Inhaltsverzeichnis

1 Das Unternehmen Nürnberger Versicherung 1
1.1 Vorstellung Unternehmen 1
1.2 Vorstellung Abteilung 2
1.3 Vorstellung Aufgabe 3

2 Projektplanung 4
2.1 Projektziele 4
2.2 Technologie Auswahl 6
2.2.1 Frontend: Angular 6
2.2.2 Backend: Spring Boot vs ASP.Net 6
2.2.3 Datenbank: MSSQL vs Oracle 8
2.2.4 Architektur: Hexagonal 8

3 Entwicklung 10
3.1 Backend 10
3.1.1 Projektstruktur 10
3.1.2 Entitäten-Implementierung 11
3.1.3 Interface-Initialisierung 12
3.1.4 Service-Implementierung 13
3.1.5 Controller-Implementierung 16
3.2 Frontend 16
3.3 Datenbank 19

4 Bewertung der Technologien 20
4.1 Angular 20
4.2 Spring Boot/Java 20
4.3 MSSQL 20

5 Bewertung und Bezug 21

6 Fazit 23

Abbildungsverzeichnis **24**

Tabellenverzeichnis **25**

List of Listings **26**

Literaturverzeichnis **27**

Kapitel 1

Das Unternehmen Nürnberger Versicherung

1.1 Vorstellung Unternehmen

Mein Praktikum absolvierte ich vom 1. September 2024 bis zum 25. Januar 2025 bei der Nürnberger Versicherung. Die Nürnberger Versicherung (im Folgenden 'Nürnberger') bietet Finanzdienstleistungen an. Dazu zählen insbesondere, wie der Name schon sagt Versicherungen.

Zu diesen zählen unter anderem Lebensversicherungen, Krankenversicherungen, Rentenversicherungen und Sachversicherungen.

Unter dem Dach der Nürnberger Beteiligungs-AG bestehen mehrere spezialisierte Gesellschaften: Die Nürnberger Lebensversicherung AG bietet Lösungen zur finanziellen Vorsorge an, während die Nürnberger Allgemeine Versicherungs-AG Sachversicherungen abdeckt. Die Garanta Versicherungs-AG fungiert als berufsständischer Versicherer für das deutsche Kraftfahrzeuggewerbe.

Zur Schadensregulierung ist die Nürnberger Sofortservice AG tätig, und der Nürnberger AutoMobil Versicherungsdienst GmbH unterstützt Autohäuser. Die Nürnberger Krankenversicherung AG bietet private Krankenversicherungen, während die Nürnberger Pensionsfonds AG und die Nürnberger Pensionskasse AG Leistungen für die betriebliche Altersversorgung bereitstellen.

Zusätzlich erbringt die Fürst Fugger Privatbank KG Private-Banking-Dienstleistungen, und die Nürnberger Communication Center GmbH übernimmt Callcenter-Aufgaben. Die CodeCamp GmbH dient als Inkubator für Finanz- und Versi-

cherungsdienstleistungen, während die Nürnberger evo-X GmbH kundenorientierte Prozesse entwickelt und berät.

Die Nürnberger Beamten Allgemeine Versicherung AG bietet spezielle Tarife für den öffentlichen Dienst, und die Nürnberger Beamten Lebensversicherung AG, die sich in Abwicklung befindet, nimmt kein Neugeschäft mehr auf.[Wiki 24]

1.2 Vorstellung Abteilung

Wir in der Abteilung Anwendungsentwicklung-Leben-Produkte kurz AE-Leben-Pro entwickeln und warten die Software für die Lebensversicherungen. Dazu zählt primär die Tarif-/Produktverwaltung und versicherungsmathematische Berechnungen. Diese werden in Programmen auf sogenannten Rechenkernen durchgeführt.

Auf diesen Rechenkernen haben wir zum einen SST-Klassik. Dies ist ein C-Programm, dass für das DB2-Bestandführungssystem am IBM-Mainframe entwickelt wurde. Die Daten, also die Tarifinformationen, werden hier indirekt aus einer Produktdatenbank bezogen.

Zusätzlich haben wir noch SST-Referenz. Wie der Name schon andeuten lässt, entstand SST-Referenz aus SST-Klassik. Es ist ebenfalls ein C-Programm, dient jedoch als Test für das neuere Bestandführungssystem Life Factory.

Life Factory ist eine Client-Server Anwendung. Zugehörig zur Life Factory ist der Rechenkern Life Produkt. Das ist ein Java Programm.

Außerdem haben wir noch drei weitere Systeme, dazu gehört die Produktdatenbank, Solvency II und der IBM i (früher AS400).

Die Produktdatenbank ist eine DB2-Datenbank, in der wirklich alle wichtigen Produktinformationen aller Tarife gespeichert werden. Es ist also der Kern von AE-Leben.

Solvency II ist ein europäisches Aufsichtsregime für Versicherungen, das seit dem 1. Januar 2016 gilt. Es legt moderne Solvabilitätsanforderungen fest, die auf einer ganzheitlichen Risikobetrachtung basieren. Vermögenswerte und Verbindlichkeiten werden nach Marktwerten bewertet. Ziel ist es, das Insolvenzrisiko von Versicherern zu verringern und das Aufsichtsrecht im europäischen Binnenmarkt zu harmonisieren. [Fina 16] Für diesen verpflichtenden Nachweise liefern wir eine Modellrechnung,

das sogenannte Leistungsspektrum erster Ordnung (LS1). Hier wird der gesamte Bestand der Nürnberger Leben bewertet. Die LS1-Programme sind in Java geschrieben und rufen sowohl SST-Klassik als auch SST-Referenz auf. Die Verträge werden über XML-Schnittstellen geliefert.

