

# Ymir

Eine Definitionssprache zum  
Generieren von Boilerplate-Code  
in der Webentwicklung

## PROJEKTDOKUMENTATION

ausgearbeitet von

Lukas Hufnagel, 11145562  
Paul Hufnagel, 11145561

vorgelegt an der  
TECHNISCHEN HOCHSCHULE KÖLN  
CAMPUS GUMMERSBACH  
FAKULTÄT FÜR INFORMATIK UND  
INGENIEURWISSENSCHAFTEN

im Studiengang  
MEDIENINFORMATIK

Prüfer/in: Prof. Dr. Hoai Viet Nguyen  
Technische Hochschule Köln

Gummersbach, im Mai 2023

**Adressen:** Lukas Hufnagel  
Dürscheider Straße 10  
51067 Köln  
lukas.hufnagel@smail.th-koeln.de

Paul Hufnagel  
Dürscheider Straße 10  
51067 Köln  
paul.hufnagel@smail.th-koeln.de

Prof. Dr. Hoai Viet Nguyen  
Technische Hochschule Köln  
Institut für Informatik  
Steinmüllerallee 1  
51643 Gummersbach  
viet.nguyen@th-koeln.de

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>3</b>
<b>2 Theoretische Ausarbeitung eines Lösungsansatzes</b>	<b>6</b>
2.1 Betrachtung des thematischen Umfelds . . . . .	6
2.2 Ausarbeitung von Ymir mittels Modellierung . . . . .	9
2.3 Mögliche Konkurrenten von Ymir . . . . .	12
<b>3 Entwurf und Umsetzung eines Boilerplate-Generators</b>	<b>15</b>
3.1 Die Idee zu Ymir . . . . .	15
3.2 Die Ymir-Qualitätsanforderungen . . . . .	16
3.3 Ymirs Semantik und Syntax . . . . .	18
3.4 Die praktische Umsetzung von Ymir . . . . .	21
3.4.1 Der Compiler . . . . .	21
3.4.2 Ziel-Plugins . . . . .	24
<b>4 Verschiedene Projekterweiterungen</b>	<b>27</b>
4.1 Das Beibringen weiterer Sprachen . . . . .	27
4.2 Plugin und Web-Präsenz für Ymir . . . . .	28
<b>5 Schlussteil der Projektdokumentation</b>	<b>30</b>
5.1 Ymir im Vergleich zu anderen . . . . .	30
5.2 Kritische Reflexion zum Projekt . . . . .	31
5.3 Ausblick auf die Zukunft von Ymir . . . . .	33
<b>Abbildungsverzeichnis</b>	<b>35</b>
<b>Literaturverzeichnis</b>	<b>37</b>

# Kurzfassung

Die Webentwicklung gewinnt immer mehr an Bedeutung und kann zur Erstellung von Desktop-Anwendungen, mobilen Anwendungen und sogar ganzen Plattformen genutzt werden. Die Mehrzahl der Webprojekte besteht aus Frontend- und Backend-Komponenten. Das Backend wird häufig über RESTful APIs angebunden. JavaScript wird sowohl in Frontend- als auch in Backend-Umgebungen häufig eingesetzt.

Die Verwendung beliebter Webframeworks wie Express oder NestJS kann sich als schwierig erweisen, da sie das Schreiben von redundantem und sich wiederholendem Code erfordern. Dies erhöht die Entwicklungs- und Wartungskosten und erschwert das Hinzufügen neuer Routen. Routes werden oft validiert, um sicherzustellen, dass alle Parameter gesetzt wurden, oder es müssen Authentifizierungsschritte hinzugefügt werden, um den Zugriff auf bestimmte Ressourcen zu beschränken.

Ymir ist ein neuer Compiler, der die Entwicklung von Webanwendungen beschleunigt. Diese Sprache basiert auf einer deklarativen Syntax namens YmirScript, die es Entwicklern ermöglicht, REST-Schnittstellen auf leicht verständliche Weise zu deklarieren. Außerdem bietet sie vorkonfigurierte Validierungen, Authentifizierungen und andere Optimierungen. Mit Ymir können Entwickler sowohl JavaScript-Code als auch andere Programmiersprachen generieren. Dieses modulare Framework bietet Entwicklern eine dynamische Flexibilität, die es ihnen ermöglicht, die volle Kontrolle über die Ausgabedateien zu haben. Dank dieses flexiblen Ansatzes kann Ymir theoretisch für die Codegenerierung in beliebigen Programmiersprachen verwendet werden, sofern die erforderlichen Anpassungen im Voraus vorgenommen werden.

Insgesamt stellen Ymir und seine Skriptsprache YmirScript eine innovative und wissenschaftlich fundierte Lösung dar, die darauf abzielt, die Effizienz und Wartbarkeit von Webprojekten zu verbessern. Sie ermöglicht es Entwicklern, sich auf die Kernfunktionen eines Projekts zu konzentrieren und gleichzeitig redundante Code-Bausteine zu reduzieren, wodurch die Overhead-Kosten gesenkt werden. Seine modulare Struktur ermöglicht eine große Flexibilität und unterstützt mehrere Programmiersprachen gleichzeitig.

# 1 Einleitung

Mit Boilerplate-Code sieht sich quasi jeder Programmierer früher oder später konfrontiert. Dieser Art von Code ist für die Lauffähigkeit eines Programms zumeist unabdingbar, jedoch muss er im Laufe der Entwicklung häufig wiederholt geschrieben werden, obwohl seine Auswirkung meist nur gering ist. Daher kommt es regelmäßig vor, dass der Aufwand dem Nutzen gegenüber überwiegt. Mit diesem Projekt soll eben jenes Problemfeld angegangen werden.

## Die Motivation hinter dem Projekt

Ein Entwickler kommt nicht um Boilerplate-Code herum, egal, in welchem Bereich. Überall dort, wo programmiert wird, lässt sich im Quellcode Boilerplate-Code finden. Als Boilerplate-Code verstehen Entwickler Code, welcher immer wieder geschrieben wird und auch benötigt wird, aber deren Funktionalität sich nur geringfügig auf das Gesamtprodukt auswirkt Zaveri (2018). Je nachdem, in welchem Bereich ein Entwickler tätig ist, desto präsenter und gegebenenfalls zeitaufwendiger ist Boilerplate-Code. So beinhaltet etwa der HTML-Header beim Programmieren einer Webseite solchen Code oder *Getter* und *Setter* in der objektorientierten Programmierung zählen zum typischen Boilerplate-Code. Während sich der eben genannte HTML-Header recht schnell in modernen Entwicklungsumgebungen generieren lässt, ist der Umgang mit solchem Code in der Objektorientierung, beispielsweise mit *Java*, teils deutlich aufwendiger. Da auch bereits Studenten der Informatik als Programmierer mit diesem Problem konfrontiert sind, gibt es damit einen praxisnahen Bezug zum Problem und somit auch zu einer möglichen Lösung.

**Erörterung einer Leitfrage** Um dieses Problem anzugehen, wurde zunächst eine Leitfrage erörtert, die dem gesamten Projekt zugrunde liegt. Dabei wurde die Leitfrage aus dem bereits bekannten „Boilerplate-Problem“ abgeleitet: *Wie würde eine Möglichkeit aussehen, um Boiler-plate für Webentwicklung zu generieren?*. Zu Beginn sollte die Leitfrage möglichst breit gefächerte Antwortmöglichkeiten bieten, um weiteren Verlauf weiter eingeschränkt zu werden und schließlich auf einen Lösungsansatz fokussiert zu werden. Obwohl bereits sehr früh klar war, wohin sich das Projekt entwickeln würde, sollte die Leitfrage dem Zweck dienen, auch Alternativen zu berücksichtigen und nicht bloß mit Tunnelblick zielstrebig auf einen einzelnen Lösungsansatz darauf hinzuarbeiten.

## Arbeitungsaufteilung im Zweier-Team

Statt das Praxisprojekt einzeln zu bearbeiten, ist es auch möglich als Zweier-Team daran zu arbeiten. Dies bietet die Möglichkeit umfangreichere Projekte umzusetzen,

erfordert jedoch auch ein größeres Maß an Vorplanung. Das liegt unter anderem auch daran, dass es dem Betreuer möglich sein soll zu verstehen, welches Teammitglied an was gearbeitet hat. Um den roten Faden der gesamten Projektdokumentation nicht zu gefährden, wird die Erläuterung der Arbeitsaufteilung daher bereits in diesem Unterabschnitt der Einleitung behandelt, so dass sich der Hauptteil voll und ganz, um das eigentliche Projekt drehen kann.

Aufgabe	Paul	Lukas
Recherche	80%	20%
Modellierung	95%	5%
Entwicklung: Compiler	10%	90%
Entwicklung: Ziel-Plugins	33%	67%
Entwicklung: Webseite	20%	80%
Entwicklung: VS Code Plugin	30%	70%
Quality Assurance	75%	25%

Abbildung 1.1: Arbeitsmatrix zum Ymir-Projekt (Hufnagel u. Hufnagel, 2023a)

Paul hat mit einem Fokus auf die Theorie an dem Projekt gearbeitet, wohingegen Lukas seinen Fokus auf die Praxis gelegt hat. Um dennoch die im Studium erlernten Fähigkeiten in beiden Bereichen anzuwenden, sind beide in allen Aufgabenbereichen zumindest zum Teil involviert. Unter den Punkt „Modellierung“ fallen alle Diagramme und Modelle, die im Laufe des Projekts entstanden sind. Unter „Recherche“ der restliche Teil der Theorie, der nicht zu den Modellierungsarbeiten gezählt werden kann. „Quality Assurance“ umfasst nicht nur den Testing-Bereich, sondern auch das daraus resultierende *Bugfixing*.

Mit dieser Arbeitsmatrix dürfte besser erkennbar sein, wer an was gearbeitet hat und somit eine Unterteilung einfacher fallen.

### Aufbau dieser Projektarbeit

Bevor nun der Hauptteil der Projektarbeit folgt, schlüsselt dieses Unterkapitel die Struktur der Arbeit, sowie die Zielsetzung dieser auf. Das nächste Kapitel handelt dabei von dem Theorieteil des Projekts. Dort wird mittels theoretischer Methoden das Themenfeld mitsamt Problematik erörtert und so der Rahmen des Projekts abgesteckt.

Im Anschluss wird in Kapitel 3 die Hauptphase des Projekts dargestellt. Dabei beginnt das Kapitel zunächst damit, konkrete Ideen und Vorstellungen zu Ymir benennen und festzusetzen, um diese anschließend in die Tat umzusetzen. Es geht hierbei allerdings vorzugsweise um das Fundament in der Basis. Besonderheiten werden dort noch nicht angesprochen.

Kapitel 4 legt besagte Besonderheiten dar. Es geht dabei auch, um etwaige Vor- und Nachteile, die sich aus der Nutzung von Ymir ergeben. Des Weiteren geht das Kapitel darauf ein, wie sich Ymir erweitern lässt und stellt somit einen Ausblick auf die Zukunft zur Verfügung. Im fünften und letzten Kapitel geht es nochmal um das Praxisprojekt als Ganzes. Im Fazit enthalten ist eine kritische Reflexion zum Arbeitsablauf des Projekts, zum Projekt selbst und auch konkret zu Ymir.

## *1 Einleitung*

Da bereits im folgenden Kapitel noch einmal näher auf die Leitfrage eingegangen wird, welche mitunter für die Zielsetzung verantwortlich ist, wird die Zielsetzung in diesem Abschnitt nur kurz umrissen. Ziel des Projekts ist es eine Möglichkeit zu erörtern und zu entwickeln, die es ermöglicht Boilerplate-Code automatisch zu generieren, um den Entwicklern damit Arbeit abzunehmen. In dem konkreten Fall dieser Arbeit geht es hierbei um den Webentwicklungsbereich. Andere Bereiche, in denen Boilerplate-Code ebenfalls vorhanden ist, wie etwa in der objektorientierten Programmierung, werden zwar ebenfalls – zumindest im theoretischen Teil – benannt, kommen in der darauffolgenden Umsetzung jedoch nicht vor, da Ymir explizit für den Web-Bereich entwickelt wird.

## **2 Theoretische Ausarbeitung eines Lösungsansatzes**

Bevor das Projekt in die praktische Phase übergehen kann, wird in der Theorie das thematische Umfeld des Projekts beleuchtet. Damit wird aufgezeigt, wo das Projekt steht und welche Zielgruppe am Ende durch das Projektresultat bedient wird.

### **Das Exposé - die Konkretisierung einer Leitfrage**

Während die Leitfrage noch einige Möglichkeiten bietet, daraus Projekte zu entwickeln, ist es die Aufgabe des Exposés einen Lösungsansatz konkret anzugehen. Das Problemfeld wird stärker eingegrenzt und es wird eine erste Idee beschrieben, das Problem zu lösen. Das Exposé zu Ymir ist einfach gehalten. Lukas hat – auch durch seine praktische Arbeit – bereits einiges an Expertise sammeln können, weswegen, viele der im Exposé getroffenen Aussagen, auf seinen Erfahrungswerten beruhen.

Zudem wurde im Exposé herausgestellt, dass sich Ymir vor allem auf den Webentwicklungsbereich konzentriert und daher das Problem vor allem dort angeht. Damit wird der Lösungsansatz nicht nur auf eine Definitionssprache beschränkt, sondern zudem auf einen Entwicklungsbereich. Dabei markiert das Exposé nicht nur den Beginn des Projekts, sondern dient auch dem Zweck einen Betreuer für dieses zu finden. Am Ende des Exposés sind außerdem noch weitere kleinere Leitfragen festgehalten, deren Antwort im Laufe des Projekts erörtert wird.

