HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Bachelorthesis

Entwicklung einer Webseite zur Veranschaulichung der Ver- und Entschlüsselungsmethoden der Enigma.

Vorgelegt von:	Tobias Steidle
wohnhaft in:	Dreiländerring 44, 88212 Ravensburg
geboren am:	18.10.1992
geboren in:	Weingarten

Hochschule Ravensburg-Weingarten

Fakultät:	Elektrotechnik und Informatik
Studiengang:	Informatik/Elektrotechnik PLUS

1. Prüfer:	Prof. Dipl.-Math. Ekkehard Löhmann
2. Prüfer:	Prof. Dr.-Ing. Thorsten Weiss

Ausgabedatum:	11.12.2024
Abgabedatum:	08.07.2025

Eidesstattliche Erklärung

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

Ravensburg, 07.07.2025

Tobias Steidle

Inhaltsverzeichnis

1 Einleitung

Die Enigma war eine der bedeutendsten Verschlüsselungsmaschinen des 20. Jahrhunderts und spielte eine zentrale Rolle in der militärischen Nachrichtenübermittlung der deutschen Wehrmacht während des Zweiten Weltkriegs. Die Entzifferung ihrer chiffrierten Nachrichten gilt als eine der größten kryptografischen Leistungen dieser Epoche. Besonders bekannt ist der Beitrag britischer Kryptoanalytiker um Alan Turing und Gordon Welchman in Bletchley Park, deren Arbeit wesentlich zum alliierten Kriegsgewinn beitrug.

Weniger im öffentlichen Bewusstsein verankert, aber ebenso grundlegend, war die Vorarbeit polnischer Kryptologen um Marian Rejewski. Bereits in den 1930er-Jahren gelang es ihnen, die logischen Strukturen der Enigma zu durchdringen und eine methodische Grundlage zu schaffen, auf der die spätere britische Entzifferungsarbeit aufbauen konnte.

Während es bereits zahlreiche und sehr gute Enigma-Simulatoren gibt, wie etwa die webbasierte Anwendung von Palloks[1], fehlt bislang eine Anwendung, die das Zyklometer und die darauf basierende Zyklenanalyse simuliert und visuell darstellt.

2 Zielsetzung

Die Zielsetzung dieser Arbeit besteht in der Bereitstellung bestehender C-Implementierungen der Enigma-Maschine und des Zyklometers als öffentlich zugängliche Webanwendung. Die Applikation soll über das Internet nutzbar sein und dabei grundlegende Anforderungen an Sicherheit und Wartbarkeit erfüllen.

Da es keine detaillierten Vorgaben zur technischen Umsetzung gab, liegt ein wesentlicher Bestandteil der Arbeit in der eigenständigen Konzeption einer geeigneten Architektur und Implementierungsstrategie.

Dabei werden folgende Teilziele formuliert und im Verlauf der Arbeit realisiert:

- Analyse und Integration einer bestehenden C-Implementierung der Enigma und des Zyklometers in eine webbasierte Anwendung
- Konzeption und Entwicklung einer geeigneten Backend-Architektur zur sicheren Anbindung der C-Bibliothek
- Gestaltung einer benutzerfreundlichen Weboberfläche zur intuitiven Nutzung der Anwendung
- Speicherung und Verwaltung kryptografischer Katalogdaten in einer geeigneten Datenbank
- Sicherstellung des sicheren und zuverlässigen Betriebs der Webanwendung

3 Kryptographische Grundlagen

Im Folgenden werden die kryptographischen Grundlagen der Enigma-Maschine sowie des Zyklometers erläutert. Ein tiefes Verständnis dieser Komponenten ist notwendig, um die Funktionsweise der Verschlüsselung und ihre darauf beruhende Analyse nachvollziehen zu können.

Dieses Kapitel basiert in Teilen auf der Vorarbeit des Kommilitonen Hasanic zu einem früheren Projekt zur Enigma-Maschine. Die inhaltliche Darstellung sowie die verwendeten Quellen orientieren sich daher an dieser Arbeit, wobei die Inhalte für den Kontext dieser Arbeit angepasst wurden.

3.1 Die Enigma

Die Enigma besteht aus mehreren Komponenten, die zusammen eine Verschlüsselung mit wechselnden Buchstabenersetzungen (polyalphabetisches Verfahren) realisieren. Abbildung 1 zeigt eine typische Enigma-Maschine. Erkennbar sind das **Steckerbrett**, das **Tastenfeld**, das **Lampenfeld** sowie die dahinterliegenden von außen sichtbaren Zahnräder der **Walzen** und die danebenliegenden **Sichtfenster**.

Abbildung 1: Enigma-Maschine mit Steckerbrett, Tastenfeld und Walzen. Quelle: [2]

Wird eine Taste gedrückt, fließt ein elektrischer Strom durch die Maschine und endet bei einer Lampe, die den verschlüsselten Buchstaben anzeigt. Die Walzen rotieren, sodass selbst identische Klartextzeichen unterschiedliche Chiffren erzeugen. Um eine Nachricht zu entschlüsseln, muss die Enigma in dieselbe Ausgangskonfiguration versetzt werden, mit der die Verschlüsselung erfolgte.

3.1.1 Komponenten der Enigma

Die Enigma I verfügt über einen Walzensatz von fünf verschiedenen **Walzen (engl. rotors)**, nummeriert mit römischen Ziffern I–V. Aus diesem Satz werden drei Walzen ausgewählt und in einer bestimmten Reihenfolge in die drei Walzenplätze der Maschine eingesetzt. Diese Auswahl und Reihenfolge bezeichnet man als **Walzenlage**. Die Walzen sind jeweils auf der einen Seite mit flachen Kontakten und auf der anderen mit federbelasteten Stiften versehen. Durch die interne Verdrahtung der Walzen wird jeder Buchstabe auf genau einen anderen abgebildet. Je nach Drehung der Walze ändert sich die Abbildung.

Jede Walze besteht aus einem festen Verdrahtungskern sowie einem drehbaren äußeren Ring, auf dem Zahlen angebracht sind. Dieser äußere Ring trägt eine **Kerbe (engl. notch)**, die bestimmt, wann die nachfolgende Walze beim Tastendruck mitrotiert.

Walzenstellung (engl. rotor position) bezeichnet die jeweils aktuelle Position der Walze im Betrieb – also welcher Buchstabe gerade oben im Sichtfenster angezeigt wird. Nach jedem Tastendruck wird die rechte (schnelle) Walze um einen Buchstaben weitergedreht. Die anderen Walzen drehen sich nur, wenn die vorherige Walze ihre Einkerbung erreicht hat.

Ringstellung (engl. ring position) beschreibt hingegen die Position der inneren Verdrahtung relativ zum äußeren Ring mit Buchstaben. Sie legt damit fest, an welcher Buchstabenposition sich die Schaltkerbe befindet und wie der elektrische Signalweg durch die Walze verläuft. Die Ringstellung wird vor dem Einsatz mechanisch über eine Schraube verstellt und bleibt während des Betriebs konstant. Sie beeinflusst sowohl den Verschlüsselungsweg als auch das Schaltverhalten der Maschine.

Die Unterscheidung zwischen Walzenstellung und Ringstellung ist entscheidend: Während die Walzenstellung angibt, wo sich die Walze gerade befindet, legt die Ringstellung fest, wie ihre interne Verdrahtung zum sichtbaren Alphabet ausgerichtet ist.

Die Kombination aus **Walzenlage**, **Walzenstellung** und **Ringstellung** wird als **Walzenkonfiguration** bezeichnet.

Abbildung 2: Enigma-Walze mit Einkerbung und Ring. Quelle: [3]

Abbildung 2 zeigt drei Walzen. Auf der linken Walze ist die Schaltkerbe gut zu erkennen. Ebenfalls zu sehen sind die flachen Kontakte auf der einen und die Stifte auf der anderen Seite der Walzen. Die **Walzenstellung** selbst wird hingegen erst in der Enigma-Maschine durch das Sichtfenster sichtbar.

Die **Eintrittswalze (ETW)** verbindet das Steckerbrett mit der ersten rotierenden Walze. Sie bildet das Alphabet 1:1 ab, rotiert selbst nicht und besitzt daher keine kryptografische Wirkung, wohl aber eine fest verdrahtete Zuordnung der Kontakte.