Der IBM i ist ein mittelgroßer Rechner, auf dem verschieden kleineren Anwendungssystem laufen, die Verträge mit Produkten verwalten, die einfacher zu handhaben sind als die restlichen Tarifdaten. Dieser soll aber bis 2030 abgeschaltet werden. Die Programme auf dem IBM i sind alle in COBOL geschrieben und dementsprechend schon recht alt.

1.3 Vorstellung Aufgabe

Die Infrastruktur der Nürnberger Versicherung besteht schon seit einiger Zeit. Die meisten Programme sind in C oder COBOL geschrieben. Es werden überflüssige Felder in den Datenbanken gespeichert und viele Workflows sind nicht mehr zeitgemäß und eher unhandlich.

Die Life Factory entstand als Antwort auf diese Probleme. Jedoch kann sie nicht alles lösen. Es gibt immer noch keine angenehme, zuverlässige Variante schnell und einfach neue Tarife oder Produkte zu erstellen oder diese zu ändern.

Zusätzlich kommt jeden November noch die Überschussneuberechnung hinzu. Diese ist ein sehr aufwendiger Prozess, der viel Zeit in Anspruch nimmt, da viele Abschnitte dieser Berechnung nicht automatisiert sind und dementsprechend von Hand durchgeführt und überprüft werden müssen.

Meine Aufgabe besteht im groben darin genau diese Möglichkeit zu schaffen. Eine Webanwendung die es ermöglicht die Inhalte der Tarife und Produkte sowie deren Sub-Entitäten zu erstellen, zu ändern, zu löschen und anzuzeigen. Zusätzlich ist es geplant in einer neuen Datenbank die Daten der Produktdatenbank (PDB) gekürzt zu speichern. Also bereits redundante und unnötige Felder sowie Tabellen zu entfernen. Die Aufgabe hier ist es die Entwicklung der Datenbank, des Backends sowie des Frontends zu übernehmen und allein durchzuführen.

Kapitel 2

Projektplanung

2.1 Projektziele

Die groben Projektziele ergeben sich aus der Projektaufgabe. Konkret umfasst dies die Implementierung der folgenden Entitäten: `Tarif`, `Tarifbaustein`, `Tarifbaustein-Pricing`, `Tafelsystem`, `Überschuss`, `Produkt` sowie `Produkt-Pricing`.

Ein `Tarif` besteht jeweils aus mehreren `Tarifbausteinen`. Ein `Tarifbaustein` wiederum aus mehreren `Tarifbaustein-Pricings`. `Tarifbausteine` können null-drei `Tafelsysteme` und `Überschüsse` haben. Ein `Produkt` besteht aus mehreren `Produkt-Pricings`.

Neben den Grundfunktionen wie CRUD (Create, Read, Update, Delete), die alle Entitäten haben sollen, gibt es noch spezielle Funktionen. Ein Tarif sowohl als auch ein Produkt sollen deaktiviert werden können. Dies dient ebenfalls als Voraussetzung für eine Löschung. Bei der Löschung von Tarifen soll darauf geachtet werden, dass `Überschüsse` und Tafelsysteme, die keinem Tarifbaustein mehr zugehörig sind auch entfernt werden um keine bezugslose Entitäten zu speichern.

Dementsprechend sollen alle Kind-Entitäten auch nur mit der Erstellung eines Vater-Elements angelegt werden können. Bei der Erstellung neuer `Tarife` soll es ebenfalls möglich sein einen bereits bestehenden `Tarif` als Vorlage zu verwenden. Hier ist es wichtig, dass nicht nur die direkten Tarifinformationen, sondern auch alle Daten der zugehörigen Kinder-Entitäten übernommen werden.

Zusätzlich zu den bereits genannten Funktionen muss es möglich sein die Inhalte der Entitäten zu exportieren. Dies geschieht in Form einer C-Datei.

Die C-Datei hat folgende Struktur:

```
1 #include "example.h"
2
3 struct uesy uesy[] = {
4   // Property names in order of appearance
5   { 0, 4, 5, 337 },
6   { 0, 3, 4, 338 }
7 };
8
9 long uesyCount = 323;
```

Listing 2.1: C-Datei Beispiel

Anzumerken ist, dass in der C-Datei als deaktiviert markierte Tarife oder Produkte nicht exportiert werden sollen. Zusätzlich sollen nur die Inhalte der aktuell ausgewählten Entität übernommen werden, also ohne Kind Daten. Diese Funktion ist besonders wichtig, da die Inhalte der C-Datei von anderen Anwendungen benötigt werden. Besonders sticht hier SST-Referenz heraus.

Die letzten Funktionalitäten dienen allein den `Überschüssen`. Die `Überschüsse` werden wie bereits im vorherigem Kapitel erwähnt, jedes Jahr neu berechnet. Dementsprechend benötigt es diverse Funktionen, um dies zu erleichtern.

Dazu gehört die Neuberechnung des Gesamtzins. Die Umsetzung ist so geplant, dass der neue Gesamtzins als Input eingegeben wird. Nun werden alle `Überschüsse` mit einem neuen Eintrag angelegt, der bis Ende des folge Jahre gültig ist. Die gültigBis Property des vorherigen Eintrags wird folglich auf das Ende des aktuellen Jahres geändert. Der neue Eintrag ist letztlich eine vollständige Kopie des alten Eintrags, jedoch mit dem neuen Gesamtzins. Dies ist der Fall, falls sich die Gesamtverzinsung in einem Jahr ändert.

Nun gibt es noch die Möglichkeit, dass sich der Gesamtzins nicht ändert. In diesem Fall müssen nicht alle Einträge geändert werden, sondern nur ausgewählte. Dafür muss es möglich sein einen Eintrag fortzuschreiben wie bei der Änderung des Gesamtzins. Also ein neuer Eintrag mit neuer Gültigkeit und aktualisieren des alten Eintrags. Im neuen Eintrag werden dann die Werte, die sich geändert haben, manuell angepasst. Nach der Anpassung müssen alle unveränderten Einträge ebenfalls fortgeschrieben werden. Also muss es eine Funktion geben, die alle aktuellen Einträge kopiert und die Gültigkeit erneuert.