### **2.1 Betrachtung des thematischen Umfelds**

Im nächsten Schritt wird sich das Themenfeld des Problems mittels Themenfeldcluster angeschaut. Zugegebenermaßen soll dieser Schritt eigentlich vor dem Exposé geschehen. Allerdings ist der Cluster im Rahmen des Seminarteils vom Praxisprojekt entstanden und zu dem Zeitpunkt ist das Exposé bereits fertig und ein Betreuer gefunden gewesen.

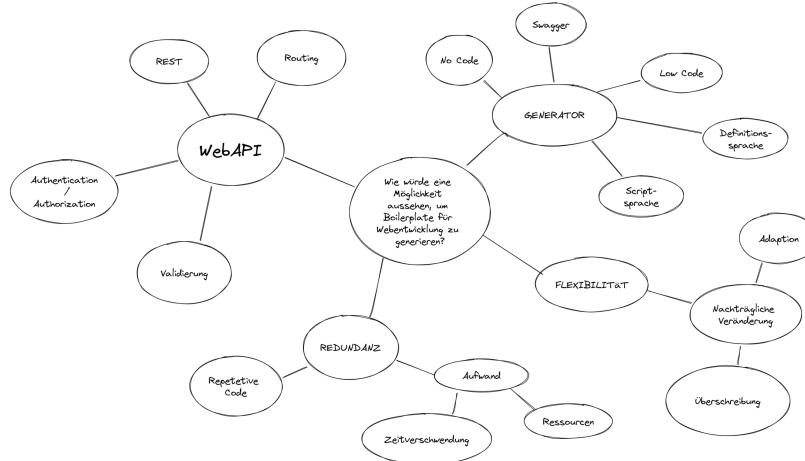


Abbildung 2.1: Themenfeldcluster zur Leitfrage (Hufnagel u. Hufnagel, 2023b)

Aus diesem Grund wird für den Themenfeldcluster 2.1 nochmal einen Schritt zurückgedacht. Statt jedoch einen konkreten Lösungsansatz in die Mitte zu setzen, stellt das Zentrum hier die zu Beginn definierte Leitfrage dar. Die davon abgeleiteten Teilbereiche sind mit „Generatoren“ zum einen konkrete Möglichkeiten das Problem anzugehen, zum anderen sind es auch Bereiche, die das Problem von „Boilerplate-Code“ beschreiben. Hier zusammengefasst unter dem Aspekt „Redundanz“. Da sich das Projekt im Webbereich bewegt, wie die Leitfrage dies auch bereits festlegt, wird unter einem weiteren Punkt verschiedene web-bezogenen Themen dargestellt. Der letzte Teilbereich bezieht sich darauf, was Ymir als möglicher Lösungsansatz erfüllen soll.

### **Themenfeldcluster-Bereich „WebAPI“**

Ein wichtiger Aspekt der heutigen Webentwicklung stellen REST-Anwendungen dar. Aus diesem Grund ist „REST“ Teil des Themenfeldclusters, da eben ein potenzieller Lösungsansatz für die Leitfrage im Webbereich „RESTful Development“ unterstützen sollte. Ein mindestens ebenso wichtiges Thema ist die Cybersicherheit. Daher stehen unter diesem Bereich auch die Punkte Authentifikation, sowie Validierung.

### **Themenfeldcluster-Bereich „Generator“**

Wie bereits etwas weiter oben erwähnt, sind unter diesem Teilbereich konkrete Lösungsansätze gefasst, die die Leitfrage beantworten. Allerdings ist es nicht gesichert, dass Low- oder No-Code Plattformen wirklich diese Leitfrage beantworten wollen. Es ist wahrscheinlich, dass diese Ansätze eher das Coding allgemein vereinfachen wollen und dabei nebenher auch die Leitfrage beantwortet haben. Als solche Plattformen sind zumeist Entwicklungsumgebungen gemeint, bei denen der Anwender wenig oder gar keinen Code schreiben muss, um Programme zu erstellen. Dies bietet den Vorteil, dass das daraus entstehende Programm von einer Person entwickelt und/oder gewartet werden kann, die kein Informatiker ist oder nur wenig Erfahrung in dem Bereich hat (Hauschildt (oD)). Dadurch, dass eben kaum oder gar kein Code geschrieben wird und

dieser automatisch erzeugt wird, wird entsprechender Boilerplate-Code ebenfalls mit erzeugt.

Eine weitere Art von Generator, der im Cluster zu finden ist, ist Swagger, welches in erster Linie zwar zu Dokumentationszwecken entwickelt wird, aber ebenfalls mit Automatischer Erzeugung arbeitet. Änderungen am Code werden automatisiert erkannt und im Code dokumentiert, um so etwa eine spätere Nachverfolgung der Änderungen möglich zu machen (SimplyTest (oD)). Swagger und No/Low-Code-Plattformen wird im späteren Verlauf des Kapitels 2.3 noch einmal aufgegriffen.

Als letzte Mögliche Generatoren-Art, sind im Cluster die Aspekte Skript- und Definitionssprache festgehalten. Unter diesen Punkt fällt auch Ymir, weswegen dieser Abschnitt für das Projekt die größte Relevanz hat. Dabei beziehen sich diese Art von Sprachen nicht konkret auf das Generieren von Boilerplate-Code, sondern ganz allgemein geht es hierbei darum aus einer festgelegten Syntax funktionalen Code zu generieren.

### **Themenfeldcluster-Bereich „Flexibilität“**

Dieser Teilbereich umreißt grob, was ein möglicher Lösungsansatz erfüllen soll. Denn dieser soll möglichst flexibel sein. Genauer geht es bei diesem Begriff darum, dass Änderungen am Code problemlos möglich sein sollen, auch im Nachhinein noch. Und statt den vorhandenen Code bei Änderungen einfach zu überschreiben, was einem Entwickler womöglich mehr Arbeit macht, als einspart, soll der Lösungsansatz so intelligent sein, dass er die Änderungen vornimmt, ohne Bereiche zu überschreiben, die gar nicht überschrieben werden sollen und stattdessen den zugrundeliegenden Code eben nur anpasst. Dies wird im Cluster 2.1 unter dem Begriff *Adaption* gefasst.

### **Themenfeldcluster-Bereich „Redundanz“**

Der verbleibende Bereich beschreibt generell, was ohne „Generatoren“-Möglichkeit meist auf geschriebenen Code zutrifft. Denn wenn Boilerplate-Code händisch geschrieben werden muss, wird dieser schnell redundant. Wie dies eben bei dem Beispiel im Einleitungskapitel der Fall ist: *Getter und Setter* müssen immer wieder - für beispielsweise verschiedene Klassen oder Objekte - geschrieben werden. Dabei sind sie immer gleich aufgebaut und sie machen auch stets das gleiche. Und dennoch sind sie notwendig, auch wenn ihre Auswirkung auch noch so klein ist. Dies kostet Aufwand und dieser Aufwand lässt sich wiederum unterteilen in Ressourcen, die dort hinein investiert werden müssen, sowohl materielle Ressourcen als auch personelle und in Zeit.

Zusammenfassend lässt sich zum Themenfeldcluster 2.1 sagen, dass dieser – auch wenn im Mittelpunkt die Leitfrage steht – nicht jeder Bereich, der darin dargestellt wurde, konkrete Lösungsmöglichkeiten anbietet, sondern einen Überblick darüber geben soll, wie das Themenfeld ungefähr aussieht, ohne dabei direkt in Details zu gehen. Und obwohl der Themenfeldcluster erst nach dem Exposé entstanden ist, stellt sich der Cluster so dar, als könnte daraus das Exposé entstanden sein, welches dann schon konkrete Ansätze umfasst.

## 2.2 Ausarbeitung von Ymir mittels Modellierung

Nachdem sich durch das Exposé bereits eine konkretere Idee heraustraktallisiert hat, wird nun die Idee Ymir mithilfe von Modellierungen ausgearbeitet, um etwaige Umsetzungsmöglichkeiten zu erörtern. Dabei wird unter anderem auf Werkzeuge der UML („Unified Modeling Language“) zurückgegriffen.

### Die Domäne für Ymir

Zuerst wird die Domäne für Ymir mittels Domänenmodell analysiert. Dabei wird dieses Modell in einen IST- und einen SOLL-Zustand unterteilt. Doch hier haben sich schnell Probleme gezeigt.

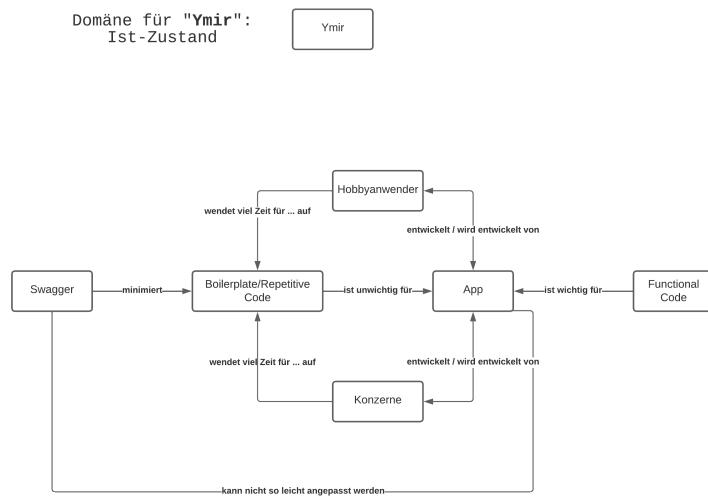


Abbildung 2.2: Domänenmodell im Ist-Zustand (Hufnagel, 2023c)

Das Domänenmodell 5.1 ist in seinem Umfang eingeschränkt. Der Anwendungsfall ist recht speziell und die Zielgruppe entsprechend auch. Da Ymir kein klassisches Programm darstellt mit Frontend und Backend, sondern eine Sprache, lässt sich die Domäne nicht so aufziehen, wie dies bei einem Programm der Fall wäre. Auch sind die involvierten Entitäten zu konkret, um noch mehr Umfang zuzulassen. Eine Verallgemeinerung kommt jedoch ebenfalls nicht infrage, da dies den Anwendungsfall von Ymir verfälschen könnte, da dadurch Entitäten in die Domäne kommen, die in der Realität keinen Bezug zum diesem hätten. So verbleibt das Domänenmodell zu Ymir eingeschränkt und einfach.

## 2 Theoretische Ausarbeitung eines Lösungsansatzes

Domäne für "Ymir":  
SOLL-Zustand

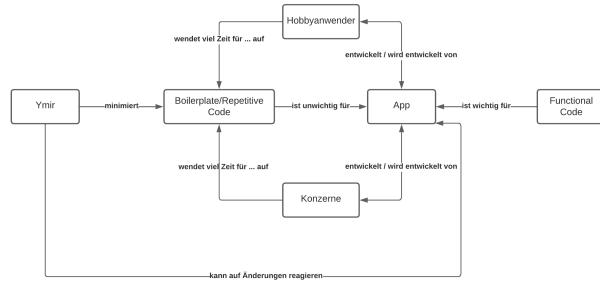


Abbildung 2.3: Domänenmodell im Soll-Zustand (Hufnagel, 2023d)

Im Gegensatz zum *Ist*-Zustand verändert sich in diesem Modell nicht viel. Anstelle von Swagger ist nun Ymir Teil des Modells. Lediglich die Verbindung zwischen Ymir und „App“ verändert sich, da ersteres sich leichter anpassen lässt als Swagger. Ansonsten sind die Modelle von *Ist*- und Soll-Zustand baugleich.

### Der UML-Versuch zu Ymir

Da die Modellierung mittels Domänenmodell sich als nicht sinnvoll herausgestellt hat, ist anschließend noch ein „Use Case“-Diagramm entstanden, um herauszufinden, ob zumindest UML-Diagramme ein solches Projekt ausreichend darstellen können.

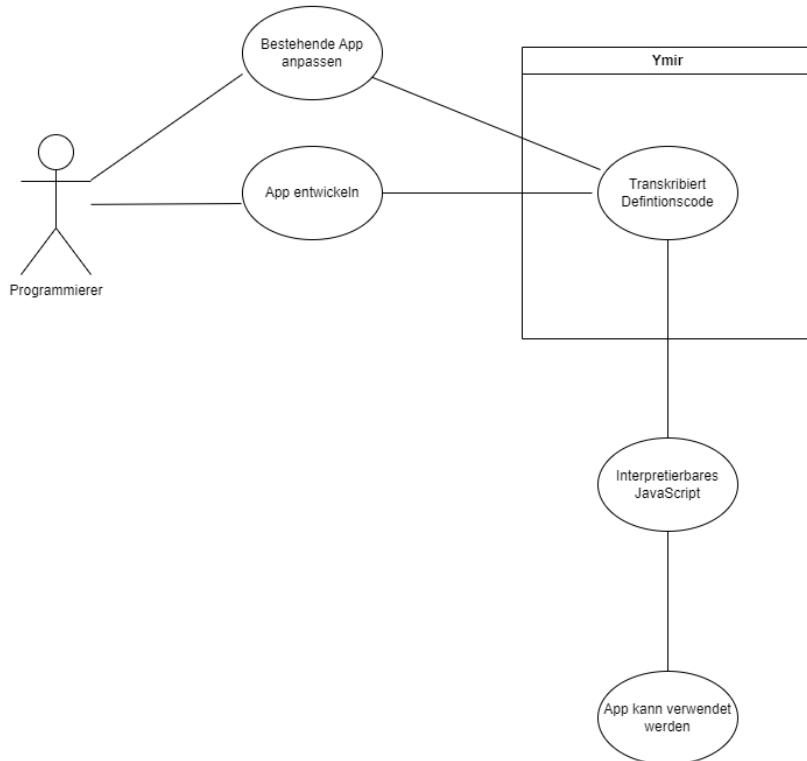


Abbildung 2.4: Ymir als Use Case-Diagramm (Hufnagel, 2023e)

Wie für Use-Case-Diagramme üblich stellt dieses nur einen Fall dar und nicht alle möglichen Fälle. Doch bereits ein Blick auf das Diagramm verrät, dass sich auch mit diesem nicht viel darstellen lässt. Es zeigt sich, was bereits bei dem Domänenmodellen das Problem gewesen ist: Ymir selbst mag komplex sein, der Anwendungsfall ist jedoch bereits konkret. So gibt sich das Use-Case-Diagramm direkt.