Am Ende des Stromkreises reflektiert die **Umkehrwalze (UKW, engl. reflector)** das elektrische Signal zurück durch die Walzenkette. Sie ist fest eingebaut und sorgt durch ihre Verdrahtung dafür, dass kein Buchstabe auf sich selbst abgebildet werden kann – eine charakteristische Eigenschaft der Enigma-Verschlüsselung.

Das **Steckerbrett (engl. plugboard)** besitzt 26 Buchstabenanschlüsse. Mit bis zu 13 Kabelverbindungen können Buchstabenpaare vertauscht werden (z. B. A $\leftrightarrow$ G). Diese Substitution findet sowohl auf dem Hin- als auch auf dem Rückweg des Signals statt und erhöht die kombinatorische Komplexität der Maschine.

3.1.2 Verschlüsselungsablauf

Der Ablauf der Verschlüsselung eines einzelnen Zeichens gestaltet sich wie folgt:

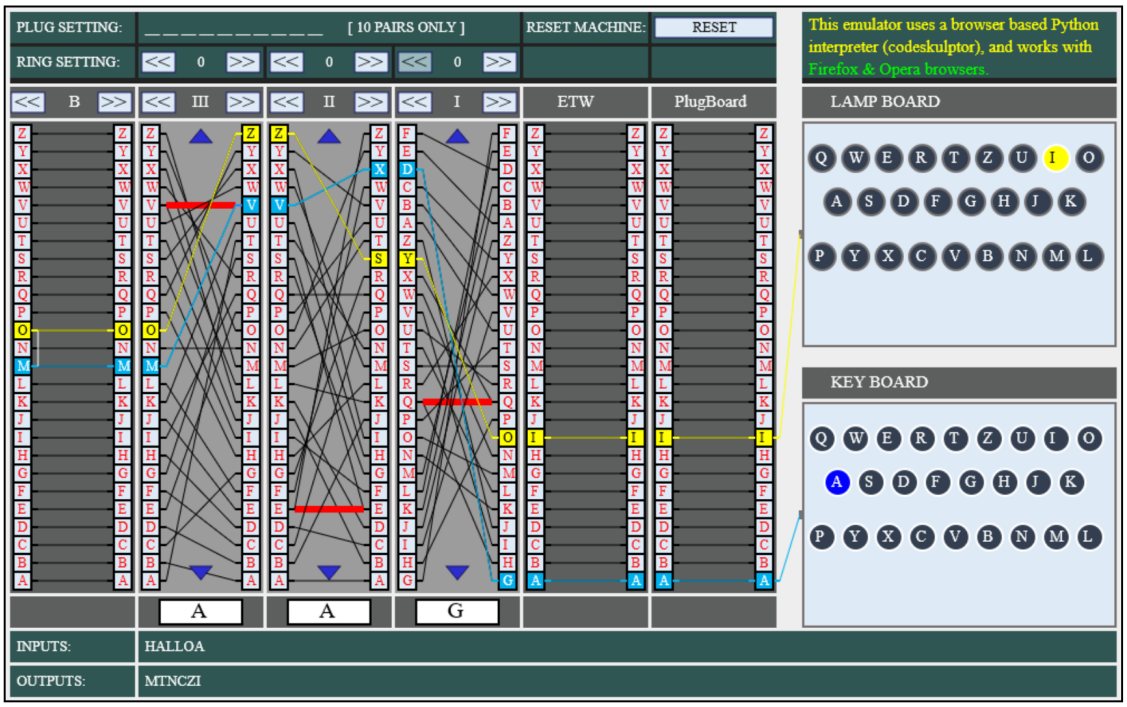


Abbildung 3: Ablauf der Verschlüsselung. Quelle: [4]

In Abbildung 3 ist der genaue Ablauf der Verschlüsselung visualisiert. Die blaue Linie stellt den Signalweg vom Eingabebuchstaben durch die Walzen zur Umkehrwalze dar, während die gelbe Linie den Rückweg von der UKW zurück zum Lampenfeld zeigt. Die roten Markierungen auf den Walzen symbolisieren die Einkerbungen, die den Signalfluss beeinflussen. Im Folgenden wird der Verschlüsselungsprozess Schritt für Schritt erläutert:

1. Konfiguration des Steckerbretts.
2. Einstellung der Ringstellungen und Walzenlage.
3. Einsetzen der Walzen in die Maschine.
4. Drücken einer Taste am Tastenfeld.
5. Die erste Walze rotiert, ggf. folgen weitere Walzen.
6. Das elektrische Signal durchläuft nacheinander:
   - das Steckerbrett,
   - die Eintrittswalze,
   - die drei rotierenden Walzen (Hinweg),

- die Umkehrwalze,
- die drei Walzen (Rückweg),
- die Eintrittswalze (Rückweg),
- das Steckerbrett (Rückweg).

7. Die entsprechende Lampe leuchtet auf und zeigt den verschlüsselten Buchstaben an.

3.1.3 Varianten der Enigma

Die **Enigma M3** stellt eine Weiterentwicklung der Enigma I dar und basiert auf demselben kryptographischen Prinzip. Diese verwendet drei Walzen, die aus einem Walzensatz von acht statt fünf Walzen ausgewählt werden. Durch die erweiterte Auswahl und variable Reihenfolge der Walzen wird die Anzahl möglicher Schlüsselkonfigurationen signifikant erhöht.

Die **Enigma M4** ersetzt die M3-Umkehrwalze durch eine zusätzliche, vierte Walze und eine schmalere Umkehrwalze, die ausschließlich an dieser Position verwendet werden können. Die ersten drei Walzen entsprechen physisch und funktional denen der M3. Die vierte Walze ist schmaler, besitzt keine Ringstellung, rotiert nicht mit und ist nicht mit den ersten drei Walzenplätzen kompatibel.

Für die vierte Position stehen zwei Walzen (`Beta`, `Gamma`) und zwei schmale Umkehrwalzen (`UKW-b`, `UKW-c`) zur Verfügung, die beliebig kombiniert werden können. Zur Abwärtskompatibilität mit der M3 wird die Walze `Beta` in Stellung 'A' fixiert und mit der `UKW-b` kombiniert. Analog gilt dies für `Gamma` und `UKW-c`.

Beispielrechnung zur Walzenwahl: Die Anzahl der möglichen Anordnungen von Walzen ergibt sich aus zwei Schritten:

1. **Auswahl** von k Walzen aus n verfügbaren Walzen (ohne Beachtung der Reihenfolge):
$$\binom{n}{k}$$
2. **Anordnung** der ausgewählten k Walzen (alle Permutationen):
$$k!$$

Die Gesamtanzahl möglicher Walzenlagen ergibt sich somit zu:

$$\text{Walzenlagen} = \binom{n}{k} \cdot k!$$

Beispiele:

- Drei aus drei Walzen: $\binom{3}{3} \cdot 3! = 1 \cdot 6 = 6$
- Drei aus fünf Walzen: $\binom{5}{3} \cdot 3! = 10 \cdot 6 = 60$
- Drei aus acht Walzen: $\binom{8}{3} \cdot 3! = 56 \cdot 6 = 336$

Wenn zusätzlich bei der Enigma M4 eine von zwei extra dünnen Walzen sowie eine von zwei möglichen Umkehrwalzen (UKW) gewählt werden kann, erweitert sich die Gesamtanzahl möglicher Walzenkonfigurationen zu:

$$\text{Permutationen Enigma M4} = \underbrace{\binom{8}{3} \cdot 3!}_{\text{Walzenlagen}} \cdot \underbrace{2}_{\text{dünne Walze}} \cdot \underbrace{2}_{\text{UKW}} = 336 \cdot 2 \cdot 2 = 1344$$

3.2 Das Zyklometer

Das **Zyklometer** ist eine von Marian Rejewski entwickelte elektromechanische Vorrichtung zur Entzifferung des Tagesschlüssels der Enigma. Rejewski erkannte einen sicherheitstechnischen Fehler im deutschen Verschlüsselungsverfahren, der als Grundlage für dieses Verfahren diente.

3.2.1 Der Fehler im deutschen Verschlüsselungsverfahren

Für den täglichen Betrieb der Enigma verwendete die deutsche Wehrmacht Codebücher, welche die Walzenlage, Walzenstellung, Ringstellung und die Steckerbrettverbindungen für jeden Tag enthielten. Diese Gesamtheit der Einstellungen wird als Tagesschlüssel bezeichnet.