2.2 Technologie Auswahl

2.2.1 Frontend: Angular

Als Frontend-Framework wurde Angular ausgewählt. Diese Entscheidung war naheliegend, da Angular das am weitesten verbreitete Framework innerhalb der Nürnberger ist und mein Chef ausdrücklich den Einsatz dieses Frameworks gewünscht hat.

Auch aus meiner Sicht ist Angular eine sinnvolle Wahl. Es ist mein bevorzugtes Framework, da ich bereits mehrere eigene Projekte damit umgesetzt habe und auch innerhalb der Nürnberger umfangreiche Erfahrungen damit sammeln konnte. Dadurch bin ich mit der Entwicklungsumgebung, den Werkzeugen und Best Practices bestens vertraut.

Zwar hätte ich es interessant gefunden, mich in andere Frameworks wie Vue.js oder Svelte einzuarbeiten, doch die Entscheidung für Angular bringt klare Vorteile: Meine bestehende Erfahrung ermöglicht eine schnellere Entwicklung und einen saubereren Code. Darüber hinaus erleichtert die Verbreitung von Angular innerhalb des Unternehmens die spätere Einarbeitung und Zusammenarbeit mit Kollegen, falls das Projekt übergeben oder erweitert werden muss.

2.2.2 Backend: Spring Boot vs ASP.Net

Im weiteren Verlauf der Planung wurde darüber diskutiert, ob die Umsetzung des Backends lieber mit Java oder C# erfolgen soll. Sowohl Java als auch C# sind objektorientierte Programmiersprachen die ausreichend Tools zur Entwicklung von Web- und CRUD- (Create, Read, Update, Delete) Anwendungen bieten. Das heißt das jegliche Sprache für die von uns definierten Projektziele mehr als ausreichend ist.

Auf der Seite von Java stehen Spring Boot für die Web-Funktionalität und Spring Data JPA für die Abstrahierung der Datenbank zur Verfügung. C# auf der anderen Seite bietet ASP.Net Core für die Web-Funktionalität und Entity Framework Core für die Abstrahierung der Datenbank. Da ich bereits reichlich Erfahrung in der Entwicklung mit C#, sowie ASP.Net und EFC habe, war dieses auch meine persönliche Präferenz für das Projekt. Innerhalb der Nürnberger ist jedoch C# nicht

weitreichend genutzt und stattdessen wird Java bevorzugt. Dieser Meinung ist auch mein Betreuer, jedoch lies er mir die Option offen, mit ausreichend Argumenten, auch C# zu verwenden. Dementsprechend recherchierte ich die Vor- und Nachteile beider Technologien und stellte diese gegenüber.

Es gibt zwei starke Argumente die für die Verwendung von der Microsoft Umgebung rund um C# sprechen. Hier ist das Hauptaugenmerk auf meine weitreichende Erfahrung in der Entwicklung mit C# und ASP.Net zu legen. Die nicht notwendige Einarbeitung in eine neue Sprache und Umgebung würde definitiv die Entwicklung beschleunigen. Ebenso würde das Vorwissen vermutlich auch die Qualität des Codes erhöhen, da ich bereits Methoden kenne, wie man Architektur, Entität-Beziehungen und Datenbankzugriffe effizient umsetzt. Ein weiterer Vorteil ist C# leicht bessere Performance und deutlich besseres Speichermanagement.

Die genauen Zahlen sind in [Debi] dargestellt. Betrachtet man die Algorithmen *fannkuch-redux* und *n-body*, zeigt sich, dass C# effizienter ist als Java.

Beim *fannkuch-redux* benötigt C# durchschnittlich 23,04 Sekunden und 29.184 KB-Speicher, während Java mit 28,38 Sekunden etwa 19 % langsamer ist und mit 43.582 KB etwa 50 % mehr Speicher verbraucht.

Beim *n-body* liegt C# mit 5,42 Sekunden Laufzeit und 28.946 KB-Speicher ebenfalls vorn. Java benötigt hier 7,36 Sekunden (26 % langsamer) und 42.226 KB (48 % mehr Speicher).

In beiden Algorithmen ist C# sowohl schneller als auch speichereffizienter. Java zeigt konstant einen höheren Speicherverbrauch, was auf die JVM-Architektur zurückzuführen sein könnte.

Das Hauptargument auf der Seite von Java ist jedoch die weitreichende Nutzung innerhalb der Nürnberger Versicherung. Dies resultiert in zu einem besseren Support innerhalb des Unternehmens. Ebenso ist das Maintaining dieser Anwendungen simpler, da die meisten Entwickler hier bereits Erfahrung in Java haben. Dementsprechend ist es für einen möglichen nachfolgenden Entwickler leichter in das Projekt einzusteigen und gegebenenfalls zu übernehmen. Das war auch der überzeugenden Punkt für die Verwendung von Java.

Die Argumente für C# und weg von Java waren letztlich nicht stark genug. Zusätzlich bedeutet das für mich auch eine Herausforderung, in der ich eine neue weitere Sprache und Umgebung lernen kann.

2.2.3 Datenbank: MSSQL vs Oracle

Der nächste Schritt war die Auswahl der Datenbank. Zur Diskussion standen MSSQL und Oracle, beides relationale SQL-Datenbanken, die die notwendigen Funktionen für das Projekt bieten. Eine grundlegende Anforderung meines Chefs war, dass es sich um eine SQL-Datenbank handeln muss.

In einem gemeinsamen Gespräch mit den internen Architekten wurde schließlich entschieden, dass MSSQL die bessere Wahl für dieses Projekt ist.