Als Akteur ist hier ein Programmierer eingesetzt, welcher mit Ymir eine App entweder von Grund auf entwickeln kann oder eine bestehende App anpassen kann. Der Programmierer verwendet dabei die Ymir-Syntax damit, der Compiler weiß, was er transkribieren muss. Ist der Entwickler mit der Programmierung fertig, sagt er dem Ymir-Compiler, wo dieser die benötigten Dateien finden kann, woraufhin dieser den entsprechenden Code in interpretierbares JavaScript umwandelt, was dann zum Beispiel vom Browser gelesen werden kann oder die App anderweitig verwendbar macht.

Es zeigt sich also, dass der Prozess direkt ist und es somit keinem weiteren Umfang bei dem Diagramm bedarf.

Wie dieses Kapitel zeigt, sind bekannte Modellierungswerkzeuge, wie es die UML eben ist, nicht immer sinnvoll. Natürlich werden diese in erster Linie für die Modellierung von vollwertigen Programmen eingesetzt, aber der Versuch diese auch für Ymir einzusetzen, ist dennoch wichtig, um solche Erkenntnisse eben zu Tage zu fördern. Nachdem auch das Use-Case-Diagramm mager ausgefallen ist, sind weitere Modellierungsmöglichkeiten aus dem Rennen gewesen. Vielleicht hätte sich nochmal ein Modellierungswerkzeug ergeben, dass sich besser geeignet hätte, jedoch wäre dies, wie die

Suche nach der Nadel im Heuhaufen. Ob das Ergebnis dem Aufwand gerecht geworden wäre, ist fraglich.

## 2.3 Mögliche Konkurrenten von Ymir

Bereits recht früh im Projekt, genauer im ersten Gespräch mit dem Betreuer hat sich gezeigt, dass die Kerbe, in der Ymir Platz nehmen soll, kaum bis gar keine Konkurrenten hat. Die Ziele, die Ymir erfüllen soll, sind einzigartig. Der Konkurrent, der Ymir am nächsten kommt ist Swagger. Und Swagger wiederum hat jedoch eine andere Aufgabe zu erfüllen als Ymir. Um jedoch nichts dem Zufall zu überlassen, ist es daher sinnvoll, sich seine Konkurrenz, egal wie ähnlich sie ist, anzuschauen und zu analysieren, wo Gemeinsamkeiten und wo Unterschiede bestehen.

### Konkurrenzanalyse: Swagger

Swagger (SmartBear (oDb)) vereinfacht das Entwickeln von APIs. Dabei muss der meiste Code nicht mehr selbst geschrieben werden, sondern kann etwa mittels *Swagger Inspector* (SmartBear (oDa)) zusammengesetzt werden. Dabei werden Parameter, Werte, etc. vom Nutzer in eine Browser-Maske eingegeben und Swagger generiert daraus den entsprechenden Code und dokumentiert diesen auch. Ferner gibt es auch Plugins, die eine Integration in bekannte Entwicklungsumgebungen, wie die von JetBrains (JetBrains (oD)), ermöglichen. Dadurch ergibt sich auch bereits eine Gemeinsamkeit zu Ymir, da sowohl Ymir als auch Swagger Zeitersparnisse mittels Code-Generierung erzielen. Der Knackpunkt an Swagger ist jedoch, dass die Generierung von Code gar nicht im Mittelpunkt steht. In erster Linie wird Swagger dazu verwendet APIs ordentlich zu dokumentieren. Denn einerseits ist die Dokumentation ein wichtiger Bestandteil der Entwicklung, andererseits jedoch auch zeitaufwendig. Hier greift Swagger ein. Es arbeitet direkt im System mit und kann so Änderungen automatisch dokumentieren. Solche Änderungen werden dabei direkt am Code dokumentiert (IONOS (2020)).

Diese Dokumentation ist sowohl von Menschen als auch von Maschinen lesbar. Aber damit ergibt sich auch schon der große Unterschied zu Ymir: Ymir versucht Boilerplate-Code zu reduzieren. Es geht nicht darum zu dokumentieren. Diejenigen, die Ymir verwenden sind in der Regel Entwickler, die ein ausreichendes Verständnis von ihrer eigenen Anwendung haben, denn Ymir erwartet konkrete Anweisungen, aus denen es den Code generieren kann. Deswegen ist Swagger eben nicht als direktes Konkurrenzprodukt zu Ymir zu sehen.

Außerdem ist Swagger nicht völlig nachteillos. So ergibt sich ein Nachteil von diesem aus einem Vorteil von Ymir: Die Code-Generierung (Barragan (2017)). Wer Tools von Swagger, wie den Inspector nutzt, generiert dabei womöglich Code, den er gar nicht generiert haben möchte. Wird dabei bereits in einer vorhandenen Anwendung gearbeitet, kann es passieren, dass bestehender Code überschrieben wird. Zwar werden Änderungen konsequent dokumentiert, können also zurückverfolgt und gegebenenfalls behoben werden, jedoch kostet das Zeit, wenn es wirklich mal zu einem solchen Problem kommt. Ein weiteres Problem an der Generierung ist, dass Swagger Typen generiert, die beispielsweise in JSON, gar nicht existieren. So ist Swagger etwa in der Lage *enums* zu generieren. In JSON gibt es jedoch keine *enums*. Diese Ungereimtheiten können im

Problemfall dazu führen, dass der Code danach nicht mehr funktioniert. Swagger hat noch weitere Probleme bzw. Nachteile, die sich hier benennen ließen. Dies tut hier aber nichts zur Sache und da bereits feststeht, dass Swagger nicht als direkte Konkurrenz zu Ymir betrachtet werden kann, ist eine weitere Aufzählungen nicht notwendig.

### **Alternativer Konkurrent mit anderem Fokus**

In diesem Abschnitt geht es nochmal um andere mögliche Konkurrenzprodukte, die ihren Fokus ganz woanders haben. Als Beispiel behandelt dieser Abschnitt Oracle's Low-Code-Plattform APEX.

Oracle APEX steht für „Oracle Application Express“ und stellt eine Low-Code-Plattform-Lösung für den Enterprise-Bereich dar (Oracle (oD)). Wie für Low-Code-Plattformen üblich ist für die Erstellung solcher Anwendung nur kaum Vorwissen im Bereich der Programmierung notwendig. Wie für Oracle typisch beschäftigt sich APEX vor allem mit dem Datenbank-Bereich und bietet dort eben die Möglichkeit entsprechende Applikationen zu erstellen. So können Anwender über ein Web-Tool auf den *App Builder* zugreifen und mithilfe von diesem Anwendungen erstellen. Da sich hierbei um Low-Code und nicht No-Code handelt, ist dennoch etwas Expertise vonnöten, um Anwendungen damit zu erstellen. So sollte beispielsweise Wissen zu SQL vorhanden sein, um die nötigen Verbindungen zu verstehen und erstellen zu können, damit die gebauten Applikation mit den entsprechenden Daten arbeiten lassen zu können. Der Code, der für solche Anwendungen normalerweise zeitaufwendig geschrieben werden muss, wird automatisch durch APEX generiert. Der Nutzer muss nur kleinere Anpassungen und das Festlegen der Werte und Parameter selbst übernehmen. Wie bereits erwähnt ist dies ohne etwas Vorwissen nicht so einfach möglich, aber dennoch übernimmt APEX einiges an Arbeit.

Aber wie bereits weiter oben erwähnt, stellt APEX keinen direkten Konkurrenten dar. Durch seine Art liegt der Fokus schlachtweg ganz woanders. Es geht nicht darum einen Entwickler in der Programmierung der eigenen Apps zu unterstützen. Es geht darum ihm die Arbeit ganz abzunehmen. Damit fällt APEX genauso wie Ymir in die Kategorie der „Code-Generatoren“, geht allerdings mit einer anderen Zielsetzung an die Sache und funktioniert grundlegend anders.

Sicherlich könnte eine noch tiefere Analyse und das Begutachten weiterer Konkurrenzprodukte neue Erkenntnisse aufzeigen, da sich doch bereits im Gespräch mit dem Projektbetreuer gezeigt hat, dass Swagger Ymir am nächsten kommt und diese beiden trotz ihrer Ähnlichkeiten andere Ziele verfolgen, lässt sich zusammenfassend sagen, dass Ymir einen Bereich bedient, in dem es ziemlich einzigartig ist.

### **Zusammenfassung zu dem Theorieteil**

Auch wenn die Ergebnisse der theoretischen Ausarbeitung eher überschaubar sind, so sind die Erkenntnisse für die weitere Planung dennoch von Relevanz. Die Konkurrenzanalyse 2.3 etwa zeigt, dass es zwar durchaus Mitbewerber gibt, diese jedoch nicht in dieselbe Kerbe schlagen, wie Ymir dies tut. Eine Ausarbeitung in Sachen Modellierung 2.2 ergibt bei dieser Art von Projekt jedoch wenig Sinn. Ymir stellt schlachtweg

## *2 Theoretische Ausarbeitung eines Lösungsansatzes*

kein klassisches Programm dar und Werkzeuge, wie die UML existieren in erster Linie für die Modellierung vollwertiger Programme. Mithilfe vom Themenfeldcluster 2.1 lässt sich Ymir in das „große Ganze“ einordnen und zeigt dadurch bereits ein paar Lösungsansätze für die Leitfrage auf, die nicht zwangsläufig auf einen „Generator“ wie Ymir hindeuten. Auch untermauert der Cluster, was ein möglicher Lösungsansatz können muss, um überhaupt Sinn zu ergeben und welche Ziele dieser verfolgt. Alles in allem hilft der Cluster dabei einen Überblick über das gesamte Themenfeld zu geben, um somit die Problematik besser begreifen zu können.

Schlussendlich ist das Exposé der Ausgangspunkt des gesamten Projekts. Es spiegelt die Vision von Ymir wider, die mit dem Projekt in Erfüllung gehen soll und hilft es Ausstehenden den Rahmen des Projekts verstehen zu können, ohne zu tief in die Materie eindringen zu müssen.

# 3 Entwurf und Umsetzung eines Boilerplate-Generators

Nachdem nun die theoretische Vorarbeit abgeschlossen ist, kann das Projekt in die Hauptphase übergehen, in der Ymir konkret ausgearbeitet und dann auch umgesetzt wird. In diesem Kapitel steht dabei der Entwurf von Ymir, sowie dessen Umsetzung im Vordergrund. Dabei wird hier noch nicht auf Besonderheiten davon eingegangen.

## 3.1 Die Idee zu Ymir

In diesem Unterkapitel wird die Idee zu Ymir behandelt. Das Kapitel betrachtet quasi einen ersten Entwurf, mit einigen Konzepten und Ideen rund um Ymir, die es zur Umsetzung geschafft haben. Im Kapitel 3.3 wird dann noch eine Idee beschrieben, die wiederum verworfen wurde.

Zunächst sollten die Leitfragen wieder eine Rolle spielen. Bevor also die Praxis angegangen wird, sollte der erste Schritt des Entwurfs eine Auseinandersetzung mit etwaiigen Leitfragen sein. Da das Hauptproblem des Projekts der Boilerplate-Code darstellt, ist dies der erste Anhaltspunkt gewesen. Im Zuge dessen gilt es die Frage zu beantworten: „Wie kompakt und nicht repetitiv kann der Code sein?“. Diese Frage setzt sich aus zwei Bausteinen zusammen, die eben Boilerplate-Code ausmachen: Zum einen der sich ständig wiederholende Code und zum anderen – je nach Sprache – die Masse an Code, der für die Lauffähigkeit des Programms benötigt wird, obwohl kaum Mehrwert dadurch entsteht.

Die Kompaktheit auf ein Minimum zu reduzieren, ist jedoch ebenfalls keine valide Option. Denn schlussendlich soll jede beteiligte Partei in der Lage sein, den Code auch lesen und verstehen zu können. Aus diesem Grund muss das Mittelmaß zwischen den beiden Polen *Kompaktheit* und *Lesbarkeit* gefunden werden. Inwiefern dieses Ziel gelungen ist, lässt sich den folgenden Unterkapiteln entnehmen. Gleichermaßen gilt natürlich auch für den zweiten Aspekt der Leitfrage. Auch hier spielt das Hauptproblem von Boilerplate-Code mit herein. Beim Schreiben des Ymir-Codes sollte der Entwickler sich nicht ständig wiederholen müssen, um die notwendigen Features unterzubringen. Denn wenn der Entwickler die Anzahl an Code-Abschnitt konkret definieren müsste, würde wieder Arbeitsaufwand für den Entwickler entstehen und Ymir versucht genau diesen zu reduzieren.