Zur Sicherung einzelner Nachrichten wählte der Absender eine eigene Walzenstellung, welche mit drei Buchstaben beschrieben (z. B. `"ABC"`) und als **Spruchschlüssel** bezeichnet wurde. Dieser Spruchschlüssel wurde verdoppelt (z. B. `"ABCABC"`), mit dem Tagesschlüssel verschlüsselt, übertragen. Der Empfänger konnte durch Entschlüsselung der ersten sechs Zeichen den Spruchschlüssel rekonstruieren, die Maschine entsprechend einstellen und die Nachricht dechiffrieren.

Das doppelte Übertragen des Spruchschlüssels erzeugte allerdings eine feste Zuordnung zwischen den Buchstabenpositionen 1 und 4, 2 und 5 sowie 3 und 6 einer jeden Nachricht eines Tages. Dieser systematische Aufbau offenbarte eine wiederkehrende Struktur, die – unter kryptoanalytischem Blickwinkel – gezielt ausgewertet und als Angriffsvektor verwendet werden konnte.

3.2.2 Zyklenbildung aus Spruchschlüsseln

Diesen Umstand erkannten bereits in den 1930er-Jahren die polnischen Kryptologen als zentrale Schwachstelle der Enigma. Ihre Analyse führte zur Entwicklung einer

Methode, mit der sich aus verschlüsselten Spruchschlüsseln charakteristische Muster ableiten ließen.

Dabei wurden aus der verdoppelten Übertragung (z. B. **"ABCABC"** → **"BJEGSM"**) charakteristische Buchstabenpaare an den Positionen 1:4, 2:5 und 3:6 extrahiert – im Beispiel also **B:G**, **J:S** und **E:M**.

Diese Zuordnungen lassen sich in sogenannte **Zyklen** überführen. Betrachtet man beispielsweise das Buchstabenpaar **B:G** (aus Position 1:4), so wird analysiert, welchem Buchstaben **G** seinerseits zugeordnet ist – z. B. in einem weiteren Paar **G:F**. Daraus ergibt sich die Sequenz **B** → **G** → **F**. Findet sich nun noch ein drittes Paar **F:B**, so schließt sich der Zyklus: **B** → **G** → **F** → **B**. Es entsteht ein geschlossener Zyklus der Länge drei.

Wichtig dabei ist, dass jeder Buchstabe im Alphabet genau einmal in einer Zuordnung vorkommt. Deshalb ist eine ausreichende Anzahl an abgefangenen Spruchschlüsseln erforderlich. Es handelt sich um eine Permutation, bei der keine Buchstaben doppelt verwendet werden dürfen – weder als Ausgangs- noch als Zielbuchstabe innerhalb desselben Zyklenbaums.

Dieser Vorgang wird getrennt für jede der drei Positionspaarungen (1:4, 2:5, 3:6) durchgeführt. Für jede dieser Zuordnungsgruppen entsteht dabei eine eigene Zyklenstruktur. Man erhält somit pro Nachricht drei charakteristische Zyklenmengen, die getrennt analysiert werden.

Das Verfahren wird auf alle Buchstabenpaare angewendet, bis das gesamte Alphabet in disjunkte Zyklen eingeteilt ist. Die Gesamtheit dieser Zyklen bezeichnet man als **Charakteristik**.

Durch die doppelte Übertragung des Spruchschlüssels entstehen jeweils zwei zueinander inverse Permutationen mit identischer Zyklenstruktur. Daher tritt jede Zyklencharakteristik im Zyklometer in doppelter Ausführung auf.

3.2.3 Technische Umsetzung des Zyklometers

Um diesen Prozess zu automatisieren, entwickelte Rejewski schließlich das **Zyklometer** – ein elektromechanisches Gerät, das im Kern aus zwei hintereinandergeschalteten Enigma-Walzen besteht, wobei die zweite Walze um drei Positionen weitergedreht ist. So konnten systematisch die möglichen Zyklen für alle Walzenstellungen berechnet und dokumentiert werden.

Abbildung 4: Zyklometer Quelle: [5]

In Abbildung 4 ist ein Zyklometer zu sehen. Jeder Buchstabe ist mit einem Schalter und einer Lampe ausgestattet. Wenn der Schalter eines Buchstabens betätigt wird, leuchten alle Buchstaben auf, die zusammen mit dem gewählten Buchstaben einen Zyklus bilden. Die Anzahl der leuchtenden Buchstaben entspricht dabei der Zykluslänge.

3.2.4 Katalog der Charakteristiken

Im Katalog der Charakteristiken werden alle Zyklen der Enigma I-Permutationen bei neutraler Ringstellung und Umkehrwalze UKW-B dokumentiert. Ist die tatsächliche Ringstellung bekannt, lässt sich ihr Einfluss durch Subtraktion der Ringstellung von der Walzenstellung (modulo 26) kompensieren. So bleibt der Abgleich mit dem Katalog gültig.

Der Katalog umfasst alle möglichen Permutationen. Insgesamt enthält der er 63 420 eindeutig unterschiedliche Zyklenlängen-Charakteristiken.

Die Gesamtzahl der möglichen Enigma I-Permutationen ergibt sich aus der Anzahl der Walzenlagen, Walzenstellungen und Ringstellungen:

$$\text{Walzenlagen} = \binom{5}{3} \cdot 3! = 10 \cdot 6 = 60$$

$$\text{Walzenstellungen} = 26^3 = 17\,576$$

$$\text{Ringstellungen} = 26^3 = 17\,576$$

$$\text{Die Gesamtanzahl der Konfigurationen} = 60 \cdot 17\,576 \cdot 17\,576 \approx 18{,}5 \text{ Milliarden}$$

Da die Ringstellung in der Analyse als bekannt vorausgesetzt wird, reduziert sich die Anzahl auf:

$$60 \cdot 17\,576 = 1\,054\,560$$

Abgleich Walzenlage und Walzenstellung: Diesen über einer Million Konfigurationen stehen 63 420 verschiedene mögliche **Zyklenlängen-Charakteristiken** gegenüber. Damit reduziert sich die Anzahl potenzieller Walzenkonfigurationen pro Charakteristik auf durchschnittlich etwa $\frac{1\,054\,560}{63\,420} \approx 16{,}6$ Kandidaten. Durch den Vergleich der Zyklenlängen der verschlüsselten Nachricht mit dem Katalog lässt sich die Menge der möglichen Konfigurationen somit deutlich eingrenzen.

Analyse des Steckerbretts: Das Steckerbrett verändert nicht die Längen der Zyklen, sondern lediglich deren innere Struktur. Daher können aus den Zyklenlängen die Walzenlage und -stellung unabhängig von den Steckerbrettverbindungen ermittelt werden.

Letzere werden anhand der Zyklenstrukturen analysiert. Dabei muss jedoch zunächst die korrekte Walzenlage und -stellung bekannt und eingestellt sein. Anschließend werden die Zyklenstrukturen, wie sie ohne Steckerbrettverbindungen im Katalog vorliegen, mit denen aus den abgefangenen Nachrichten verglichen.

Bestimmung der Ringstellung: Eine direkte Bestimmung der Ringstellung mit dem Zyklometer ist nicht möglich. Stattdessen wurde die Ringstellung durch systematisches Ausprobieren aller 26^3 Kombinationen mithilfe der von Rejewski entwickelten **Grill-Methode** ermittelt. Diese Methode nutzte die Annahme, dass viele Nachrichten mit dem bekannten Präfix **"ANX"** (Leerzeichen als **"X"** geschrieben) begannen, was die Suche erheblich erleichterte [6].

3.3 Bedeutung des Zyklometers

Das Zyklometer stellte eine zentrale methodische Innovation im Kampf gegen die Verschlüsselung durch die Enigma dar. Entwickelt und eingesetzt von den polnischen Kryptographen, insbesondere Marian Rejewski, ermöglichte es eine systematische Analyse der durch die Enigma erzeugten Permutationen. Die Erstellung eines Katalogs von Zyklencharakteristiken reduzierte die enorme Komplexität der möglichen Walzenkonfigurationen und erlaubte erstmals die gezielte Bestimmung von Schlüsseln.