Auch wenn Oracle eine gängige Datenbank innerhalb der Nürnberger ist, ist sie teuer und zu überdimensioniert für die Anforderungen des Projekts. MSSQL hingegen ist kostengünstiger und ist für die Datenmengen und möglichen Herausforderungen des Projektes mehr als ausreichend. [Petr 05]

2.2.4 Architektur: Hexagonal

Der letzte Planungsabschnitt ist die Auswahl der Architektur. Hier wurde mir komplette Freiheit gelassen. Die einzige Bedingung hierbei war, dass die Logik aufgeteilt wird und somit eine Schichtentrennung erfolgt.

Hierbei fiel meine Entscheidung auf die Hexagonale Architektur.

Das Grundprinzip der Hexagonalen Architektur ist es die Anwendung in verschiedene Schichten aufzuteilen. Diese unterschiedlichen Schichten sind von außen nach innen abhängig voneinander. Das heißt das jede innere Schicht nichts von einer Schicht weiter außen weiß. In der Regel ist die Model-Schicht die innerste Schicht. Dann kommt die Service-Schicht mit der Anwendungslogik und die Controller-Schicht mit den APIs als Schnittstelle zum User-Interface.

Im meinem konkreten Fall bedeutet das, dass die Web-Layer alle Schnittstellen anbietet die mein Fronted benötigt. Darunter zählen `GET`-Anfragen um Daten zu bekommen oder `POST`-Anfragen um Daten zu speichern. Die `Domain-Layer` ist die eigentliche Logik der Anwendung. Hier finden alle Berechnungen und Datenverarbeitungen, wie das Ändern der `Überschüsse` oder die Generierung der C-Datei, statt. Die `Data-Layer` ist die Schnittstelle zur Datenbank. Hier werden alle Datenbankzugriffe durchgeführt. In dieser Schicht sind auch alle Entitäten definiert.

In 2.1 ist die Architektur des Systems dargestellt. Der Nutzer interagiert mit dem Frontend. Die Anfrage wird an die `Web-Layer` geschickt. Diese ruft eine Methode im Service auf. Der Service in der `Domain-Layer` führt die Anfrage aus und ruft, falls Daten benötigt oder bearbeitet werden, die `Data-Layer` auf. Die `Data-Layer` greift auf die Datenbank zu und gibt die Daten zurück.

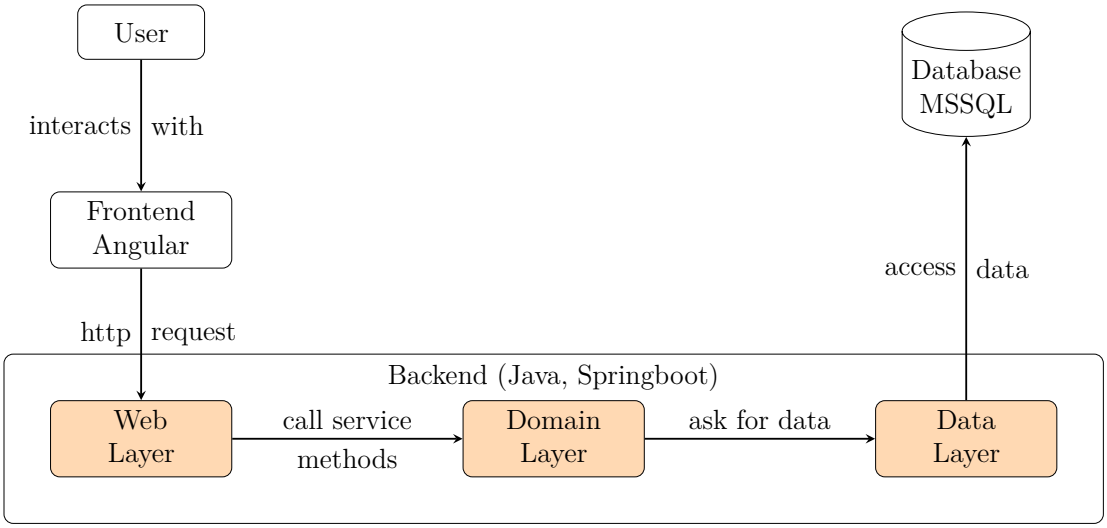

Abbildung 2.1: Architektur des Systems

Diese Architektur bietet mehrere Vorteile. Durch die genannte Aufteilung ist es einfacher, einzelne Schichten zu ändern und auszutauschen. So ist es beispielsweise dem `Web-Layer` egal, welches Frontend verwendet wird, und dem `Data-Layer`, welche Datenbank zum Einsatz kommt. Ebenfalls kann die Business-Logik unabhängig von den anderen Schichten getestet werden. Das wichtigste Argument für diese Architektur war jedoch, dass sie zum einen die Anforderung der geforderten Trennung erfüllt und die einzige Architektur ist, die ich bereits in der Praxis verwendet habe.

Kapitel 3

Entwicklung

3.1 Backend

3.1.1 Projektstruktur

Die Projektstruktur beziehungsweise die grobe Implementierung der Architektur wurde bereits in der Planung festgelegt. Der erste Schritt zur Erstellung war die Initiierung des Projekts mit dem sogenannten `Spring-Initializr`. Hierbei wurden folgende Dependencies hinzugefügt: `Spring-Web`, `Spring-Data-JPA`, `H2-Database`, `Microsoft-SQL-Server`, `Spring-Doc` und `Spring-Boot-DevTools`.

Die Projektstruktur wurde dann initial als drei Ordner `domain`, `application` und `infrastructure` verwirklicht. Der Ordner `domain` enthält die Entitäten, der Ordner `application` die Services und der Ordner `infrastructure` die Controller. Zusätzlich dazu wurde noch ein Konfigurationsordner erstellt, der die `Cors`-Einstellungen festlegt. Des Weiteren gab es noch ein SQL-Skript für die Testdaten und eine `application.properties`, um die Datenbank zu konfigurieren.