Damit geht eine weitere Frage einher, welche den Entwurf begleitet hat, und zwar: „Wie kann Zeit eingespart werden?“. Besonders in der freien Marktwirtschaft gilt häufig der Grundsatz „Zeit ist Geld“. Und da Ymir die Zeitproblematik angeht, ist diese Frage für das Projekt relevant. Um Zeit einzusparen sind Keywords Teil von

Ymir. So können etwa mithilfe eines Wortes http-Requests definiert werden. Es ist zudem mögliche eine solche Request spezifischer zu definieren, falls dies nötig ist. Dabei bleibt die Syntax jedoch einfach und verständlich. Die Antwort auf die Zeitersparnis-Frage ist somit eine möglichst einfache Syntax, die es mit wenigen Wörtern erlaubt den benötigten Code zu generieren.

Mit der Antwort auf die zweite Frage wird zudem ersichtlich, dass die Leitfragen zumindest teilweise miteinander zusammenhängen: Einerseits wird der Code möglichst kompakt und nicht repetitiv gestaltet und andererseits geht damit bereits eine Zeiter-sparnis einher. Nachdem sich bereits erste Antworten auf etwaige Leitfragen ergeben haben, ist der nächste Schritt dies in einem Entwurf zu erfassen.

Für Softwareentwicklung typisch ist nicht sofort das fertige Produkt geplant worden. Stattdessen sollte ein Prototyp als erster praktischer Schritt die Funktionalität von Ymir demonstrieren. Viele der finalen Features enthält dieser noch gar nicht und bezieht sich dabei vielmehr auf die Basis-Funktionalitäten. Das Ziel, welches auf dem Papier festgelegt wurde, ist ein einfacher Prototyp, der die Syntax versteht und umwandeln kann. Dabei umfasst dieser die Parsing- und die Kompilierungskomponente, wodurch das Parsen und Kompilieren der festgelegten Syntax möglich ist. Ebenfalls im Prototyp enthalten sind Standardfeatures des Web-Bereichs wie *Router*, *Routen*, *Validation*, sowie *Authorization* – also ob der Benutzer entsprechende Rechte hat, um bestimmte Dinge zu tun – und *Authentication* – ob der Benutzer überhaupt eingeloggt ist, um bestimmte Dinge zu tun. Beispielhaft angewendet wird dies beim Prototypen mit der Sprache JavaScript, welche aufgrund ihrer Position im Web-Bereich als unabdingbar bezeichnet werden kann und deswegen ein repräsentatives Beispiel darstellt. Damit steht fest, was der Prototyp kann – zumindest auf dem Papier, doch bevor dieser umgesetzt wird – ebenso wie die finale Version, sind weitere Schritte erfolgt, um präziser an Ymir arbeiten zu können.

## 3.2 Die Ymir-Qualitätsanforderungen

Unter diesem Punkt werden Anforderungen gefasst, die Ymir erfüllen soll. Das können auch Aspekte sein, die Ymir von vergleichbaren Produkten abheben. Diese Anforderungen ergeben sich aus der Idee zu Ymir und sind bei der weiteren Entwicklung von Relevanz, da sie den Prozess bis hin zur Fertigstellung leiten und lenken. Sie bringen die Ideen, die im ersten Unterkapitel stehen, auf einen gemeinsamen Nenner, der über Ymir im Zentrum mit den Anforderungen verbunden ist. Um dies besser darstellen zu können, sind die Qualitätsanforderungen in einem Modell zusammengebracht worden.

Der folgende Abschnitt beschäftigt sich nun mit den einzelnen Anforderungen, die in der Grafik 3.1 zu finden sind.

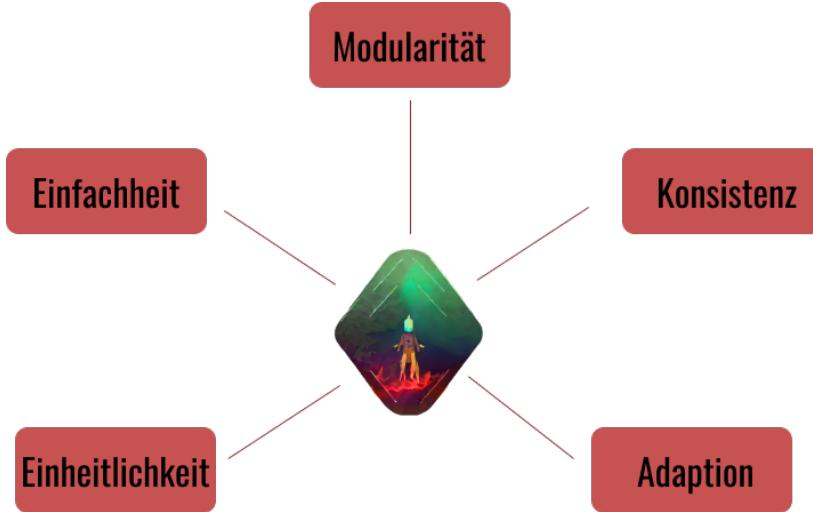


Abbildung 3.1: Die Ymir-Qualitätsanforderungen (Hufnagel, 2023f)

**Modular** Der erste Aspekt, den Ymir umfasst, ist die Modularität. Unter diesen Punkt fällt etwa die Erweiterbarkeit beziehungsweise Flexibilität, die von Anfang an bei Ymir vorgesehen ist. Es soll stets möglich sein Ymir zu erweitern oder verändern ohne dabei den Code der gesamten Sprache umschreiben oder anpassen zu müssen. So soll es möglich sein Ymir neben JavaScript neue Sprachen beizubringen und das mit möglichst wenig Aufwand, sowie möglichst geringen Hürden.

**Einfach** Besonders der letzte Satz des vorherigen Absatzes hat diesen Aspekt bereits etwas angerissen. Ymir soll einfach zu verstehen und zu erlernen sein. Konkret bezieht sich dies natürlich auf Webentwickler, kann sich aber auch auf generell auf Programmierer beziehen, wobei Ymir speziell im Web-Bereich verwendet wird, da es dafür entwickelt wird. Trotz der Einfachheit ist Vorwissen über Programmierung notwendig, um Ymir verstehen und damit umgehen zu können.

**Einheitlich** Sowohl Ymir selbst als auch der von Ymir generierte Code setzt Standards um. Das meint etwa etablierte Webstandards oder Naming Conventions. Des Weiteren setzt Ymir in der eigenen Syntax auf Standards, so dass der Code, der für Ymir geschrieben wird – etwa Code, den Ymir umwandeln soll, immer einheitlich aussieht, egal, welche Ausgangssprache verwendet wird.

**Konsistent** Hiermit wird beschrieben, wie Ymir arbeitet. Die Prozesse, die Ymir ausführt sollen stets nachvollziehbar und verständlich sein. Der Zusammenhang zwischen dem Eingangscode und dem daraus generierten Code ist immer vorhanden und wirft keine Fragen auf. Beim Umwandeln soll nicht das Gefühl auftreten, dass irgendwelche „Magie“ zum Einsatz kommt. Resultate sollen klar sein und unabhängig von Hardware oder Software sein.

**Adaptieren nicht überschreiben** Dieser Grundsatz ist für Ymir elementar. Die Adoption beschreibt die Fähigkeit von Ymir sich an bereits bestehenden Code anzupassen und Generierungen so vorzunehmen, dass es nicht zu Überschreibungen kommt. Damit ist diese Anforderung vor allem an die Projekte gerichtet, bei denen Ymir eingesetzt wird, wo bereits eine Code-Basis vorhanden ist, und kein neues Projekt entsteht. Dieser Grundsatz ist auch deswegen so wichtig, da er das Ziel der Zeitersparnis garantiert. Würde Ymir bestehenden Code überschreiben, würde dies unter Umständen für mehr Arbeit bei den Entwicklern sorgen, da dann nachvollzogen werden müsste, welchen Code Ymir überschreibt und wie die Nutzung von Ymir dann in bereits bestehende Abläufe integriert werden kann. Zusammenfassend sagt dieser Aspekt aus, dass Ymir bestehenden Code nicht überschreibt, sondern sich diesem anpasst und Generierungen basierend auf dem bestehenden Code vornimmt.

Mit diesen Anforderungen gibt es nun quasi Richtlinien, die Ymir beschreiben und die von Ymir erfüllt werden. Als nächstes wird die definierte Semantik und Syntax von Ymir beleuchtet und erklärt, um im Anschluss daran zum praktischen Teil übergehen zu können.

### 3.3 Ymirs Semantik und Syntax

Im Folgenden werden Semantik und Syntax von Ymir dargelegt. Es wird auch aufgezeigt, warum die entsprechenden Strukturen genutzt werden.

In Sachen Semantik und Syntax erfindet Ymir das Rad nicht neu. Stattdessen sind die Konzepte so gewählt, dass insbesondere Webentwickler sich schnell und einfach in Ymir einarbeiten können. Mit einem Blick auf ein Ymir-Skript fällt auf, dass auch der Aufbau der Dateien so ziemlich dem entspricht, was Webentwickler bereits von ihren eigenen Apps kennen. Der genaue Aufbau solcher Dateien wird in diesem Unterkapitel noch nicht beleuchtet, sondern erst im Folgekapitel – also Unterkapitel 3.4. Durch die Generierung von Objekten wird die grundsätzliche Semantik gepasst. Was das genau bedeutet, wird im nächsten Unterkapitel 3.4 näher erläutert. Ein detaillierte Erklärung an dieser Stelle ist nicht sinnvoll, da dies technisch zu erklären ist und das folgende Unterkapitel sowieso auf die Technik eingeht.

Obwohl bereits in dieser Dokumentation im Zusammenhang mit Ymir der Begriff „Compiler“ häufiger verwendet wird, ist dies leicht misszuverstehen, was der Compiler ist und was nicht. Denn zu sagen Ymir hätte einen Compiler mag zwar irgendwie stimmen, ist allerdings nicht richtig. Eigentlich ist das große Ganze der Compiler, also das, was in der Dokumentation als Projekt bezeichnet wird. Ymir ist sowohl die Sprache, die zum Generieren von Boilerplate-Code in Ymir-Skripts verwendet wird, als auch der Compiler, der die Skripts in zum Beispiel verwendbaren JavaScript-Code umwandelt. Das „Herz“ des Compilers sind dabei die Plugins, mit denen Ymir erweitert werden kann, also womit Ymir zum Beispiel neue Sprachen beigebracht werden kann. Natürlich setzt Ymir dabei auf eine einheitliche Semantik, doch die einzelnen Targets können Sonderregeln mit ins Spiel bringen. Das liegt auch daran, dass nicht jede Sprache gleich funktioniert und aufgebaut ist. Daher ist das Festhalten an einer

strikten Semantik nicht nur nicht sinnvoll, sondern kann womöglich auch gar nicht konsequent umgesetzt werden, da dies sonst zu Problemen führen kann.

Syntaktisch hat Ymir ein paar Iterationen durchlebt. Zunächst ist das Ziel gewesen eine Variante umzusetzen, bei der Router zwar benannt werden können, aber nicht explizit als solche deklariert werden. In der aktuellen Syntax werden Router mittels gleichnamigem Keyword aufgerufen, dann wo diese hinführen und dann der Inhalt, wie etwa eine GET-Request. Dies sieht dann etwa so aus:

```
router /another-router
{
    GET /test;
}
```

In der Variante, die oben als diejenige beschrieben wird, die zunächst umgesetzt werden sollte, wäre das „Router“-Keyword weggefallen. Derselbe Code-Schnipsel hätte dann stattdessen so ausgesehen:

```
another-router
{
    GET /test;
}
```

Darunter hätte jedoch die Klarheit gelitten. Während in dem Demo-Schnipsel *another-router* noch eindeutig auf einen Router verweist, wäre dies in der Realität häufig vermutlich nicht der Fall. Der Entwickler, der das Skript für Ymir schreibt, hätte den Inhalt also noch verstanden, aber Ausstehende hätten mit dem Verständnis möglicherweise Probleme gehabt. Aus diesem Grund wurde diese Iteration fallen gelassen und ein Keyword eingeführt, dass die Eindeutigkeit fördert.

Der nun folgende Abschnitt beschäftigt sich mit Ymir-eigene Vokabeln, die von Ymir als Keywords verstanden werden und damit besondere Funktionalitäten übernehmen.

**target** Dieses Keyword definiert die Zielsprache, also welche Sprache der generierte Code zugehörig sein soll. „target JavaScript“ würde dafür sorgen, dass aus dem Skript JavaScript-Code generiert wird.

**use** Das Wort *use* deklariert Module oder Middleware. So kann zum Beispiel Json eingebunden werden oder ein CORS-Modul aufgerufen werden.

**auth** Damit werden Authentication- oder Authorization-Funktionalitäten deklariert. Dabei können auch Details festgelegt werden, wie etwa die Quelle oder das Standard-Access-Level.

**router** Die Erklärung zu diesem ist bereits im vorherigen Abschnitt zu finden. Der Vollständigkeit halber, taucht es dennoch nochmal hier auf. Hiermit werden Router aufgerufen. Es kann festgelegt werden, über welche Route der Router abgerufen werden kann und anschließend wird der Inhalt benannt.