Diese Vorarbeiten bildeten die Grundlage für spätere Erfolge der britischen Kryptanalytiker um Alan Turing in Bletchley Park. Die Konstruktion der elektromechanischen Entschlüsselungsmaschine **Turing-Welchman-Bombe** baute direkt auf den Konzepten des Zyklometers auf und überführte sie in eine automatisierte Form. Somit markierte das Zyklometer einen entscheidenden Meilenstein auf dem Weg zur erfolgreichen Entzifferung der Enigma.

4 Technische Vorarbeiten und Codebasis

Die technische Grundlage der vorliegenden Arbeit bildet eine in mehreren Phasen weiterentwickelte C-Implementierung der Enigma-Maschine. Diese wurde im Laufe verschiedener studentischer Projekte schrittweise erweitert und verbessert.

Den Ausgangspunkt bildete eine erste Version der Simulation, die von Arif Hasanic entwickelt wurde. Diese umfasste eine funktionale Nachbildung sowohl der Enigma-Maschine als auch des Zyklometers[7].

Aufbauend auf dieser Codebasis erstellte Emanuel Schäffer im Rahmen seiner eigenen Arbeit eine erweiterte Version. Diese beinhaltete neben Verbesserungen in Typsicherheit, Speicherverwaltung und Codequalität auch die Simulation der Turing-Welchman-Bombe[8].

Für die vorliegende Arbeit wurde schließlich ein Fork dieser erweiterten Implementierung angelegt, der von Emanuel Schäffer und mir gemeinsam weiterentwickelt worden ist[9]. Diese dritte Entwicklungsstufe bildet die Grundlage für die hier vorgestellten Analyse- und Erweiterungsschritte.

5 Konzeption der Anwendung

Dieses Kapitel beschreibt die Architektur und den Betrieb der Anwendung. Zunächst wird der logische Aufbau erläutert, also wie die einzelnen Komponenten zusammenspielen. Anschließend folgt eine Beschreibung der konkreten Umsetzung auf Serverebene.

5.1 Architekturüberblick

Die Anwendung ist modular aufgebaut und besteht aus vier Hauptkomponenten:

- **Frontend:** Browserbasierte Single-Page Application (SPA).
- **Backend:** Java-Anwendung mit Spring Boot.
- **Native Bibliothek:** In C geschriebene Shared Library für kryptografische Berechnungen.
- **Datenbank:** Relationale PostgreSQL-Datenbank.

Alle Komponenten kommunizieren über klar definierte Schnittstellen. Das Frontend nutzt eine HTTP-Schnittstelle zur Interaktion mit dem Backend. Das Backend greift über das Java Foreign Function & Memory API (FFM API) direkt auf die native Bibliothek zu. Die Datenbank wird ausschließlich über JDBC angesprochen. Innerhalb des Backends sind diese Funktionalitäten modular getrennt, was die Wartbarkeit verbessert.

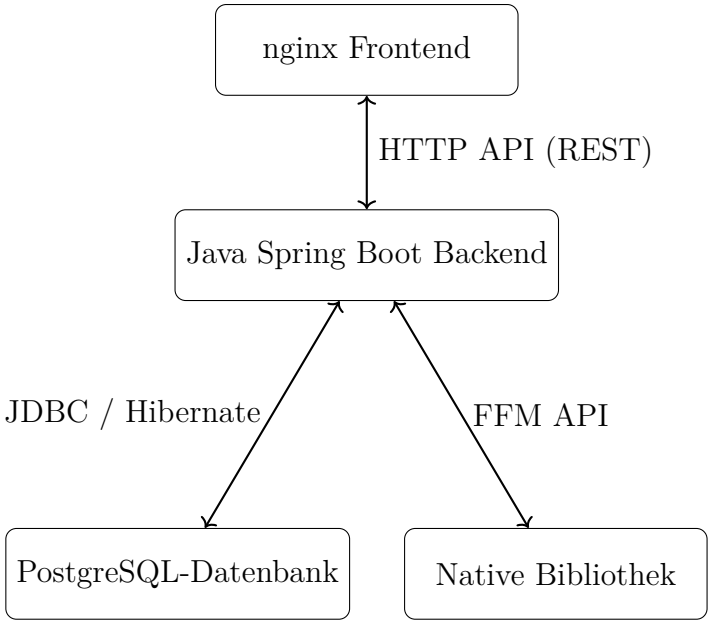


Abbildung 5: Architektur der Anwendung mit Schnittstellen

Die Anwendung unterstützt drei zentrale Funktionen: Verschlüsselung, Zyklenerzeugung und Katalogabfrage. Die ersten beiden werden über die native Bibliothek

im Backend berechnet, während die Katalogabfrage über die PostgreSQL-Datenbank erfolgt.

5.2 Betrieb und Hostingstrategie

Die Anwendung wird vollständig auf einem dedizierten Ubuntu-Server (ein Linux-basiertes Betriebssystem) der Hochschule Ravensburg-Weingarten betrieben. Die Bereitstellung erfolgt mit Hilfe etablierter Open-Source-Komponenten:

- **Frontend:** `nginx` übernimmt das Routing eingehender HTTP-Anfragen (Standard-Port 80 für Webverkehr) und dient sowohl als Reverse-Proxy als auch als Webserver für die statischen Frontend-Dateien. Konfiguration unter `/etc/nginx/sites-available/enigma`, Inhalte unter `/var/www/enigma`.
- **Backend:** Die Spring Boot-Anwendung (Version 3.4.5, kompiliert mit Java 23) liegt als ausführbare JAR unter `/opt/enigma/enigma_api.jar` vor. Sie wird als systemd-Dienst auf dem lokalen Port 8081 betrieben (`enigma.service`) und kommuniziert mit der C-Bibliothek `/opt/enigma/enigma_c/libenigma.so` sowie der PostgreSQL-Datenbank. HTTP-Kompression und Caching sind aktiviert.
- **Datenbank:** PostgreSQL 17.5 läuft lokal auf dem Server an Port 5432 und ist ausschließlich für das Backend zugänglich. Genutzt wird die Datenbank `catalogue`.
- **Netzwerkanbindung:** Der Server ist über einen hochschulinternen Reverse-Proxy erreichbar. Zugriffsbeschränkungen über IP-Filter, VPN oder Authentifizierung sind derzeit nicht implementiert.

6 Anpassungen des C-Quellcodes

Aufbauend auf der in Kapitel 4 beschriebenen Codebasis wurden im Rahmen dieser Arbeit verschiedene Erweiterungen und strukturelle Verbesserungen am bestehenden C-Quellcode vorgenommen. Ziel dieser Maßnahmen war es, die bestehende Funktionalität robuster und flexibler zu gestalten sowie eine problemlose Integration in eine moderne Systemarchitektur zu ermöglichen.

Die vorgenommenen Änderungen lassen sich im Wesentlichen in drei Bereiche unterteilen:

- strukturelle und funktionale Optimierungen,
- Erweiterungen zur Generierung von Datenbankinhalten,
- Modularisierung für die Anbindung an einen HTTP-Controller.

6.1 Strukturelle Optimierungen und Refactoring

Die internen Datenstrukturen wurden überarbeitet: Ursprünglich wurden Permutationen der Walzenlagen als `char`-Arrays mit 26 Elementen gespeichert, was wiederholte Umrechnungen zwischen Zeichen und Indizes erforderte und potenzielle Fehlerquellen eröffnete. Die Umstellung auf ein integerbasiertes `RotorPermutations`-Struct erlaubt nun direkte Indexzugriffe und vereinfacht die Zyklusberechnung.

Darüber hinaus wurde die Zyklusberechnung selbst überarbeitet. Die ursprüngliche Funktion `find_cycle_lens` wurde durch `calculate_cycle_len` ersetzt, welche speziell auf integerbasierte Permutationen ausgelegt ist und Fehlerzustände wie ungültige Indizes (z. B. `-1`) zuverlässiger erkennt und behandelt.

Auch die Speicherverwaltung wurde robuster gestaltet: Durch die Festlegung einer maximalen Anzahl von Tagesschlüsseln können Speicherüberläufe verhindert werden.

Diese Anpassungen wurden im Rahmen gemeinsamer Weiterentwicklung mit Emanuel Schäffer vorgenommen.