Die Fremdheit zu Java an sich erschwerte jedoch die initiale Umsetzung der Architektur. Die ursprüngliche Implementierung war nicht optimal und erfüllte die Schichtenarchitektur nicht.

Im weiteren Verlauf wurde die Struktur so umgeändert, dass ich drei verschiedene Projekte angelegt habe: Ein Projekt `Domain` für die Entitäten und Repository-Interfaces, ein Projekt `Web` für die Controller und Datenbankkonfiguration, sowie ein Projekt `Data`, welches alle Services und die `Cors`-Konfiguration enthält. Jedes dieser Projekte hatte eine eigene `pom.xml`-Datei, die die Abhängigkeiten und den

Build-Prozess regelt. So entsteht ein zusammenhängendes Projekt mit einer klaren Struktur und Trennung.

3.1.2 Entitäten-Implementierung

Die Entitäten sind das Grundgestein des Backends und auch der Datenbank. Hier habe ich zuerst mit meinem Chef in mehreren Gesprächen definiert, welche Entitäten wir für das Projekt benötigen.

Die zu implementierenden Entitäten belaufen sich auf: `Tarif`, `Tarifbaustein`, `Tarifbaustein-Pricing`, `Produkt`, `Produkt-Pricing`, `Tafelsystem` und `Überschuss`.

Die Entitäten wurden als Klassen in dem Projekt `Domain` implementiert. Hierfür wurden zuerst die Properties der Klasse hinzugefügt und dann Getter und Setter geschrieben. Da die Primärschlüssel von allen Entitäten bis auf `Tarif` und `Produkt` aus mehreren Properties bestehen, habe ich zusätzlich die Annotation `@IdClass` hinzugefügt. Das heißt, ich habe für die Schlüssel eine extra Klasse erstellt, die das Interface `Serializable` implementiert und die Properties enthält. Dies ermöglicht mir ein Importieren des zusammengesetzten Schlüssels in die Entitäten.

Anschließend wurden die Entitäten mit der JPA-Annotation `@Entity` und `@Table` versehen, um die Verbindung zur Datenbank zu ermöglichen.

Der nächste Schritt war es, die Beziehungen zwischen den Entitäten hinzuzufügen. Hierbei gab es mehrere einfache 1:n Beziehungen: `Tarif` zu `Tarifbaustein`, `Tarifbaustein` zu `Tarifbaustein-Pricing` und `Produkt` zu `Produkt-Pricing`.

Die Beziehung zwischen `Tarifbaustein` und `Tafelsystem` sowie `Überschuss` waren jedoch ein bisschen komplexer. Hier hatte ich ursprünglich, anhand der Vorbesprechungen, jeweils drei n:m Beziehungen geplant. Dementsprechend habe ich diese Beziehungen als je drei Listen implementiert, die die Keys der jeweils anderen Entitäten als Fremdschlüssel nutzen. Dies war für mich eher schwierig umzusetzen, da ich in Java noch nicht mit den Annotationen `@ManyToMany` und `@JoinTable` gearbeitet hatte.

In weiteren Gesprächen mit meinem Chef wurde dann letztlich deutlich, dass es keine drei n:m Beziehungen sind, sondern drei 1:n Beziehungen. Obwohl die Beziehungen bereits implementiert waren, fand ich die Notwendigkeit, die Entitäten

anzupassen, gut. Diese Änderung nimmt nämlich einiges an Komplexität aus dem Projekt heraus und erleichterte mir somit alle weiteren Schritte zur Implementierung der Services und Controller.

Nachdem die Beziehungen definiert waren, habe ich bei den Kind-Entitäten die Annotation `@JsonIgnore` hinzugefügt, um bei der JSON-Serialisierung eine korrekte Darstellung der Beziehungen sicherzustellen und zyklische Abhängigkeiten zu vermeiden.

Im späteren Verlauf wurde noch eine Funktion zum String-Serialisieren der Properties hinzugefügt. Dies wird benötigt, um die Entitäten in die C-Datei schreiben zu können.

Zusätzlich wurden alle Getter und Setter entfernt, da ich im Seminar von dem Package `Lombok` erfahren habe und die `@Getter` und `@Setter` Annotationen die Klassen lesbarer machen.

3.1.3 Interface-Initialisierung

Die Interfaces für die Entitäten wurden in dem Projekt `Domain` erstellt. Danach wurden die Annotation `@Repository` hinzugefügt, dies kennzeichnet intern die Klasse als Repository und ermöglicht die Verwendung der JPA-Annotationen.

Spring Data JPA bietet zusätzlich die Möglichkeit diverse Grundfunktionen zu implementieren. Hierfür erweitert man das die eigenen Interfaces mit dem Spring Interface `JpaRepository`. Diese Erweiterung wurde in den Interfaces hinzugefügt und die Entitäten als Typenparameter übergeben.

Die Funktionen, auf die man nun in allen Entitäten zugreifen kann, sind unter anderem `save`, `findAll`, `deleteAll`, `saveAll`, `getAll`, `getById` und viele weitere. Diese Möglichkeit, die CRUD-Operationen so zu implementieren, erspart einem viel Zeit und Arbeit.

Als extra Query habe ich noch eine Funktion zum Finden von leeren Überschüssen und Tafelsystemen hinzugefügt. Leer bedeutet in diesem Fall, dass die Liste der Tarifbausteine leer ist. Dies wird benötigt, um verwahrloste Kind-Entitäten zu finden und zu löschen.

Der Aufbau dieser Query ist folgendermaßen:

```
1 @Query("SELECT t FROM Tafelsystem t WHERE
2 t.tarifbausteine IS EMPTY")
3 List<Tafelsystem> findEmptyTafelsysteme();
```

Listing 3.1: Interface Query

Des Weiteren habe ich noch eine Query findMinUesel() hinzugefügt, um das momentan niedrigste Üsel zu finden. Diese Funktion wird benötigt, da in einer anderen Funktion Überschüsse kopiert werden und die Generierung der Üsel manuell geschehen muss.