**include** Hiermit wird eine Präprozessor-Anweisung aufgerufen. Damit kann an der Stelle, wo das *include* steht eine Datei eingebunden werden. Wie genau dies vonstatten geht, wird ebenfalls im Unterkapitel 3.4 erläutert, da dieser Aspekt technisch fokussiert ist und das nächste Unterkapitel auf die Technik eingehen wird.

Der letzte Abschnitt dieses Unterkapitels sollte eher als Exkurs beziehungsweise Abstecher verstanden werden. Das folgende Thema ist aus reinem Interesse entstanden und wird im kommenden Verlauf des Projekts nicht weiterverfolgt.

#### Exkurs: Esoterische Syntax für Ymir

Bei esoterischen Programmiersprachen handelt es sich um Sprachen, die weder ernst gemeint sind noch einen produktiven Zweck erfüllen sollen beziehungsweise wollen (Lang u. Augsten (2017)). Meist geht es nur darum irgendetwas zu beweisen, zum Beispiel eine Turing-vollständige Sprache mit dem kleinstmöglichen Interpreter auf die Beine zu stellen, wie es das Ziel der esoterischen Sprache „Brainfuck“ ist (Kuhlemann (oD)). Oder es geht nur darum einfach Spaß zu haben und eine verrückte Sprache zu entwickeln.

Die Syntax esoterischer Sprachen ist meist nur für den Computer lesbar, selbst erfahrene Programmierer können mit solchem Code kaum etwas anfangen. Bei der Entwicklung der Ymir-Syntax ist, als die finale Iteration erreicht wurde, auch eine esoterische Version der Syntax entstanden. Über den Entwurf hinaus ist diese Variante nie gekommen. Die Syntax soll lediglich zeigen, dass eine esoterische Version in der Theorie möglich ist und wie dies aussehen würde. Dabei ist dieser Ansatz entstanden:

```
>JS;#env;c:(o:@env.ALLOWED_ORIGIN);r:/api;ah:(X-API-Key:s);[  
    g:/hello?name=s;a:HelloRoute;  
    p:/person;a>CreatePerson;b:(name:s,age:i);  
];;"include.ymr"
```

Ganz so unlesbar, wie etwa „Brainfuck“ ist dieser esoterische Code nicht. Damit zu arbeiten wäre jedoch zumindest eine Geduldsprobe. Des Weiteren würde Ymir damit das Ziel der Zeitersparnis - ein Ziel, welches in dieser Doku bereits häufiger genannt wurde – vermutlich nicht erfüllen können.

Wie bereits eingangs erwähnt, ist dieser Entwurf alles, was entstanden ist. Möglich wäre Ymir als esoterische Sprache durchaus. Nur dann würde sich die Zielsetzung grundlegend ändern. Mit diesem kleinen Abstecher zu den esoterischen Sprachen geht es nun weiter mit dem praktischen Teil der Ymir-Entwicklung. In den letzten 3 Unterkapiteln ist alles rund um die Theorie von Ymir behandelt worden. Nachdem nun auch die Semantik, sowie die Syntax feststeht, kann mit der praktischen Umsetzung begonnen werden.

## 3.4 Die praktische Umsetzung von Ymir

Da nun Semantik und Syntax feststehen, kann die Umsetzung von Ymir beginnen. Dieser Abschnitt beschreibt Ymir Entstehung vom Prototyp bis hin zu den fertigen Hauptbausteinen, wie Compiler oder Parser. Direkt am Anfang muss für das Verständnis klargestellt werden, was die Differenzen zwischen Ymir und *YmirScript* sind. Als YmirScript wird der Name der eigens entwickelten Definitionssprache gesehen. Ymir wiederum ist der Name des Programms, des Compilers, der YmirScript Source-Code nimmt und in die gewünschte Zielsprache umwandelt.

### 3.4.1 Der Compiler

Der Compiler für YmirScript besteht aus mehreren wesentlichen Komponenten, die zusammenarbeiten, um den Quellcode in eine Zielsprache oder eine ausführbare Form zu transformieren. Die folgenden Abschnitte befassen sich mit den einzelnen Bausteinen des YmirScript-Compilers und geben einen umfassenden Überblick über die Prozesse bei der Übersetzung der Sprache. In den folgenden Abschnitten wird auf Code-Blocks verzichtet, da sie entweder zu klein sind und somit völlig den Kontext verlieren und schwer zu verstehen sind oder viel zu groß sind und den Rahmen dieser Dokumentation sprengen würden. Ein paralleler Blick in das Projekt-eigene Repository beim Lesen dieser Dokumentation ist daher womöglich sinnvoller (Hufnagel u. Hufnagel (2023c)).

#### Lexikalische Analyse

Der erste Schritt im Kompilierungsprozess ist die lexikalische Analyse. In dieser entscheidenden Phase wird der eingegebene Quellcode in eine Reihe von Token umgewandelt, die es dem Compiler ermöglichen, die Sprachkonstrukte besser zu verstehen und zu verarbeiten. Diese Token stellen die kleinsten sinnvollen Einheiten der Sprache dar, darunter Schlüsselwörter, Bezeichner, Operatoren und Literale. Die lexikalische Analyse wird von einer speziellen Komponente durchgeführt, die als Lexer oder Scanner bezeichnet wird. Der Lexer liest den Quellcode Zeichen für Zeichen, identifiziert Muster, die den vordefinierten Syntaxregeln der Sprache entsprechen, und erzeugt die entsprechenden Token. Beim Scannen des Quellcodes verwirft der Lexer Leerzeichen und den Zeilenumbruch („Backslash-n+Backslash+r“), um sicherzustellen, dass nur sinnvolle Informationen an die nachfolgenden Stufen des Kompilierungsprozesses weitergegeben werden.

Während des Tokenisierungsprozesses zeichnet der Lexer auch die genauen Zeilen- und Spalteninformationen für jedes Token auf. Diese Informationen sind von unschätzbarem Wert, wenn in späteren Phasen des Kompilierungsprozesses Fehler entdeckt werden, da sie es dem Compiler ermöglichen, den genauen Ort des Problems zu lokalisieren und dem Entwickler ein genaues Diagnose-Feedback zu geben.

In YmirScript sind Kommentare nicht erlaubt, was den lexikalischen Analyseprozess noch weiter vereinfacht. Das Fehlen von Kommentaren bedeutet, dass sich der Lexer ausschließlich auf die wesentlichen Elemente des Quellcodes konzentrieren kann, was ihn effizienter und einfacher macht. Zusammenfassend lässt sich sagen, dass die lexikalische Analysephase ein wesentlicher Bestandteil des Kompilierungsprozesses ist, der

für die Umwandlung des eingegebenen Quellcodes in eine Reihe von Token verantwortlich ist.

Durch die Erzeugung dieser Token und die Aufzeichnung ihrer Zeilen- und Spalteninformationen vereinfacht der Lexer den Quellcode und ermöglicht es dem Compiler, die Sprachkonstrukte in den nachfolgenden Phasen leichter zu verarbeiten und zu analysieren.

#### Syntaxanalyse mit knotenbasierter Struktur

Nach Abschluss der lexikalischen Analyse geht der YmirScript-Compiler zur Phase der Syntaxanalyse über, die gemeinhin als Parsing bezeichnet wird. In dieser kritischen Phase geht es darum, die vom Lexer erzeugte Abfolge von Token zu interpretieren und in eine kohärente hierarchische Struktur zu bringen. Anstelle eines traditionellen abstrakten Syntaxbaums (AST) entscheidet sich YmirScript für eine knotenbasierte Struktur, die mehrere Vorteile in Bezug auf Flexibilität und Modularität bietet.

Im YmirScript-Compiler sind die Knoten JavaScript-Objekte, die die Sprachkonstrukte verkörpern, wie Anweisungen, Ausdrücke und Deklarationen. Jeder Knoten enthält relevante Informationen über das entsprechende Sprachkonstrukt, einschließlich seines Typs, der Zeilen- und Spalteninformationen und aller zugehörigen Kindknoten. Durch den Aufbau der gesamten Struktur mit untergeordneten Knoten erstellt der YmirScript-Compiler eine klare, hierarchische Darstellung des Quellcodes, die die syntaktischen Beziehungen zwischen verschiedenen Elementen beibehält.

Die Verwendung einer knotenbasierten Struktur in YmirScript bietet mehrere Vorteile. Erstens ermöglicht sie eine flexiblere und modulare Architektur, die es einfacher macht, den Compiler an neue Sprachfunktionen oder Änderungen anzupassen. Zweitens, da die Knoten JavaScript-Objekte sind, können sie leicht manipuliert und verarbeitet werden, was die nachfolgenden Phasen des Kompilierungsprozesses vereinfacht. Drittens erleichtert die knotenbasierte Struktur die Entwicklung von Ziel-Plugins, da sie die Knoten für ihre spezifischen Typsysteme rekonstruieren können, ohne die exakt gleichen Knotenklassen replizieren zu müssen. Dieser Ansatz rationalisiert den Prozess der Erweiterung von YmirScripts Fähigkeiten auf zusätzliche Sprachen.

Während der Syntaxanalyse ist der Parser auch für die Erkennung und Meldung von Syntaxfehlern im Quellcode verantwortlich. Anstatt beim ersten gefundenen Fehler aufzuhören, parst der YmirScript-Parser den Quellcode weiter und versucht, alle Fehler in einem einzigen Durchgang zu identifizieren. Dieser Ansatz ermöglicht es den Entwicklern, mehrere Probleme auf einmal anzugehen, was die Gesamteffizienz des Fehlerbehebungsprozesses verbessert.

Insgesamt spielt die Phase der Syntaxanalyse im YmirScript-Compiler eine wichtige Rolle bei der Übersetzung der Tokenfolge in eine knotenbasierte Struktur, die die syntaktischen Beziehungen des Quellcodes genau darstellt. Dieser einzigartige Ansatz verleiht YmirScript eine flexible und modulare Architektur, die eine nahtlose Anpassung an sich entwickelnde Sprachmerkmale und -anforderungen sowie eine optimierte Integration mit verschiedenen Zielsprachen durch Plugins ermöglicht.

### **Integrierte semantische Analyse in YmirScript**

In YmirScript ist die semantische Analyse nahtlos in den gesamten Kompilierungsprozess integriert, anstatt als separater zusätzlicher Schritt behandelt zu werden. Dieser integrierte Ansatz kombiniert die semantische Validierung effektiv mit der Syntaxanalyse und der Codegenerierung, was zu einer schlankeren und effizienteren Handhabung der semantischen Einschränkungen des Quellcodes führt.

In der Phase der Syntaxanalyse führt der YmirScript-Parser neben dem Parsing-Prozess eine Reihe von semantischen Prüfungen durch. Diese Prüfungen stellen die Korrektheit des Quellcodes sicher, indem verschiedene Aspekte wie Routerdeklaration, Routensignatur und Typkonsistenz validiert werden. Durch die Integration dieser semantischen Prüfungen in die Parsing-Phase kann YmirScript potenzielle Probleme frühzeitig im Kompilierungsprozess erkennen und melden, wodurch die Ausbreitung von Fehlern auf nachfolgende Phasen verhindert, und eine reibungslose Entwicklung ermöglicht wird.

Darüber hinaus erstreckt sich der integrierte semantische Analyseansatz auch auf die Phase der Codegenerierung, in der Ziel-Plugins eine entscheidende Rolle spielen. Da diese Plugins Code für bestimmte Zielsprachen generieren, behandeln sie auch alle sprachspezifischen semantischen Regeln und Validierungen. Dieser Ansatz ermöglicht es dem YmirScript-Compiler, sich an die einzigartigen Anforderungen jeder Zielsprache anzupassen und gleichzeitig ein hohes Maß an Codequalität und -konsistenz beizubehalten.

Die integrierte semantische Analyse in YmirScript bietet mehrere Vorteile. In erster Linie rationalisiert sie den Kompilierungsprozess, indem sie semantische Validierungen mit der Syntaxanalyse und der Codegenerierung kombiniert und so den Bedarf an zusätzlichen Verarbeitungsschritten reduziert. Zweitens ermöglicht er eine effizientere und genauere Handhabung semantischer Einschränkungen und stellt sicher, dass der generierte Code den Anforderungen der Zielsprache und den Best Practices entspricht. Schließlich unterstützt dieser Ansatz die modulare und erweiterbare Natur von YmirScript, die eine nahtlose Integration mit verschiedenen Zielsprachen durch die Verwendung von Zielplugins ermöglicht.

Die integrierte semantische Analyse in YmirScript ist eine Schlüsselkomponente des Kompilierungsprozesses und trägt zu einem schlankeren und effizienteren Umgang mit semantischen Einschränkungen bei, während die Flexibilität und Anpassungsfähigkeit für eine breite Palette von Zielsprachen erhalten bleibt.

### **Code-Generierung mit Ziel-Plugins in YmirScript**

Der Höhepunkt des YmirScript-Kompilierungsprozesses ist die Codegenerierungsphase, in der der Compiler die knotenbasierte Struktur in die gewünschte Zielsprache oder ein ausführbares Format übersetzt. Diese entscheidende Phase wird von Target-Plugins angetrieben, die als treibende Kraft hinter YmirScripts Kernanforderungen der Modularität, Flexibilität und Erweiterbarkeit dienen.