6.2 Erweiterungen für die Datenbankgenerierung

Zur Erstellung eines Katalogs der charakteristischen Zyklen aller möglichen Walzenkonfigurationen wurde das Programm in ein eigenständiges Kommandozeilenwerkzeug überführt. Dieses nimmt eine Zieldatei als Argument entgegen und schreibt die Ergebnisse direkt in eine Textdatei. Der anschließende Import dieser Datei in eine relationale PostgreSQL-Datenbank wird im Kapitel 8 erläutert.

Ziel dieser Erweiterung ist es, für alle Kombinationen aus Walzenlage und Walzenstellung, die charakteristischen Zyklen zu ermitteln. Das Kernprinzip des Algorithmus umfasst folgende Schritte:

1. Für jede Walzenlage werden alle Permutationen generiert (`generate_permutations`), also $\binom{5}{3} \cdot 3! = 60$.
2. Für jede Walzenlage werden alle $26^3 = 17\,576$ Walzenstellungen (von `AAA` bis `ZZZ`) durchlaufen.
3. Für jede dieser Konfigurationen werden 26 Spruchschlüssel (`"AAAAAA"` bis `"ZZZZZZ"`) verschlüsselt, wobei Ringstellung und Steckerverbindungen konstant bleiben.
4. Aus diesen verschlüsselten Schlüsseln werden die Zyklen bestimmt (`calculate_cycle_lengths`).
5. Die Ergebnisse werden in einer Textdatei persistiert (`print_whole_cycle`).

Die Ausgabe enthält $60 \cdot 17\,576 = 1\,054\,560$ Zeilen, wobei jede Zeile die folgenden Informationen umfasst:

- Drei Arrays mit Zyklenlängen,
- Die Walzenstellung im Klartext (`AAA` bis `ZZZ`),
- Die Walzenlage als Integer-Array.

```
{12,12,1,1};{12,12,1,1};{7,7,3,3,2,2,1,1};ABC;{1,2,3}
```

Die obige Zeile beschreibt eine Enigma-Konfiguration mit der Walzenlage `{1,2,3}` und der Walzenstellung `"ABC"` sowie den jeweils ermittelten Zyklenlängen. Die Wiederholung gleicher Zyklenlängen innerhalb einer Permutation ist dabei zulässig.

6.3 Modularisierung für die Anbindung an einen HTTP-Controller

Im Rahmen der Integration der Enigma-Funktionalität in eine moderne Webarchitektur wurde der C-Code so angepasst, dass zentrale Funktionen wie `enigma_encrypt`, `calculate_cycle_lengths` und `rotor_mapping` als klar modularisierte, aufrufbare Einheiten vorliegen. Diese Modularisierung ermöglicht es, die Enigma-Logik bequem und zuverlässig von einem HTTP-Controller aus aufzurufen.

Um die Wartbarkeit und Parallelisierbarkeit zu erhöhen, wurde bewusst auf den Einsatz von globalen Zuständen verzichtet, also Variablen, die außerhalb von Funktionen gemeinsam genutzt werden. Stattdessen werden alle notwendigen Daten über Funktionsparameter und lokale Variablen verwaltet, wodurch Seiteneffekte vermieden und die Integration in unterschiedliche Softwareumgebungen erleichtert wird.

Zentraler Bestandteil dieser Modularisierung ist die Einführung einer strukturierten Datenstruktur `EnigmaConfiguration`, die alle erforderlichen Parameter der Enigma-Verschlüsselung bündelt. Diese Struktur dient als Eingabe für die C-Funktionen und bildet somit eine übersichtliche Schnittstelle zwischen der nativen C-Implementierung und der höheren Anwendungsschicht, beispielsweise einem HTTP-Controller:

```c
typedef struct {
  char plugboard[26];
  char *message;
  uint8_t *rotor_positions;
  uint8_t *ring_settings;
  enum ROTOR_TYPE *rotors;
  enum ENIGMA_TYPE type;
  enum REFLECTOR_TYPE reflector;
} EnigmaConfiguration;
```

Listing 1: Struct `EnigmaConfiguration` in C

Zusätzlich wurde eine Begrenzung der Nutzeranfragen implementiert, um die Stabilität des Servers zu gewährleisten. So wird sichergestellt, dass jede Anfrage innerhalb einer kontrollierbaren Zeit beantwortet wird, was Überlastungen durch besonders rechenintensive Anfragen verhindert.

Die konkrete technische Umsetzung der Integration in höhere Softwareebenen, insbesondere die Anbindung an RESTful APIs, wird im folgenden Kapitel beschrieben.

7 Vom C-Code zur RESTful API

Aufbauend auf den im vorherigen Kapitel beschriebenen technischen Vorarbeiten und den modularisierten Anpassungen des bestehenden C-Codes wird in diesem Abschnitt gezeigt, wie aus einer ursprünglich eigenständigen Programm-Implementierung eine native Bibliothek entwickelt wurde. Diese Bibliothek dient als Grundlage für einen RESTful Backend-Server auf Basis von Java.

Der Fokus liegt dabei auf den konzeptionellen sowie technischen Herausforderungen bei der Integration der C-Funktionalität mittels der Java Foreign Function & Memory API (FFM API) und deren Einbindung in ein modernes Spring-Boot-Backend. So wird der Weg von einem monolithischen C-Programm hin zu einer flexiblen, webbasierten API beschrieben, die den Anforderungen moderner Webanwendungen gerecht wird.

Die in diesem Kapitel beschriebenen Entwicklungen sowie der zugehörige Code sind im begleitenden GitHub-Repository dokumentiert und verfügbar[10].

7.1 Von nativer C-Implementierung zur Foreign Function & Memory API

Der initiale Ansatz bestand darin, die Enigma-Funktionalität über einen in C implementierten REST-API-Server bereitzustellen. Im praktischen Einsatz zeigte sich jedoch, dass die manuelle De-/Serialisierung von JSON, die Validierung komplexer Konfigurationen sowie das Speicherzugriffsmanagement in C äußerst fehleranfällig und wartungsintensiv sind. Die experimentelle Serverimplementierung wurde daher zugunsten einer robusteren Architektur verworfen.

Alternativ wurde das Java Native Interface (JNI) evaluiert. Trotz prinzipieller Funktionalität erwies sich der damit verbundene Boilerplate-Code[1], sowie das manuelle Speicher- und Fehlerhandling und der Debugging-Aufwand als nicht zielführend. Zudem ist bei JNI häufig auch sogenannter Glue-Code[2] erforderlich.

Die Wahl fiel schließlich auf die FFM API. Sie ermöglicht typsichere, speichereffiziente und performante Interoperabilität mit nativen C-Bibliotheken – ohne zusätzliche Glue-Code-Ebenen.

[1] Standardisierter, wiederkehrender Code, der in vielen Programmen ähnlich geschrieben werden muss, aber wenig direkte Logik enthält.

[2] Code, der verschiedene Software-Komponenten oder Systeme miteinander verbindet und ihre Kommunikation ermöglicht.

7.2 Technische Integration mit der FFM API

Die FFM API bietet mit `MemorySegment`, `MethodHandle` und `FunctionDescriptor` präzise Kontrollmechanismen über Speicherlayout, Funktionssignaturen und Datenmarshalling (d.h. die Konvertierung von Java-Datenstrukturen in native Repräsentationen und umgekehrt). Zwei Funktionen wurden angebunden: `enigma_encrypt(...)` zur Verschlüsselung sowie
`manual_get_cycles_from_cyclometer(...)` zur Berechnung der Zyklenlängen.