3.1.4 Service-Implementierung

Die Services sind die Klassen, die die Logik enthalten. Hier werden die Daten verarbeitet und die Operationen durchgeführt. Hier wurden für alle Entitäten jeweils eine Service-Klasse erstellt. Um diese als Service zu kennzeichen, wurde die Annotation `@Service` hinzugefügt. Zusätzlich wurde noch ein Service für die Suche und ein weiterer für die Generierung der C-Datei hinzugefügt.

Die Grundstruktur der Entitäten-Service ist immer gleich. Zuerst wird das Repository injected, um auf die Methoden des JpaRepositories zugreifen zu können. Ich habe mich hierbei für die sogenannte Dependency Injection entschieden. In Java sieht das beispielsweise so aus:

```
1 @Service
2 public class ProduktService{
3   private final ProduktRepository produktRepository;
4
5   public ProduktService(ProduktRepository
6   produktRepository){
7     this.produktRepository = produktRepository;
8   }
9 }
```

Listing 3.2: Dependency Injection

Diese Variante fördert eine lose Kopplung, verbessert die Testbarkeit, erhöht die Flexibilität durch austauschbare Implementierungen und vereinfacht das Konfigurationsmanagement, was insgesamt zu wartbarer und anpassungsfähiger Software führt. [Seem 19]

Danach werden die CRUD-Operationen, die in den Interfaces schon vorgegeben wurden, implementiert.

Hierfür reicht es aus einen Methodenrumpf zu definieren und dann die jeweilige Methode des Repositories aufzurufen. Anhand des Beispiels Produkt-Service sieht das so aus:

```
1  public List<Produkt> findAll(){
2    return produktRepository.findAll();
3  }
4
5  public Produkt findById(String produktId){
6    return produktRepository.findById(produktId);
7  }
8
9  public Produkt createProdukt(Produkt produkt){
10   return produktRepository.save(produkt);
11 }
12
13 public void deleteProdukt(String produktId){
14   produktRepository.deleteById(produktId);
15 }
```

Listing 3.3: Service Implementierung

Im weiteren Verlauf wurde für den Tarif-Service noch die Methode zum Löschen von Elternlosen Tafelsystemen und Überschüssen hinzugefügt. Diese Funktion wird aufgerufen, wenn ein Tarif gelöscht wird.

Die meisten zusätzlichen Funktionen wurden im Überschuss-Service hinzugefügt. Hier wurde die Funktion `changeUeberschussSatz()` hinzugefügt, um die aktuelle Gesamtverzinsung jährlich ändern zu können. Besonderheit bei dieser Funktion ist die benötigte Annotation `@Transactional`. Diese Annotation sorgt dafür, dass bei Fehlern alle Änderungen zurückgerollt werden.

Um die Anforderung zu erfüllen, dass Überschüsse fortgeschrieben werden können wenn der Gesamtzins unverändert bleibt, wurde die Funktion `overWriteUesy()` hinzugefügt. Diese Funktion legt eine Deep-Copy von dem mitgegebenen Überschuss an, setzt das Datum des alten Überschusses auf das Ende des aktuellen Jahres, und das Datum des neuen Überschusses auf das Ende des nächsten Jahres. Danach werden beide Überschüsse gespeichert.

Die vorherige Funktion spiegelt letztlich den Prozess wider, Überschüsse manuell zu kopieren und zu verändern. Ein Problem hierbei ist jedoch, dass nicht alle Überschüsse geändert werden. Jedoch muss man diese auch fortschreiben ins nächste Jahr. Hierfür wurde eine weitere Funktion `updateUnchangedUesy()` hinzugefügt. Diese Funktion sucht alle Überschüsse, bei dem kein deklariertBis mit dem folgenden Jahr als Wert existiert und setzt dann das deklariertBis auf das Ende des nächsten Jahres.

Eine weitere notwendige Funktion war die `splitUesyHistory()`. Diese Funktion wird benötigt, um Überschüsse aufzutrennen. Der Hintergrund ist hierbei der folgende: Momentan werden Überschüsse manchmal für mehrere Tarife genommen, da die Werte gleich sein sollen. Jedoch können die Anforderungen an die Überschüsse sich ändern. Da wir die Historie der Überschüsse benötigen, muss diese also aufgeteilt und kopiert werden. Dafür ist diese Funktion zuständig.

Wie zuvor erwähnt wurden noch zwei weitere Services implementiert. Der `CFile-GenerationService` und der `SearchService`.

Der `CFileGenerationService` implementiert eine generische Funktion für die Generierung des C-Files. Diese Methode wird dann in den jeweilige Entität-Services aufgerufen. Hierfür war es zusätzlich notwendig den CFileGenerationService zu injecten.

Der `SearchService` implementiert ebenfalls eine generische Funktion. Für die Suche gibt es zwei Optionen: direkte Suche eines Wertes in allen Feldern oder Suche eines Wertes in einem bestimmten Feld. Die Methode berücksichtigt dabei, ob die Suchanfrage in der Form `key=value` oder einfach nur `value` vorliegt. Für jede Entität gibt es einen eigenen Controller-Endpunkt zur Durchführung der Suche.

3.1.5 Controller-Implementierung

Die Controller sind die Schnittstelle zwischen dem Frontend und dem Backend. Hier werden die REST-Apis implementiert, die die Daten an das Frontend senden. Der erste Schritt war die Implementierung aller Controller Klassen. Hierfür erstellt man eine normale Java Klasse und annotiert diese mit `@RestController` und `@RequestMapping("path")`. Wie bei den anderen bis jetzt erwähnten Annotationen, diene diese der Kennzeichnung für Spring.