Target-Plugins in YmirScript sind für die Konvertierung der durch die knotenbasierte Struktur dargestellten Sprachkonstrukte in die entsprechende Syntax und Semantik der Zielsprache verantwortlich. Diese Plugins sind so konzipiert, dass sie leicht

austauschbar sind, so dass Entwickler aus einer Vielzahl von unterstützten Sprachen wählen oder sogar eigene Plugins für neue oder spezielle Sprachen entwickeln können. Diese Vielseitigkeit stellt sicher, dass YmirScript ein breites Spektrum an Projekten und Entwicklungsszenarien abdecken kann, indem es sich an die einzigartigen Anforderungen jeder Zielsprache anpasst.

Ein weiterer Vorteil der Plugin-basierten Code-Generierung ist, dass sie ein hohes Maß an Wiederverwendbarkeit und Wartungsfreundlichkeit fördert. Da jedes Plugin für eine bestimmte Zielsprache zuständig ist, können Aktualisierungen oder Verbesserungen an der Implementierung einer bestimmten Sprache unabhängig vorgenommen werden, ohne die anderen Plugins oder den YmirScript-Kerncompiler zu beeinträchtigen. Dieser modulare Aufbau fördert ein effizientes und nachhaltiges Entwicklungs-Ökosystem, in dem Verbesserungen leicht integriert und von der Community genutzt werden können.

Darüber hinaus ermöglichen die Target-Plugins eine nahtlose Integration mit der bereits erwähnten integrierten semantischen Analyse, die sicherstellt, dass der generierte Code den spezifischen Anforderungen und Best Practices der Zielsprache entspricht. Diese Synergie zwischen den verschiedenen Phasen des Kompilierungsprozesses ermöglicht es YmirScript, konsistent hochwertigen, zuverlässigen und wartbaren Code in verschiedenen Zielsprachen zu erzeugen.

Zusammenfassend lässt sich sagen, dass die Plugin-basierte Code-Generierung in YmirScript eine wichtige Rolle bei der Erreichung der Ziele des Compilers, nämlich Modularität, Flexibilität und Erweiterbarkeit, spielt. Durch die Unterstützung einer Vielzahl von Zielsprachen und die Förderung einer kollaborativen Entwicklungsumgebung etabliert sich YmirScript als leistungsfähige und anpassungsfähige Lösung für moderne Softwareentwicklungsprojekte. Im nächsten Abschnitt werden wir die drei vordefinierten Zielsprachen-Plugins und ihre Struktur näher beleuchten, um ein umfassendes Verständnis ihrer Implementierung und Fähigkeiten zu vermitteln.

#### 3.4.2 Ziel-Plugins

Eines der Hauptmerkmale von Ymir ist die Unterstützung von Ziel-Plugins, die eine entscheidende Rolle dabei spielen, den Compiler in die Lage zu versetzen, Code in mehreren Sprachen zu erzeugen und sich an verschiedene Projektanforderungen anzupassen. Die Hauptmotivation für den Einsatz von Target-Plugins liegt in der Gewährleistung von Modularität und Flexibilität. Dieser Ansatz ermöglicht es den Entwicklern, aus einer Vielzahl von Ausgabesprachen und -formaten zu wählen, so dass sie nicht mehr an eine einzige Sprache gebunden sind. Diese Flexibilität ermöglicht es Entwicklungsteams, Ymir als ihr bevorzugtes Werkzeug einzusetzen, unabhängig von ihrer Wahl der Programmiersprache oder des Technologie-Stacks.

Ymir unterscheidet zwischen internen und externen Ziel-Plugins, die jeweils ihre eigenen Vorteile haben. Interne Plugins werden direkt in Ymir integriert und mit TypeScript/JavaScript entwickelt. Als Teil des Kerncompilers gewährleisten diese Plugins eine nahtlose und optimierte Erfahrung für Entwickler, die mit den unterstützten Sprachen arbeiten. Interne Plugins sorgen nicht nur für eine konsistente Entwicklungsumgebung, sondern garantieren auch, dass der generierte Code die Best Practices und Konventionen für die jeweiligen Sprachen einhält.

### *3 Entwurf und Umsetzung eines Boilerplate-Generators*

Externe Plugins hingegen sind eigenständige Programme, die in jeder beliebigen Sprache entwickelt werden können. Ymir führt diese Plugins aus und versorgt sie mit den erforderlichen Knoten, um Code für die Zielsprache zu generieren. Durch die Auslagerung der Code-Generierung an externe Plugins demonstriert Ymir seine Anpassungsfähigkeit, indem es ein breiteres Spektrum von Sprachen und Entwicklungsumgebungen unterstützt. Dieser Ansatz fördert ein integratives Ökosystem, das sich an Entwickler mit unterschiedlichen Hintergründen und Fähigkeiten wendet, und trägt so zur Vielseitigkeit und allgemeinen Attraktivität von Ymir als Mehrzweck-Kompilierungswerkzeug bei.

Um ein breites Spektrum von Entwicklungsanforderungen zu erfüllen, werden drei vordefinierte Ziel-Plugins integriert. Diese Plugins wurden sorgfältig ausgewählt, um die unterschiedlichen Anforderungen von Entwicklern zu erfüllen, die von Webanwendungen über Projekte auf Unternehmensebene bis hin zu umfassender API-Dokumentation reichen. Die Einbeziehung dieser Plugins zeigt Ymirs Engagement für Anpassungsfähigkeit und seine Fähigkeit, robuste Lösungen anzubieten, die für den dynamischen und sich ständig weiterentwickelnden Bereich der Softwareentwicklung relevant sind. Durch das Angebot einer Reihe integrierter Plugins zeigt Ymir seine Fähigkeit, eine Vielzahl von Anwendungsfällen zu bedienen, und etabliert sich damit als vielseitiges und wertvolles Werkzeug innerhalb der Softwareentwicklergemeinschaft.

**JavaScript – ExpressJS** Das JavaScript-ExpressJS-Plugin wurde entwickelt, um den Bedürfnissen von Entwicklern gerecht zu werden, die mit dem weit verbreiteten ExpressJS-Framework für JavaScript arbeiten. Durch die Integration der Unterstützung für dieses weit verbreitete Framework erleichtert Ymir die nahtlose Erstellung von Webanwendungen und beweist damit seine Anpassungsfähigkeit und Vielseitigkeit bei der Erfüllung der unterschiedlichen Anforderungen der Entwicklergemeinschaft. Die Integration dieses Plugins unterstreicht das Engagement von Ymir, robuste Lösungen im sich schnell entwickelnden Bereich der Web-Entwicklung anzubieten, und unterstreicht die Bedeutung von Ymir als leistungsstarkes und flexibles Werkzeug für Entwickler, die in verschiedenen Bereichen tätig sind. Die Integration des JavaScript-ExpressJS-Plugins unterstreicht auch die Bedeutung der Anpassung an etablierte Frameworks und Technologien und stellt sicher, dass Ymir für seine Zielgruppe relevant und wertvoll bleibt.

**Java – Spring Boot** Das Spring Boot-Plugin stellt eine bedeutende Erweiterung der Fähigkeiten von Ymir dar und erweitert seine Anwendbarkeit auf das Java-Ökosystem, das für seine Bedeutung in Unternehmensanwendungen weithin anerkannt ist. Durch die Erleichterung der Code-Generierung für Spring Boot-Projekte gewährleistet dieses Plugin die Einhaltung der Best Practices der Branche und fördert die Entwicklung qualitativ hochwertiger, wartbarer Software.

Die Einführung einer Konfigurationsdatei, die mit der Entwicklung des Spring Boot-Plugins zusammenfiel, markierte einen entscheidenden Meilenstein in der Entwicklung von Ymir. Diese Konfigurationsdatei ermöglicht die Anpassung verschiedener Einstellungen, wie z.B. Java-Pakete, an die individuellen Anforderungen eines jeden Projekts.

### *3 Entwurf und Umsetzung eines Boilerplate-Generators*

Dadurch wird die Anpassungsfähigkeit von Ymirs Ziel-Plugins erhöht, so dass Entwickler den generierten Code an die spezifischen Projektanforderungen anpassen können. Die Einbindung der Konfigurationsdatei ist ein Beispiel für Ymirs Engagement, Flexibilität und Anpassungsfähigkeit in verschiedenen Entwicklungsszenarien zu fördern.

**Swagger** Das Swagger-Plugin wurde entwickelt, um API-Dokumentation für die definierten Routen zu generieren. Durch die Unterstützung von Swagger ermöglicht Ymir den Entwicklern, gut dokumentierte und leicht verständliche APIs zu erstellen. Diese Fähigkeit fördert die effektive Zusammenarbeit und Kommunikation zwischen Teammitgliedern und Nutzern und stellt sicher, dass alle Beteiligten die Struktur und Funktionalität der API leicht navigieren und verstehen können.

# 4 Verschiedene Projekterweiterungen

Dieses Kapitel beinhaltet verschiedene Aspekte, die im Rahmen des Projekts rund um Ymir noch ausgearbeitet wurden. Es geht dabei nicht um eine konkrete Sache. Hier wird all das behandelt, was im Zuge des Projekts zu Ymir noch entstanden ist und was in den vorherigen Kapiteln keinen Platz gefunden hat, da es entweder nicht in das jeweilige Kapitel thematisch gepasst hätte oder erst im Nachhinein entstanden ist. Denn wie sich bereits vermuten lässt, bildet der Aufbau der Projektarbeit bis hierhin stets die chronologischer Zeitabfolge des gesamten Projekts ab. Dieses Kapitel handelt Dinge ab, die am Ende des Projekts entstanden sind. Der Inhalt dieses Kapitels wurde in der Einleitung deswegen auch als „Besonderheiten“ des Projekts bezeichnet.

## 4.1 Das Beibringen weiterer Sprachen

Dieser Abschnitt behandelt das Erweitern von Ymir um weitere Sprachen. Zu Beginn des Projekts ist es nur vorgesehen gewesen anhand von JavaScript die Funktionalität von Ymir zu demonstrieren. Nachdem das Projekt jedoch schneller vorangeschritten ist als gedacht, sind weitere Sprachen hinzugekommen.

Viel Mehraufwand ist das nicht, denn wie bereits beschrieben, ist die Modularität von Ymir ein Grundpfeiler des gesamten Projekts, welches auch als Anforderung in den Ymir-Qualitätsanforderungen niedergeschrieben wurde. Aufgrund der Machart von Ymir ist es grundsätzlich möglich diesem jede Sprache beizubringen, vorausgesetzt es handelt sich um eine Sprache des Web-Bereichs. Hinzugekommen sind im Nachhinein noch Unterstützung für einerseits Spring Boot – genauer Spring Boot mit Java – und andererseits Swagger. Damit bedient Ymir also auch eine „Sprache“, die in dieser Dokumentation bereits als Konkurrenzprodukt behandelt wird.

Um eine neue Sprache einzubinden, bedarf es im Grunde nur einer entsprechenden TypeScript-Datei im Repository von Ymir (Hufnagel u. Hufnagel (2023c)). Darin enthalten ist ein Ordner mit der Bezeichnung *targets*. Wie bereits in Kapitel 3 erläutert, werden als *targets* die Zielsprachen für den Compiler bezeichnet. In diesem Ordner werden wiederum Unterordner abgelegt, die den Namen ihrer jeweiligen Sprache tragen. Der Inhalt dieses Unterordners ist dann das eigentliche Plugin-Skript. Da Ymir mit TypeScript entwickelt wird, ist es erforderlich die Erweiterung ebenfalls in TypeScript zu schreiben. Durch die Richtlinien und Anforderungen, die Ymir beim Aufbau der Erweiterungen fordert, ähneln sich die einzelnen Plugins und sind dennoch verschieden, da eben jede Sprache ihre Eigenheiten mit sich bringt. Durch die Regeln, wie Plugins aufzubauen sind, aber vor allem dank der Modularität von Ymir, können Plugins binnen weniger Stunden geschrieben werden. Programmierer müssen dabei nicht sonderlich in den Compiler eingreifen oder diesen verändern, sondern nehmen Änderungen ausschließlich in ihrem Plugin-TypeScript-Skript vor. Der einzige Schritt, bei dem eine

weitere Datei, neben der Plugin-Datei angepasst werden muss, ist, dass eine Zeile in der „cli.ts“ im compiler-Ordner von Ymir hinzugefügt werden muss:

```
const plugins = [
    new JavaScriptExpressJsTargetPlugin(),
    new JavaSpringBootTargetPlugin(),
    new SwaggerTargetPlugin()
];
```

Dem hier zu sehenden plugins-Array muss das entsprechende neue Plugin angefügt werden. Alles weitere wird über die Plugin-eigene TypeScript-Datei geregelt.

Nähtere Code-Inspektionen an der Stelle wären möglicherweise sinnvoll, würden den Rahmen dieser Dokumentation jedoch sprengen. Aus diesem Grund wird darauf nun verzichtet. Details lassen sich selbst begutachten, in dem das Ymir-GitHub-Repository an dieser Stelle (Hufnagel u. Hufnagel (2023d)) aufgerufen wird. Im Repository eigenen Wiki ist zudem eine kleine Anleitung zum Schreiben von target-Plugins hinterlegt. Diese umfasst auch Guidelines, die für *targets* gelten und somit bei der Erstellung dieser beachtet werden sollen.