Speicherabbildung der Struktur `EnigmaConfiguration`: Die native C-Struktur `EnigmaConfiguration` (siehe Abschnitt 6.3) wurde mittels `MemoryLayout.structLayout(...)` in Java modelliert. Zur Sicherstellung der korrekten Speicher-Alignment-Anforderungen wurde zwischen statischem Plugboard-Feld und Pointer-Feldern ein Padding-Segment eingefügt:

```
private static final MemoryLayout ENIGMA_LAYOUT = MemoryLayout.
    structLayout(
MemoryLayout.sequenceLayout(26, JAVA_BYTE).withName("plugboard"),
MemoryLayout.paddingLayout(6),
ADDRESS.withName("message"),
ADDRESS.withName("rotor_positions"),
ADDRESS.withName("ring_settings"),
ADDRESS.withName("rotors"),
JAVA_INT.withName("type"),
JAVA_INT.withName("reflector")
).withName("EnigmaConfiguration");
```

Listing 2: FFM-Deklaration der Struktur `EnigmaConfiguration`

Aufruf der Verschlüsselungsfunktion: Instanzen der Struktur werden durch eine Hilfsklasse `JavaToCFactory` erstellt. Speicherbereiche werden ASCII-kodiert allokiert, Pointer korrekt gesetzt.
Der Funktionsaufruf erfolgt über ein `MethodHandle`:

```
final var enigmaSeg = EnigmaFactory.createEnigmaSegment(enigma, arena);
final var outputSeg = arena.allocate(byteSize);
final int ret = (int) ENIGMA_ENCRYPT.invoke(enigmaSeg, outputSeg);

if (ret == 0) {
  final String outputString = outputSeg.reinterpret(byteSize).getString
    (0);
  return Optional.of(outputString);
}
```

Listing 3: Aufruf der FFM-Funktion `enigma_encrypt`

Konvertierung von String[] nach char:** Zur Rotorzyklenermittlung erwartet die C-Funktion ein nullterminiertes `char**`-Array. In Java erfolgt die Umsetzung durch ASCII-kodierte Strings, deren Adressen in einem Pointer-Array abgelegt werden. Ein NULL-Zeiger terminiert das Array:

```
public static MemorySegment allocateTerminatedASCIIArrayFromStringArray(
final String[] stringArr, final Arena arena) {
  int length = stringArr.length;
  MemorySegment segment = arena.allocate(
  ValueLayout.ADDRESS.byteSize() * (length + 1),
  ValueLayout.ADDRESS.byteSize()
  );
  for (int i = 0; i < length; i++) {
    MemorySegment stringSegment =
    allocateTerminatedASCIIFromString(stringArr[i], arena);
    segment.setAtIndex(ValueLayout.ADDRESS, i,
    MemorySegment.ofAddress(stringSegment.address()));
  }
  segment.setAtIndex(ValueLayout.ADDRESS, length, MemorySegment.NULL);
  return segment;
}
```

Listing 4: Konvertierung von String zu char**

Rückgabewert als Struktur: Cyclometer-Daten extrahieren: Die Funktion `manual_get_cycles_from_cyclometer` liefert eine `ComputedCycles`-Struktur zurück. Diese wird in Java über ein entsprechendes Layout modelliert und mit `VarHandle` feldweise gelesen:

```
private static final MemoryLayout CYCLOMETER_LAYOUT = MemoryLayout.
    structLayout(
JAVA_INT.withName("cycles_1_4_len"),
JAVA_INT.withName("cycles_2_5_len"),
JAVA_INT.withName("cycles_3_6_len"),
MemoryLayout.sequenceLayout(26, JAVA_INT).withName("cycles_1_4"),
MemoryLayout.sequenceLayout(26, JAVA_INT).withName("cycles_2_5"),
MemoryLayout.sequenceLayout(26, JAVA_INT).withName("cycles_3_6")
);
```

Listing 5: Layout des Rückgabewerts ComputedCycles

Die extrahierten Felder werden in ein dediziertes Java-Datenobjekt übertragen:

```
return new CyclometerCycles(firstToThird, secondToFourth, thirdToSixth)
    ;
```

Technische Schlussfolgerungen: Die Java FFM API erlaubt eine technisch saubere, speichersichere und performante Anbindung nativer C-Bibliotheken. Besondere

Herausforderungen bestehen in der korrekten Modellierung von Speicherstrukturen, der Zeigerarithmetik und der Datenkonversion. Durch eine strikte Modularisierung und den Einsatz von Factory-Klassen entsteht eine robuste, wartbare und portable Lösung.

7.3 Entwicklung des Backends mit Spring Boot

Nach der erfolgreichen Implementierung der nativen FFM-API-Schnittstelle stellte sich die Frage, wie diese als Webservice effizient bereitgestellt werden kann. Aufgrund der hohen Verbreitung, der umfangreichen Dokumentation sowie der ausgereiften Unterstützung für REST-APIs wurde Spring Boot als Framework gewählt.

Controller-Architektur: Zentrale Komponente ist der `EnigmaController`, der die drei Hauptendpunkte `/api/enigma`, `/api/cyclometer` und `/api/catalogue` bereitstellt. Diese Aufteilung gewährleistet eine klare Trennung der Zuständigkeiten:

- `/api/enigma`: Empfang und Validierung von Enigma-Konfigurationen und Klartext; Aufruf der nativen Verschlüsselungsfunktion über einen Service.
- `/api/cyclometer`: Verarbeitung von Anfragen zur Berechnung charakteristischer Zyklenlängen mittels manuellem Zyklometer.
- `/api/catalogue`: Schnittstelle zur Abfrage von Walzencharakteristika; Rückgabe paginierter DTOs (Data Transfer Objects).

Der folgende Codeauszug illustriert die Struktur des Controllers und den typischen Ablauf einer Anfrage:

```
@RestController
@RequestMapping("/api")
public class EnigmaController {

  @PostMapping("/enigma")
  public ResponseEntity<EnigmaResponse> enigma(@Valid
   ↪ @RequestBody EnigmaRequest req) {...}

  @PostMapping("/cyclometer")
  public ResponseEntity<CyclometerResponse> cyclometer(@Valid
   ↪ @RequestBody ManualCyclometerRequest req) {...}

  @PostMapping("/catalogue")
  public PageDTO<RotorCharacteristic> catalogue(@Valid
   ↪ @RequestBody CatalogueRequest req) {...}
}
```

Listing 6: Ausschnitt aus dem EnigmaController

Die Methodenstruktur folgt einem konsistenten Ablauf: Validierung der Eingaben, Vorbereitung der Daten für die native Schnittstelle sowie Rückgabe der Antwort oder eines Fehlerstatus. Die eigentliche Anwendungslogik ist in separate Services ausgelagert, um die Lesbarkeit und Wartbarkeit zu verbessern.[3]

Validierung: Vor der Weiterverarbeitung prüft das Backend alle eingehenden Anfragen auf Vollständigkeit und Korrektheit.

Zur Sicherstellung korrekter Konfigurationen kommen mehrere benutzerdefinierte Validatoren zum Einsatz. Das folgende Beispiel zeigt exemplarisch die Logik eines Validators, der überprüft, ob eine gültige Umkehrwalze übergeben wird:

```
1  public class EnigmaReflectorValidator
2  implements ConstraintValidator<ValidEnigmaReflector, Character>
       ↪ {
3
4    private static final List<Character> ALLOWED_REFLECTORS =
5    List.of('A', 'B', 'C', 'b', 'c');
6
7    @Override
8    public boolean isValid(Character c, ConstraintValidatorContext
       ↪  ctx) {
9      return c != null && ALLOWED_REFLECTORS.contains(c);
10   }
11 }
```

Listing 7: Validierung zulässiger Umkehrwalzen

Die Validierungslogik wird über eine Annotation, hier `@ValidEnigmaReflector`, an ein konkretes Feld der DTO-Klasse gebunden. Die Zuordnung erfolgt durch eine Klasse, die das Interface `ConstraintValidator` implementiert und die entsprechende Logik kapselt.[4]

[3] In Spring Boot (bzw. in Java allgemein) sind Annotationen Metadaten, die mit dem @-Symbol eingeleitet werden und zur Laufzeit vom Framework ausgewertet werden – etwa zur Validierung, Dependency Injection oder HTTP-Konfiguration.

[4] Ein `ConstraintValidator` implementiert die Schnittstelle `ConstraintValidator<A, T>` und definiert die Prüfung einer benutzerdefinierten Annotation (`A`) auf einem bestimmten Typ (`T`).

Die folgende Record-Klasse `Enigma` zeigt exemplarisch wie mehrere dieser Annotationen zur Validierung verwendet werden:

```
@ValidEnigmaRotorSize
public record Enigma(
@ValidEnigmaModel Integer model,
@ValidEnigmaReflector char reflector,
// ...
@ValidEnigmaPlugboard String plugboard,
@ValidEnigmaInput String input) { }
```

Listing 8: DTO-Klasse `Enigma` mit Validierungsannotationen

Alle Annotationen wie `@ValidEnigmaModel` oder `@ValidEnigmaPlugboard` gehören zur gleichen Kategorie benutzerdefinierter Validierungen. Sie erweitern den Standardmechanismus des Pakets `javax.validation`[5] und ermöglichen eine semantisch präzise Prüfung der Eingabedaten bereits beim Mapping auf die Data Transfer Objects.