Der nächste Schritt war die Injection der Services in die Controller. Das läuft genauso ab, wie in den Services.

Der letzte Schritt war die Implementierung der REST-Endpunkte. Hierfür habe ich in jedem Controller die Schnittstellen `getAll`, `getById`, `generateCFile`, `create`, `update`, `deleteById` und `search` hinzugefügt. Bei jeder Schnittstelle musste noch die passende Annotation, die den Request-Type angibt, hinzugefügt werden.

Im späteren Verlauf kamen noch weitere spezifischere Endpunkte hinzu. Dazu gehören zum Beispiel die Endpunkte zum Deaktivieren von Tarifen und Produkten oder zum ändern der aktuellen Gesamtverzinsung.

3.2 Frontend

Das Frontend, also die Benutzeroberfläche, hat mehr Zeit in Anspruch genommen, als ursprünglich angenommen.

Angefangen habe ich indem ich ein neues Angular Projekt, mithilfe des Command-Line-Interface, erstellt habe. Dies erstellt eine Grundstruktur für das Projekt und definiert benötigte Abhängigkeiten.

Danach war es mein erstes Ziel, die Api-Definition und einen Service zu erstellen, über den die Kommunikation zum Backend läuft. Hierfür habe ich zuerst die Api-Definition mit dem Swagger-Editor erstellt und in das Projekt eingefügt. Danach habe ich den Service erstellt der die Grundkonfiguration für die Kommunikation enthält. Die Klasse ist folgendermaßen aufgebaut:

```
1 export class ApiService {
2   public readonly basePath: string = "http://localhost";
3   public productService: ProduktControllerService;
4   constructor(private http: HttpClient) {
5     const configuration = new Configuration();
6     this.productService = new ProduktControllerService(
7       this.http, this.basePath, configuration);
8   }
9 }
```

Listing 3.4: Api-Service

Zusätzlich habe ich noch einen HTTP-Interceptor hinzugefügt. Dieser fängt alle HTTP-Requests ab. Momentan wird dieser nur genutzt um zwischen Aufbau und Abbau eines Requests eine Ladeanimation anzuzeigen. Jedoch ist es denkbar zukünftig, dass darin auch eine Fehlerbehandlung implementiert wird.

Mein nächster Schritt war es für jede Entität eine Komponente zu erstellen. In diesen habe ich dann eine Tabelle hinzugefügt, die alle Daten enthält.

Um zu diesen Komponenten navigieren zu können, habe ich eine Navigationsleiste hinzugefügt. Für das Routing hat Angular ein eigenes System. Dort werden alle Routen und die dazugehörigen Komponenten definiert. Anschließend fügt man diese in der `app.config.ts` als Provider hinzu.

Im weiteren Verlauf habe ich zwei Buttons in der Tarif- und Produkt-Komponente implementiert. Beim ersten Klick auf diesen Button wird der Tarif beziehungsweise das Produkt deaktiviert und beim zweiten Klick wird er gelöscht. Nachdem ersten Klick wird zusätzlich der zweite Button sichtbar. Dieser reaktiviert den Tarif beziehungsweise das Produkt.

Danach war es wichtig neue Entitäten hinzufügen zu können. Hierfür habe ich einen weiteren Button hinzugefügt. Dieser öffnet ein Dialog-Fenster mit allen Inputs die für die Entität notwendig ist. Um das zu ermöglichen habe ich für jede Entität eine Form-Komponente hinzugefügt. Das Dialog-Fenster um einen neuen Tarif hinzuzufügen besteht also aus mehreren Form-Komponenten.

In den Form-Komponenten habe ich dann zum einen noch die Validierung der Inputs hinzugefügt, damit nur richtige Werte gespeichert werden. Und zum ande-

ren habe ich noch Funktionen zum Löschen und hinzufügen von Tarifbausteinen und Tarifbaustein-Pricings hinzugefügt. Dies ist notwendig da ein Tarif mehrere dieser Entitäten besitzen kann. Zusätzlich habe ich noch Logik hinzugefügt, damit die Fremdschlüssel der Kinder automatisch mit dem Eltern-Primärschlüssel synchronisiert werden.

Bei der Implementierung des Dialog-Fensters war die größte Herausforderung die Kommunikation zwischen den Komponenten. Um das zu lösen habe ich die `ParentFormGroup` immer als Input in die Komponenten mitgegeben.

Als nächsten Punkt musste die Möglichkeit hinzugefügt werden, die Entitäten verändern zu können. Hierfür habe ich wieder einen Button implementiert, der die Felder zu Inputs macht und diese dann die veränderten Daten ans Backend sendet.

Hier ist mir aufgefallen, dass meine momentane Implementierung der Entitäten im Frontend nicht optimal ist. In der momentanen Umsetzung muss ich alle Erweiterungen für jede Entität hinzufügen. Das ist nicht nur sehr aufwendig, sondern auch fehleranfällig.

Meine Lösung war hierfür, dass ich eine neue Komponenten erstellt habe, die die Felder der Entität als Objekt übergeben bekommt und dann dynamisch alle Felder darstellt. Das sorgt dafür, dass die Implementierung nur noch in einer Komponente stattfindet und somit die Erweiterungen oder Änderungen nur an einer Stelle gemacht werden müssen.

Danach habe ich noch die restliche Funktionen hinzugefügt. Dazu gehören folgende Funktionen: `overWriteUesy()`, `changeGesamtZins()`, `splitUesyHistory()`, `updateUnchangedUesy()` und `search()`.

Die Suche ist wieder eine eigene Komponente und auch Service, der die Entität als Input bekommt und so entscheidet, an welche Api die Anfrage geschickt wird.

Nach der Implementierung der Suche gab es nur noch kleine Erweiterungen und Verbesserungen. Ein Beispiel hierfür ist eine Combo-Box, ein Dropdown-Menü mit integrierter Suche, um die Auswahl der Tarifbausteine, Überschüsse und Tafelsysteme zu erleichtern.