### 4.2 Plugin und Web-Präsenz für Ymir

Dieses Unterkapitel behandelt zwei Dinge, die aufgrund von genügend Zeit entstanden sind. Dabei bezieht sich der zweite Teil dieses Unterkapitels auf die Web-Präsenz von Ymir. Im ersten Teil geht es um eine Erweiterung für Visual Studio Code. Da sich für diese beiden Themen kein eigenes Kapitel angeboten hätte – dafür hätte der Umfang nicht ausgereicht – werden diese nun in diesem Unterkapitel zusammengefasst.

#### Ymir als Plugin für Visual Studio Code

Grundsätzlich lässt sich mit Ymir in jeder Entwicklungsumgebung arbeiten, die für Webentwicklung gedacht ist. Der Einfachheit halber ist im Projekt jedoch vorzugsweise Visual Studio Code verwendet worden. Einerseits, weil es kostenlos zur Verfügung und somit einem größtmöglichen Publikum offensteht und andererseits, weil Visual Studio Code dafür bekannt ist, sich einfach mit Erweiterungen personalisieren zu lassen, wobei diese Erweiterungen einfach zu konfigurieren sind. Wie bereits beim vorangegangenen Unterkapitel erwähnt, handelt auch dieses Unterkapitel von etwas, dass aufgrund der verfügbaren Zeit entstanden ist. Um die Entwicklung mit Ymir noch einfacher zu gestalten, ist für Visual Studio Code eine Erweiterung entstanden, die dem Programmierer unter die Arme greift. Diese Erweiterungen ersetzt die Installation von Ymir auf dem PC nicht. Die eigenständige Installation Ymir ist dennoch vonnöten.

Die Erweiterung bietet dabei dem Entwickler einige Basis Features, die bei der Entwicklung mit Ymir hilfreich sein sollen. Zunächst wird mit der Erweiterung ein einfaches Syntax-Highlighting eingeführt. Des Weiteren umfasst die Erweiterung für die wesentlichen Keywords Autovervollständigung. Die meisten Ymir-eigenen Keywords sollen sich damit vervollständigen lassen. Die Erweiterung bietet außerdem Kontrollmechanismen für zum einen Klammern und zum anderen Einrückungen, um den Code

funktional, sauber und übersichtlich zu halten. Die letzten zwei involvierten Funktionen sind keine notwendigen Kernfunktionen, sondern vielmehr „Nice to Have“-Funktionen. Das eine Feature gibt dem Entwickler die Möglichkeit die Kompilierung von Ymir-Skripts direkt innerhalb von Visual Studio Code zu starten. Dabei muss der Programmierer nur den Befehl „YmirScript: Compile“ in das Terminal schreiben. Die andere optionale Zusatzfunktion stellt Snippets für Standard-Code-Blöcke in Ymir bereit.

Zu finden ist diese Erweiterung, im Visual Studio Code eigenen Marktplatz für Erweiterungen. Dort einfach Ymir in die Suchleiste eingeben und auf eine Erweiterung mit dem Namen „YmirScript“ klicken. Dort stehen ebenfalls die hier benannten Funktionen noch einmal aufgelistet. Natürlich wäre für eine Erweiterungen noch mehr möglich gewesen. Jedoch sollte diese Erweiterung als kleiner Zusatz zum Projekt verstanden werden und nicht als Hauptbestandteil des Projekts, weswegen sich der Umfang der Erweiterung in Grenzen hält.

### Die Ymir-Webseite

Mit Blick auf die im Projekt entstandene Sprache mitsamt Compiler und nun auch eigener Visual Studio Code Erweiterung zeigt sich, dass Ymir viel mehr ist als nur eine Projektarbeit in einem Studium. Die letzten Wochen haben gezeigt, dass Ymir in der Lücke, die es füllt, kaum ähnliche „Konkurrenzprodukte“ hat und somit durchaus etwas Besonderes ist. Dies hat sich bereits in der in Kapitel 2 behandelten Recherche ergeben. Um die Präsenz von Ymir im Web zu vervollständigen, ist es daher sinnvoll gewesen, zum Abschluss des Projekts eine Webseite für Ymir bereitzustellen, die neben dem eigenen Repository eine weitere Möglichkeit bietet, sich etwas über Ymir zu informieren (Hufnagel (2023a)).

Den Hauptzweck, den die Webseite erfüllt ist, Informationen darüber bereitzustellen, wie sich Ymir installieren lässt. Wie bereits bei dem vorherigen Unterkapitel erläutert, ist die Webseite – ebenso wie die Erweiterung für Visual Studio Code – einfach gehalten und dient in erster Linie dazu kurz und prägnant die Hauptinformationen zu Ymir preiszugeben. Auf einer Übersichtsseite werden in Kurzform alle Funktionalitäten zu Ymir aufgezeigt. Innerhalb weniger Sätze wird das beschrieben, was im Laufe dieser Projektdokumentation ausführlich erläutert und beschrieben wurde. Und am Fuß des Seiten gibt es eine Möglichkeit das Repository zu Ymir aufzurufen.

Viel bietet die Webseite zu Ymir nicht. Zumaldest noch nicht. Sie hat für die Funktionalität des Projekts eigentlich keinen Mehrnutzen. Sie ist nur deswegen entstanden, weil bereits jetzt absehbar ist, dass aus Ymir noch mehr wird als nur diese Projektarbeit.

# 5 Schlussteil der Projektdokumentation

In erster Linie soll das in diesem Kapitel enthaltene Fazit aufzeigen, ob und welche Ziele Ymir nun schlussendlich erreicht hat. Damit geht auch eine Gegenüberstellung der Vor- und Nachteile von Ymir einher, die das Projekt vergleichbaren Produkten gegenüber hat. Im letzten Abschnitt wird mittels kritischer Reflexion ein Fazit zu Ymir und dem Projekt drumherum gezogen und abschließend ein Ausblick gegeben, welcher darstellt, wie sich Ymir noch weiter entwickeln könnte.

## 5.1 Ymir im Vergleich zu anderen

Nachdem nun Ymir soweit für das Projekt entwickelt ist, lässt sich ein Vergleich zu anderen Produkten ziehen. Diese Gegenüberstellung wird mittels Kriterien-Tabelle realisiert, wobei Ymir und Swagger in dieser gegenübergestellt werden. Auch hier wird Swagger herangezogen, da dieses, wie bereits in der Konkurrenzanalyse erwähnt, das Produkt ist, dass Ymir am nächsten kommt. Im Anschluss an jede Tabelle kommt ein kleiner Erläuterungstext, wofür die einzelnen Punkte in den Tabellen stehen.

Kriterium	Ymir	Swagger
Einfachheit	Einfachere, minimalere Syntax	Etwas komplexere Syntax, YAML- oder JSON-Format
Dokumentation	Weniger verbreitet, begrenzte Ressourcen und Dokumentation	Umfangreiche Ressourcen und Dokumentation verfügbar
Integration	Integration in begrenzte Anzahl von Frameworks und Tools	Breite Integration in verschiedene Frameworks und Tools
Code-Generierung	Flexibel passt sich dynamisch an die Definitionen an	Code-generierung basierend auf festgelegten Definitionen
Code-Adaption	Unterstützt kontinuierliche Code-Anpassungen während der Entwicklung	Begrenzte Unterstützung für kontinuierliche Code-Anpassungen
Wartung	Einfacher, leichter zu pflegen. Einstieg einfach	Möglicherweise schwieriger zu pflegen aufgrund größerer Komplexität. Einstieg etwas komplizierter
Standardisiert	Weniger standardisiert und verbreitet	Weit verbreitet und standardisiert ( <a href="#">OpenAPI-Spezifikation</a> )
Flexibilität	Bietet eine flexible, leichtgewichtige Lösung	Flexibel, aber möglicherweise umfangreicher und komplexer

Abbildung 5.1: Ymir im Vergleich zu Swagger (Hufnagel, 2023b)

Mit Blick auf die Tabelle fällt auf, dass weder Ymir noch Swagger eindeutig besser oder schlechter ist. Stattdessen gibt es Anwendungsbereiche, in denen die Nutzung von Ymir sinnvoller ist und es gibt Fälle, in denen die Nutzung von Swagger besser ist. So sollte Swagger zum Beispiel eher in Projekt zum Einsatz kommen, bei dem die Entwickler nur ein geringes Vorwissen an Programmierung haben und somit eine

umfangreichere Dokumentation eher benötigt wird. Ymir wiederum ist wartungstechnisch besser zu handhaben und kann somit auch bei bereits bestehenden Projekt gut integriert werden.

### 5.2 Kritische Reflexion zum Projekt

Dieses Unterkapitel unterteilt sich in zwei Teile. Zum einen geht es hier um die erreichten Ziele von Ymir. Dabei werden diese mit den zu Beginn gesteckten Zielen verglichen und geschaut, welche erreicht wurden und welche eben nicht. Und zum anderen wird das gesamte Projekt kritisch reflektiert. Hierbei geht es darum, was gut funktioniert hat und was schlecht funktioniert hat. Sowohl im Bezug auf Ymir selbst als auch eben im Bezug auf das Projekt im Großen und Ganzen. Aber den Anfang macht die Analyse der erreichten Ziele. Wobei es mehrere Ziele eigentlich nicht gibt. Stattdessen ist zu Beginn des Projekts ein umfangreiches Hauptziel entstanden, welches sich aus mehreren kleinen Zielen zusammensetzt.

#### Möglichkeit zum Generieren von Boilerplate-Code

Das Hauptziel des gesamten Projekts ist es, eine Möglichkeit zu entwickeln, die Boilerplate-Code generieren kann und dabei dem Entwickler möglichst viel Arbeit abnimmt. Dieses Ziel ist natürlich erreicht worden, denn ohne dessen Erreichung wäre das Projekt schlichtweg nicht fertig. Mit Ymir ist ein Compiler entstanden, der aus einem Skript, welches Programmierer in der Ymir-eigenen Sprache YmirScript anlegen, funktionalen Code für das Web generieren kann.

Ymir kommt dabei modular daher. Zwar gibt es durchaus strikte Regeln und Richtlinien, die es bei der Nutzung von Ymir einzuhalten gilt, jedoch kann Ymir auf die eigenen Bedürfnisse angepasst werden. So lässt der Compiler die Einarbeitung weiterer unterstützter Sprachen zu und die bereits vorhandenen Sprachen können weiter ausgebaut und angepasst werden. Um weiter Zeit zu sparen, überschreibt Ymir bestehenden Code nicht einfach, sondern passt sich diesem an. Dies gibt dem Programmierer möglichst viel Kontrolle darüber, was passiert.

In Sachen Semantik und Syntax ist sowohl Ymir als auch YmirScript möglichst einfach gehalten. Trotz der Hürden, die das Erlernen einer neuen „Sprache“ so mit sich bringt, kann der Einstieg in Ymir recht einfach gelingen. Das Ergebnis, welches Ymir liefert ist dabei unabhängig von Soft- oder Hardware. Die Nutzung von Ymir an sich ist dabei möglichst einfach gehalten. Mit einem Befehl in die PowerShell (unter Windows) lässt sich Ymir mitsamt PATH-Variablen installieren. Des Weiteren gibt es, wie bereits im vorherigen Kapitel benannt, ein eigenständiges Visual Studio Code Plugin, um den Umgang zu erleichtern.

Alles in allem lässt sich sagen, dass das Ziel vollumfänglich erreicht worden ist. Die ursprüngliche Zielsetzung wurde sogar übertroffen, da das Projekt schneller vorangeschritten ist als geplant. Damit hat es im Laufe der letzten Wochen mehr an Umfang gewonnen, um die Zeit bis zur Deadline effektiv und effizient zu nutzen. Damit besitzt das Projekt nicht nur ein ordentliches Fundament, auf dem sich weiteres aufbauen lässt,

sondern bietet auch Open-Source-Ansätzen die Möglichkeit mit Ymir zu arbeiten und es weiterzuentwickeln.

### Selbstkritisches Fazit

Der letzte Absatz lässt vermuten, dass das Fazit dann ja nur positiv ausfallen kann. Aber dem ist nicht so. Insbesondere die Arbeitsweise lässt noch etwas zu wünschen übrig. Wie ja bereits bekannt ist, ist Ymir in Teamarbeit eines Zweier-Teams aus Lukas und Paul Hufnagel entstanden. Während Paul eher ein Theoretiker ist, arbeitet Lukas stets mit Fokus auf die Praxis. Das klingt eigentlich nach der perfekten Kombination, sorgt jedoch hin und wieder für Probleme. Durch seine umfangreiche praktische Erfahrung ist Lukas jemand, der dazu tendiert den zweiten Schritt vor dem ersten zu machen und direkt in die Entwicklung einzusteigen. Wenn es nur nach ihm ginge, würde der Theorieteil entweder komplett zu kurz kommen oder gar nicht beachtet werden. Paul wiederum ist in Sachen Planung zwar durchaus strukturiert, hält sich jedoch gerne zu sehr mit Details auf, weswegen sich Zeitpläne häufig strecken. Trotz ihrer Präferenzen haben beide jeweils sowohl an der Theorie als auch an der Praxis mitgearbeitet, da nach 5 Semestern Studium jeder in beiden Bereichen Erfahrung hat und somit auch das volle Erfahrungsspektrum zum Einsatz kommen sollte. Die entsprechenden Anteile lassen sich der Arbeitsmatrix im Einleitungskapitel entnehmen. Dabei zeichnet sich Paul hauptverantwortlich für die Theorie, wohingegen Lukas Hauptverantwortlicher des praktischen Teils ist. Zudem haben beide eben jeweils den anderen in seinem Spezialgebiet unterstützt.

