/diff/translations/{language}'
  ,CreateTranslationDiff::class);
```

B.3 Beispielimplementierung einer GET-Route

B.4 Beispielimplementierung einer POST-Route

B.5 Configurtion.ini

B.6 Singlton