3.3 Datenbank

Die Datenbank wurde bislang noch nicht aufgesetzt. Für die Entwicklung wurde jedoch eine H2-Datenbank verwendet.

Diese ist eine In-Memory Datenbank. Das heißt, dass die Datenbank nur so lange existiert, wie die Anwendung läuft. Das hat den Vorteil, dass die Datenbank bei jedem Start der Anwendung neu initialisiert wird und somit immer der gleiche Ausgangszustand herrscht. Die Option die Daten in einer Datei zu speichern, wurde nicht genutzt, da ein Zurücksetzen nach einem Build-Prozess besser für die Entwicklung geeignet ist.

Um die Daten zu befüllen, wurde ein SQL-Script mit ein paar Test Daten für jede Entität geschrieben. Dieses Script wird bei jedem Start ausgeführt und füllt dann die Datenbank.

Die Datenbank wird dann in der weiteren Tätigkeit als Werkstudent beantragt, fertig aufgesetzt und verbunden.

Kapitel 4

Bewertung der Technologien

4.1 Angular

4.2 Spring Boot/Java

4.3 MSSQL

Da ich die Datenbank bisher noch nicht implementieren konnte, kann ich hierzu keine abschließende Bewertung abgeben. Ich möchte jedoch anmerken, dass ich mit MSSQL ohnehin kaum direkt hätte arbeiten müssen, da die Einrichtung von einer separaten Abteilung übernommen wird und die Konfiguration über Spring Boot erfolgt, anstatt direkt in der Datenbank.

Kapitel 5

Bewertung und Bezug

Im Rahmen des Studiums erhält man eine fundierte Einführung in verschiedene Technologien und deren Grundlagen. Dazu gehören Programmierkenntnisse, grundlegendes Wissen über Datenbanken, eine Einführung in Algorithmen und Datenstrukturen sowie die Basisprinzipien der Informatik.

Diese Inhalte bilden zwar ein solides Grundwissen, sind jedoch oft weit von den Anforderungen und der Arbeitsweise in der Praxis entfernt. Diese Erfahrung konnte ich unter anderem während meiner Tätigkeit bei der Nürnberger Versicherung machen.

Die an der Hochschule vermittelte Basis erleichtert zweifellos die initiale Einarbeitung in neue Technologien und Projekte. Allerdings sind viele der im Studium behandelten Themen in der Praxis stark abstrahiert oder werden in einer anderen Form umgesetzt. Dies erfordert eine ständige Weiterentwicklung und Lernbereitschaft, um mit aktuellen Technologien klarzukommen.

Ein gutes Beispiel dafür ist der Umgang mit Datenbanken. Im Studium lernt man den Umgang mit SQL und versteht, wie relationale Datenbanken strukturiert sind. In der Praxis wird SQL jedoch häufig durch Frameworks abstrahiert. Technologien wie Entity Framework oder Spring Boot generieren SQL automatisch und ermöglichen den Zugriff auf Datenbanken durch Abstraktionsebenen wie ORMs (Object-Relational Mappers). Dadurch tritt das direkte Arbeiten mit SQL oft in den Hintergrund, und der Fokus liegt vielmehr auf der korrekten Integration und Konfiguration dieser Frameworks.

Ähnlich verhält es sich mit Algorithmen und Datenstrukturen. Während diese Konzepte im Studium intensiv behandelt werden, um ein tiefes Verständnis für die Grundlagen der Informatik zu vermitteln, begegnen sie einem in der Praxis meist

nur indirekt. Häufig bieten Frameworks und Bibliotheken bereits optimierte Implementierungen für viele dieser Konzepte. Der Fokus liegt dann darauf, die eigenen Methoden der Programmiersprachen für die Implementierung dieser Konzepte zu kennen und richtig zu nutzen.

Trotz dieser Abstraktionen ist das im Studium vermittelte Wissen sehr wichtig, da es hilft, die Hintergründe und Funktionsweisen moderner Technologien zu verstehen. Dieses Verständnis erleichtert es, sich schnell in neue Frameworks oder Technologien einzuarbeiten und deren Grenzen sowie Einsatzmöglichkeiten besser zu erkennen.

Zusammenfassend lässt sich sagen, dass das Studium eine wichtige Grundlage legt, auf der man in der Praxis aufbauen kann. Die wahre Herausforderung besteht jedoch darin, diese Basis kontinuierlich zu erweitern, neue Technologien zu erlernen und sich den Anforderungen der beruflichen Realität anzupassen.

Kapitel 6

Fazit

Abbildungsverzeichnis

2.1 Architektur des Systems 9

Tabellenverzeichnis

List of Listings

2.1 C-Datei Beispiel 5

3.1 Interface Query 13
3.2 Dependency Injection 13
3.3 Service Implementierung 14
3.4 Api-Service 17

Literaturverzeichnis

[Debi] Debian. "C# .NET versus Java fastest performance". https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/csharp.html. Accessed: (06.01.2025).

[Fina 16] B. für Finanzdienstleistungsaufsicht. "Solvency II". https://www.bafin.de/DE/Aufsicht/VersichererPensionsfonds/Allgemeines/SolvencyII/solvency_II_node.html, 2016. Accessed: (14.12.2024).

[Petr 05] G. Petri. "A comparison of Oracle and MYSQL". *Select Journal*, Vol. 1, 2005.

[Seem 19] M. Seemann and S. van Deursen. *Dependency Injection Principles, Practices, and Patterns*, Chap. 1.2.2. Simon and Schuster, 2019.

[Wiki 24] Wikipedia. "Nürnberger Versicherung". https://de.wikipedia.org/wiki/NÃrnberger_Versicherung, 2024. Accessed: (15.11.2024).