Trotz dieser Problemstellen spricht das Projekt für sich. Es werden mehr Meilensteine erreicht als geplant. Das Projekt hat mehr Umfang als zu Beginn festgehalten. Dies lässt darauf schließen, dass die Arbeitsweise dennoch funktioniert hat.

In Bezug auf Ymir selbst lässt sich das Resümee ziehen, dass aus dem Projekt ein funktionierender Compiler für die Generierung von Boilerplate-Code in der Webentwicklung entstanden ist. Der Compiler kommt mit einer eigenen Sprache – YmirScript – daher. Diese nutzt der Compiler, um daraus den besagten Boilerplate-Code zu generieren. Syntax und Semantik dürften Webentwicklern bekannt vorkommen, wodurch die Einarbeitungszeit möglichst gering gehalten wird. Zusätzliche Sprachen und Features lassen sich ohne große Umwege einbauen, da die Möglichkeit zur Erweiterung offiziell geboten wird. Insbesondere die Funktion, dass sich der Compiler dem Code anpasst und ihn nicht einfach überschreibt, ist von Anfang an ein elementares Feature gewesen. Auch dieses Feature ist erfolgreich umgesetzt worden, um so möglichst viel Zeittersparnis für den Entwickler zu gewährleisten.

Trotz dieser ganzen Errungenschaften ist Ymir keineswegs perfekt. Zunächst fällt negativ auf, dass die Einstiegshürden hoch sind. Wohingegen Swagger, welches im Laufe dieser Dokumentation schon häufiger herangezogen wurde, durchaus auch von weniger erfahrenen Personen verwendet werden kann, bedarf es der Nutzung von Ymir ein grundlegendes Vorwissen in der (Web-)Entwicklung. Es gibt kein Online-Tool mit sich der Code einfach zusammenklicken lässt. Stattdessen muss der Entwickler eigenhändig ein Ymir-Skript schreiben, aus dem der Compiler Code generieren kann. Somit muss der

Programmierer dem Compiler konkret sagen, was er braucht, wodurch die Zielgruppe von Ymir stark eingeschränkt wird.

Des Weiteren bedient Ymir aktuell noch ein eingeschränktes Feld im Web-Bereich. Zu der Webentwicklung zählt ja mehr als Web-APIs, wie REST, wie zum Beispiel die Arbeit mit Frontend-Frameworks wie etwa VueJS.

Was im Bezug auf das Projekt und Ymir zudem negativ ausfällt, ist die Tatsache, dass die Modellierung, wie mit den UML-Werkzeugen nicht funktioniert haben. Alle erstellten Diagramme fallen in ihrer Art recht spartanisch aus und haben für den weiteren Verlauf des Projekts eigentlich keine weitreichende Bedeutung. Das kann natürlich daran liegen, dass Ymir nicht in die Kategorie eines klassischen Programms fällt. So besitzt Ymir kein Frontend und wird vor allem über ein Terminal gesteuert. Und Modellierungswerzeuge wie die UML gibt es in erster Linie für die Ausarbeitung von solch klassischen Programmen. Dennoch fällt der theoretische Teil dadurch eher mickrig aus und fällt damit eher in die Kategorie negativer Projektaspekte.

Unabhängig von den positiven oder negativen Errungenschaften, konnte das Projekt sein Ziel erreichen und sogar übertreffen. Zumindest nach der Meinung der Autoren. Mit dem daraus resultierenden Fundament kann im nächsten und letzten Unterkapitel nun ein Ausblick darüber gegeben werden, womit es bei Ymir weitergehen könnte.

### 5.3 Ausblick auf die Zukunft von Ymir

Im letzten Abteil gibt es einen Ausblick auf eine mögliche Zukunft von Ymir. Wie für einen Zukunftsausblick üblich enthält dieser Abschnitt kaum handfeste Fakten, sondern vor allem Spekulationen.

Wie bereits in vorherigen Kapiteln erwähnt, bietet Ymir nun mehr Umfang als ursprünglich vorgesehen. Somit sind einige Ideen, die gar nicht hätten umgesetzt werden sollen nun doch im Verlauf des Projekts Wirklichkeit geworden. So zum Beispiel die Unterstützung mehrerer Sprachen direkt zu Beginn. Anfangs geplant gewesen ist, dass der Compiler aus der eigenen Sprache YmirScript JavaScript-Code generieren können soll. Geplant gewesen ist jedoch auch, die Möglichkeit zu bieten, Ymir, um weitere Sprachen erweitern zu können. Durch den Mehrgewinn an Zeit sind weitere Sprachen hinzugekommen. Damit gibt es einen Beweis, dass der Compiler fähig ist weitere Sprachen zu erlernen. Und dies ist bereits die erste Möglichkeit, wie sich Ymir in Zukunft ausbauen lässt: Die Unterstützung der Sprachen kann ausgebaut werden. Jeder, der mit Webentwicklung (etwas) Erfahrung hat, kann die Sprache seiner Wahl in den Compiler einbringen, ohne tiefgreifende Änderungen an diesem vornehmen zu müssen.

Des Weiteren ließe sich der Zielbereich von Ymir erweitern. Mit Web-APIs, wie REST, deckt Ymir einen zentralen Aspekt des heutigen Webs ab. Dennoch bietet dieses noch mehr Möglichkeiten. So könnte Ymir beispielsweise Einzug in die Frontend-Entwicklung erhalten. Entweder direkt eingebunden in gängige Frameworks, wie Vue oder Angular oder in pures HTML, (S)CSS und ähnliches. Im Bezug auf die bereits unterstützten Sprachen ließe sich deren Code ebenfalls erweitern. So bisher sicherlich nicht jedes Feature von beispielsweise Spring Boot abgedeckt. Die bisher unterstützten Sprachen könnten weiter ausgearbeitet werden.

## *5 Schlussteil der Projektdokumentation*

Obwohl das Projekt selbst in TypeScript geschrieben ist, ist die Generierung bisher auf JavaScript beschränkt. Es ist vermutlich nur eine Frage der Zeit, bis der Support für TypeScript dazu kommt. Durch die für das Projekt gewählte GPL-3.0 Lizenz ist das Projekt Ymir für Open Source geeignet. Wenn Ymir entsprechende Beachtung findet, bietet diese Lizenz die Möglichkeit, dass Ymir in alle möglichen Bereiche Einzug erhält, egal ob private Projekte oder kommerzielle Projekte bei namenhaften Firmen. So kann Ymir von einer Projektarbeit zweier Studenten, eventuell zu einem bedeutenden Compiler in der Webentwicklung heranwachsen.

Neben all den genannten Spekulationen gibt es jedoch auch Fälle, in denen Ymir sich definitiv weiterentwickeln wird. So wird Lukas Ymir in seiner auf dieses Projekt folgenden Bachelorarbeit Ymir als zentrales Thema weiter behandeln und ausarbeiten. Und da eine Bachelorthesis einen größeren Umfang besitzt als eine solche Projektarbeit, wird Ymir vermutlich in den kommenden Wochen und Monaten noch einige Iterationen durchlaufen und weiter heranwachsen.

Womöglich wird aus diesem Projekt mehr entstehen, als zu Beginn gedacht und geplant gewesen ist. Doch statt darüber weiter zu spekulieren, sollte nun einfach abgewartet werden. Denn die Zeit wird zeigen, welchen Weg das Projekt Ymir einschlagen wird.

# **Abbildungsverzeichnis**

1.1	Arbeitsmatrix zum Ymir-Projekt (Hufnagel u. Hufnagel, 2023a) . . . . .	4
2.1	Themenfeldcluster zur Leitfrage (Hufnagel u. Hufnagel, 2023b) . . . . .	7
2.2	Domänenmodell im Ist-Zustand (Hufnagel, 2023c) . . . . .	9
2.3	Domänenmodell im Soll-Zustand (Hufnagel, 2023d) . . . . .	10
2.4	Ymir als Use Case-Diagramm (Hufnagel, 2023e) . . . . .	11
3.1	Die Ymir-Qualitätsanforderungen (Hufnagel, 2023f) . . . . .	17
5.1	Ymir im Vergleich zu Swagger (Hufnagel, 2023b) . . . . .	30

# Literaturverzeichnis

- [Barragan 2017] BARRAGAN, Carlos: *The problems with Swagger*. <https://www.novatec-gmbh.de/en/blog/the-problems-with-swagger/>. Version: März 2017. – Abgerufen am 21. April 2023
- [Hauschildt oD] HAUSCHILD, Pia-Sophie: *Low Code/No Code - die neue Art der Softwareentwicklung?* <https://nativdigital.com/low-code-no-code/>. Version: o.D.. – Abgerufen am 21. April 2023
- [Hufnagel 2023a] HUFNAGEL, Lukas: *Ymir - Simplify REST APIs with Adaptive Compiler*. <https://ymirscript.dev/>. Version: April 2023. – Abgerufen am 28. April 2023
- [Hufnagel 2023b] HUFNAGEL, Lukas: *Ymir im Vergleich zu Swagger*. 2023
- [Hufnagel u. Hufnagel 2023a] HUFNAGEL, Lukas ; HUFNAGEL, Paul: *Arbeitsmatrix zum Ymir-Projekt*. 2023
- [Hufnagel u. Hufnagel 2023b] HUFNAGEL, Lukas ; HUFNAGEL, Paul: *Themenfeldcluster für Ymir*. 2023
- [Hufnagel u. Hufnagel 2023c] HUFNAGEL, Lukas ; HUFNAGEL, Paul: *ymirscript/ymir: The main repository*. <https://github.com/ymirscript/ymir>. Version: April 2023. – Abgerufen am 28. April 2023
- [Hufnagel u. Hufnagel 2023d] HUFNAGEL, Lukas ; HUFNAGEL, Paul: *ymir/targets at main*. <https://github.com/ymirscript/ymir/tree/main/targets>. Version: April 2023. – Abgerufen am 28. April 2023
- [Hufnagel 2023c] HUFNAGEL, Paul: *Domänenmodell im IST-Zustand*. 2023
- [Hufnagel 2023d] HUFNAGEL, Paul: *Domänenmodell im SOLL-Zustand*. 2023
- [Hufnagel 2023e] HUFNAGEL, Paul: *Use Case zu Ymir*. 2023
- [Hufnagel 2023f] HUFNAGEL, Paul: *Die Ymir-Qualitätsanforderungen*. 2023
- [IONOS 2020] IONOS: *Swagger: Mehr Komfort bei der API-Entwicklung*. <https://www.ionos.de/digitalguide/websites/web-entwicklung/was-ist-swagger/#:~:text=Ein%20kleiner%20Nachteil%20von%20Swagger,Blueprint%20mit%20einer%20Markdown%2DSyntax>. Version: September 2020. – Abgerufen am 21. April 2023
- [JetBrains oD] JETBRAINS: *Essential tools for software developers and teams*. <https://www.jetbrains.com/>. Version: o.D.. – Abgerufen am 21. April 2023

## Literaturverzeichnis

- [Kuhlemann oD] KUHLEMANN, Oliver: *Brainfuck Programmiersprache / Interpreter.* <https://kryptografie.de/kryptografie/chiffre/brainfuck.htm>. Version: o.D.. – Abgerufen am 27. April 2023
- [Lang u. Augsten 2017] LANG, Mirco ; AUGSTEN, Stephan: *Was sind esoterische Programmiersprachen?* <https://www.dev-insider.de/was-sind-esoterische-programmiersprachen-a-617013/>. Version: Juli 2017. – Abgerufen am 27. April 2023
- [Oracle oD] ORACLE: *Oracle APEX*. <https://apex.oracle.com/de/>. Version: o.D.. – Abgerufen am 21. April 2023
- [SimplyTest oD] SIMPLYTEST: *SWAGGER: DIE POPULÄRE TECHNOLOGIE FÜR DIE API-ENTWICKLUNG*. <https://www.testautomatisierung.org/lexikon/swagger/#:~:text=Was%20ist%20Swagger%3F,Wordnik%2DMitbegr%C3%BCnders%2C%20Tony%20Tam>. Version: o.D.. – Abgerufen am 21. April 2023
- [SmartBear oDa] SMARTBEAR: *How to Use Swagger Inspector*. <https://swagger.io/docs/swagger-inspector/how-to-use-swagger-inspector/>. Version: o.D.. – Abgerufen am 21. April 2023
- [SmartBear oDb] SMARTBEAR: *Swagger: API Documentation Design Tools for Teams*. <https://swagger.io/>. Version: o.D.. – Abgerufen am 21. April 2023
- [Zaveri 2018] ZAVERI, Meet: *What is boilerplate and why do we use it? Necessity of coding style guide.* <https://www.freecodecamp.org/news/whats-boilerplate-and-why-do-we-use-it-lets-check-out-the-coding-style-guide-ac2>. Version: Januar 2018. – Abgerufen am 20. April 2023

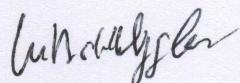
## **Eidesstattliche Erklärung - Lukas Hufnagel**

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben.

Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Köln, 1. Mai 2023



Lukas Hufnagel

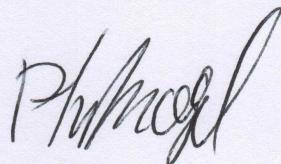
## **Eidesstattliche Erklärung - Paul Hufnagel**

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben.

Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Köln, 1. Mai 2023

A handwritten signature in black ink, appearing to read "Hufnagel".

Paul Hufnagel