So stellt das Backend sicher, dass jede Enigma-Anfrage sowohl formal korrekt als auch semantisch valide ist. Die Fehlerbehandlungsstrategie wird im Abschnitt 7.4 detailliert behandelt.

Integration der nativen Funktionen: Verschlüsselung und Zyklenberechnung werden über dedizierte Service- und Connector-Klassen gekapselt, welche die Kommunikation mit den nativen C-Funktionen übernehmen. Dadurch fokussiert sich die REST-Controller-Schicht auf Validierung und Antworterstellung. Der native Code ist über die FFM API angebunden und performant eingebunden.

Die Schnittstellenmethoden zur FFM API bleiben bewusst schlank. Für die Enigma-Verschlüsselung und die manuelle Zyklusberechnung stehen unter anderem folgende Methoden zur Verfügung:

```
public Optional<String> getOutputFromEnigma(
final org.api.restObjects.enigma.Enigma enigma);

public Optional<CyclometerCycles> getManualCyclesFromCyclometer(
final ManualCyclometerRequest req);
```

Listing 9: Schnittstellenmethoden zur nativen FFM API

Diese Methoden akzeptieren validierte, normalisierte Eingabeobjekte und liefern optional ein Ergebnis zurück, das direkt vom Controller verarbeitet wird.

[5] `javax.validation` ist ein Java-Standardpaket für die deklarative Validierung von Objekten mit Hilfe von Annotationen. Es wird häufig in Verbindung mit Frameworks wie Spring Boot verwendet.

7.4 Die Enigma-API

Die entwickelte REST-API simuliert die historische Enigma-Maschine und stellt zusätzlich Funktionen zur Simulation des Zyklometernverfahrens sowie zur Abfrage eines Katalogs möglicher Walzenkonfigurationen bereit. Die API ist derzeit ausschließlich für die Nutzung durch einen eigenen Frontend-Webserver vorgesehen, der die Funktionen in eine Webanwendung integriert.

Die API stellt drei zentrale Endpunkte unter dem Präfix `/api` bereit:

- `/api/enigma` — Verschlüsselt Texte anhand spezifischer Enigma-Konfigurationen.
- `/api/cyclometer` — Führt eine Zyklometer-Simulation basierend auf Enigma-Konfigurationen durch.
- `/api/catalogue` — Ermöglicht die Abfrage eines Katalogs möglicher Walzenlagen und Walzenstellungen anhand von Zyklusdaten.

HTTP-Methoden und Content-Type: Alle Endpunkte nutzen die HTTP-Methode `POST` und erwarten den Request-Body im JSON-Format mit dem Header `Content-Type: application/json`.

Statuscodes:

- `200 OK` — Anfrage wurde erfolgreich bearbeitet, die Antwort enthält die erwarteten Daten.
- `400 Bad Request` — Validierungsfehler bei den Eingabedaten; die Antwort enthält ein JSON-Objekt mit detaillierten Fehlermeldungen.
- `404 Not Found` — Der angefragte API-Pfad existiert nicht.

Fehlerbehandlungsstrategie: Die API implementiert eine konsistente Fehlerbehandlungsstrategie, bei der Validierungs- und Strukturfehler des Requests mit einem `400 Bad Request` beantwortet werden. Die Antwort enthält ein JSON-Objekt, das die spezifischen Fehler pro Feld beschreibt. Der Ablauf gestaltet sich wie folgt:

Fehlt der Request-Body vollständig, wird ein Fehler mit dem Hinweis auf den fehlenden Body zurückgegeben, z. B.:

```
Invalid JSON request: Required request body is missing: public org.
    ↪ springframework.http.ResponseEntity<?> ...
```

Bei einem leeren JSON-Body (`{}`) meldet die API das Fehlen notwendiger Hauptobjekte, etwa:

```
{
  "errors": {
    "enigma": "Missing required 'enigma' object."
  }
}
```

Sind die erwarteten Objekte vorhanden, jedoch leer (z. B. `"enigma": {}`), erfolgt eine feldbezogene Validierung innerhalb dieser Objekte mit detaillierten Fehlermeldungen für jedes fehlende Pflichtfeld, z. B.:

```
{
"errors": {
  "enigma.plugboard": "Must contain letters [A-Z] with an even count,
   ↪ maximum 26 letters, no separators allowed.",
  "enigma.rotors": "The first three 'rotors' must be numbered 1 to 8;
   ↪ if a fourth rotor is used, it must be 9 or 10.",
  "enigma.model": "Must be exactly 3 for Enigma machines with 3 rotors,
   ↪  and exactly 4 for machines with 4 rotors.",
  "enigma.positions": "All values must be integers between 0 and 25 (
   ↪ inclusive) and the array length must correspond to the number of
   ↪  rotors",
  "enigma.rings": "All values must be integers between 0 and 25 (
   ↪ inclusive) and the array length must correspond to the number of
   ↪  rotors",
  "enigma.reflector": "Only the values 'A', 'B', 'C', 'b', and 'c' are
   ↪ permitted.",
  "enigma.input": "Must contain only letters [A-Z] or [a-z]. Max length
   ↪  is 10000"
  }
}
```

7.4.1 Enigma (/enigma)

Der Endpunkt ermöglicht die Verschlüsselung eines Textes anhand einer spezifischen Enigma-Konfiguration. Die Antwort enthält den verschlüsselten Text.

Beispielanfrage:

```
{
"enigma": {
  "model": "3",
  "reflector": "B",
  "rotors": ["5", "2", "1"],
  "positions": ["25", "0", "4"],
  "rings": ["7", "25", "0"],
  "plugboard": "",
  "input": "Test"
  }
}
```

Listing 10: Gültige Anfrage an /enigma

Folgende Validierungsregeln gelten für die Eingabewerte:

- `enigma.plugboard`: Muss aus Buchstaben `[A-Z]` bestehen, eine gerade Anzahl (max. 26) enthalten und darf keine Trennzeichen besitzen.
- `enigma.rotors`: Die ersten drei Werte müssen zwischen 1 und 8 liegen; bei Verwendung einer vierten Walze muss diese 9 oder 10 sein.
- `enigma.model`: Muss exakt 3 für Maschinen mit 3 Walzen oder 4 für Maschinen mit 4 Walzen sein.
- `enigma.positions` und `enigma.rings`: Alle Werte sind Ganzzahlen zwischen 0 und 25; die Array-Länge muss der Anzahl der Walzen entsprechen.
- `enigma.reflector`: Nur die Werte `'A'`, `'B'`, `'C'`, `'b'` und `'c'` sind zulässig.
- `enigma.input`: Darf ausschließlich Buchstaben `[A-Z]` oder `[a-z]` enthalten. Maximale Länge beträgt 10000 Zeichen.

Antwort:

```
{
  "output": "VFIN"
}
```

Listing 11: Antwort auf /enigma-Anfrage

7.4.2 Zyklometer (/cyclometer)

Dieser Endpunkt simuliert das Zyklometer-Verfahren. Dabei werden Spruchschlüssel mit einer Enigma-Konfiguration verschlüsselt, um daraus Zyklen zu ermitteln. Die berechneten Zyklen werden zurückgegeben.

Beispielanfrage:

```
{
  "enigma": {
    "model": "3",
    "reflector": "B",
    "rotors": ["1", "2", "3"],
    "positions": ["0", "0", "0"],
    "rings": ["0", "0", "0"],
    "plugboard": "",
    "input": ""
  },
  "parameters": {
    "daily_key_count": 100,
    "manual_keys": []
  }
}
```

Listing 12: Gültige Anfrage an /cyclometer

Erforderliche Werte:

- Alle Validierungsregeln des Endpunkts `/api/enigma` gelten entsprechend.
- `parameters.daily_key_count`: Ganzzahl zwischen 0 und 1024.
- `parameters.manual_keys`: Array von Strings; jeder String muss genau 6 Zeichen lang sein, ausschließlich Großbuchstaben `A-Z` enthalten und die ersten drei Buchstaben exakt zweimal wiederholen (z. B. `ABCABC`). Ein leeres Array wird ignoriert.

Antwort:

```
{
  "computedCycles": {
    "one_to_four_permut": [9, 9, 4, 4],
    "two_to_five_permut": [6, 5, 5, 2, 2],
    "three_to_six_permut": [6, 6, 4, 4, 3, 3]
  }
}
```

Listing 13: Antwort auf /cyclometer-Anfrage

7.4.3 Katalog (/catalogue)

Der Endpunkt erlaubt die Abfrage eines Katalogs möglicher Walzenlagen und Walzenstellungen, die zu gegebenen Zyklusdaten passen. Die Antwort erfolgt paginiert mit bis zu 100 Einträgen pro Seite.

Beispielanfrage:

```
{
  "cycles": {
    "one_to_four_permut": [7, 4, 2],
    "two_to_five_permut": [4, 4, 2, 2, 1],
    "three_to_six_permut": [7, 6]
  },
  "parameters": {
    "page": 0,
    "sortBy": "rotor_order",
    "sortDir": "asc",
    "rotorOrder": [],
    "rotorPosition": []
  }
}
```

Listing 14: Gültige Anfrage an /catalogue

Erforderliche Werte:

- `parameters.page`: Ganzzahl im gültigen Bereich von 0 bis zur letzten verfügbaren Seite.
- `parameters.sortBy`: Zulässige Werte sind `"rotor_order"` oder `"rotor_position"`.
- `parameters.sortDir`: Muss `"asc"` oder `"desc"` sein.
- `cycles`: Das Objekt `cycles` ist zwingend erforderlich.

Antwort:


```
{
  "content": [
  {
    "cycles": {
      "one_to_four_permut": [7, 4, 2],
      "two_to_five_permut": [4, 4, 2, 2, 1],
      "three_to_six_permut": [7, 6]
    },
    "enigmaConfiguration": {
      "rotor_position": [15, 9, 9],
      "rotor_order": [4, 2, 5]
    }
  }
  ],
  "pageNumber": 0,
  "pageSize": 100,
  "totalElements": 1,
  "totalPages": 1
}
```


Listing 15: Antwort auf /catalogue-Anfrage

8 Erstellung des Katalogs

Zur Erstellung des Katalogs der Charakteristiken diente die bestehende C-Codebasis als Grundlage für die Datengenerierung. Der Code wurde dafür gemäß Abschnitt 6.2 entsprechend erweitert.

Die erzeugte `.txt`-Datei wird anschließend in eine relationale PostgreSQL-Datenbank importiert.

8.1 Import in die PostgreSQL-Datenbank und Anpassung

Nach der Erzeugung der Rohdaten erfolgt der Import in eine PostgreSQL-Datenbank mittels eines separaten Scripts. Zur besseren Strukturierung wird hierfür ein eigenes Schema `cycles` verwendet. Die zentrale Tabelle `catalogue` speichert jede Enigma-Konfiguration als einzelnen Datensatz mit folgender Struktur:

Spalte	Beschreibung
`1st_rotor_cycles`	Integer-Array mit den Zyklenlängen, z. B. `{12,12,1,1}`
`2nd_rotor_cycles`	Integer-Array mit den Zyklenlängen, z. B. `{12,12,1,1}`
`3rd_rotor_cycles`	Integer-Array mit den Zyklenl., z. B. `{7,7,3,3,2,2,1,1}`
`rotor_order`	Zeichenkette der Walzenlage, z. B. `{1,2,3}`
`rotor_positions`	Integer-Array mit der Walzenstellung, z. B. `"ABC"`

Zur Sicherstellung der Eindeutigkeit wurde ein zusammengesetzter Primärschlüssel aus `rotor_order` und `rotor_positions` definiert.

Im Anschluss an den initialen Import wurden Struktur und Datenbankdesign optimiert, um die Effizienz und Benutzerfreundlichkeit zu erhöhen.

Ein Problem der ursprünglichen Datenstruktur war die Verdoppelung der Zyklenlängen in den Arrays `#_rotor_cycles`. Diese Duplikate enthalten keine zusätzlichen Informationen und verkomplizieren die Abfragen unnötig. Daher wurde eine Bereinigung vorgenommen, sodass in den Arrays nur noch jeweils eindeutige Zyklenlängen gespeichert werden.

Darüber hinaus wurde das Format der Walzenstellung überarbeitet. Anstelle der ursprünglichen dreibuchstabigen Klartextdarstellung (z. B. `"ABC"`) wird das Feld `rotor_positions` nun als Integer-Array geführt, wobei jeder Buchstabe als Wert zwischen `0` (`A`) und `25` (`Z`) kodiert wird. Diese Änderung ermöglicht eine einheitliche und effizientere Verarbeitung, da der Zugriff auf einzelne Walzenpositionen als numerische Indizes erfolgt.

Die aktualisierte Tabellenstruktur von `catalogue` ist daher wie folgt definiert:

Spalte	Beschreibung
`1st_rotor_cycles`	Int-Array Zykl.l., z. B. `{12,1}` statt `{12,12,1,1}`
`2nd_rotor_cycles`	Int-Array Zykl.l., z. B. `{12,1}` statt `{12,12,1,1}`
`3rd_rotor_cycles`	Int-Array Zykl.l., z. B. `{7,3,2,1}`
`rotor_order`	Int-Array Walzenlg., z. B. `{1,2,3}`
`rotor_positions`	Int-Array Walzenstl., z. B. `{0,1,2}` statt `"ABC"`

8.2 Normalisierung und Datenbankstruktur

Nach der initialen Erfassung und Speicherung der Rohdaten in der Tabelle `catalogue` sowie den ersten Abfragen zur Validierung der Datenbasis wurde die Datenbankstruktur weiter optimiert und in eine normalisierte Form überführt. Ziel war es die Performance bei Abfragen zu verbessern.

Hierzu wurde die ursprüngliche Tabelle in zwei logisch getrennte Einheiten zerlegt: `public.config` und `public.permutation_counts`.

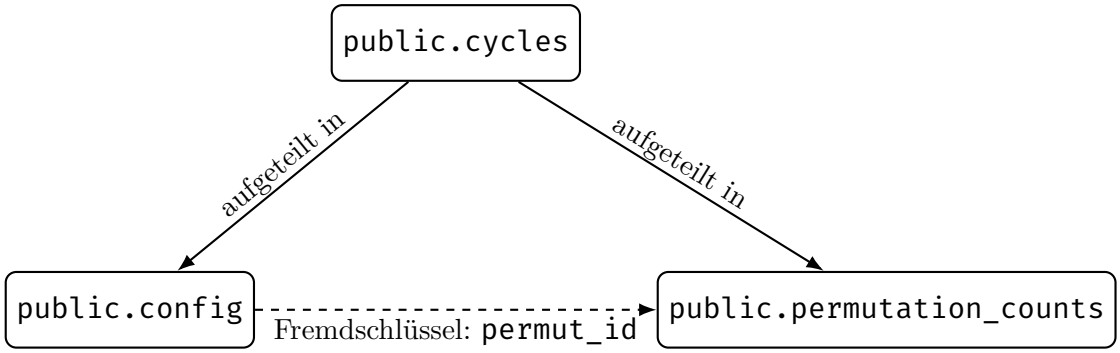


Abbildung 6: ER-Diagramm zur Umstrukturierung der Tabelle `public.cycles`

Die Aufteilung erfolgte nach folgendem Prinzip:

- In `public.config` werden alle eindeutigen Kombinationen aus Walzenlage (`rotor_order`) und Walzenstellung (`rotor_positions`) gespeichert. Jeder Eintrag verweist mittels eines Fremdschlüssels (`permut_id`) auf die zugehörigen Permutationsdaten.

- Die Permutationsdaten selbst, bestehend aus drei Arrays mit Zyklenlängen, werden in die separate Tabelle `public.permutation_counts` ausgelagert. Durch die Auslagerung der Permutationsdaten reduziert sich die Tabellengröße von 1 054 560 auf lediglich 63 420. Zusätzlich werden dort Felder für die Zählung einzelner Zykluswerte ergänzt, um präzise Abfragen effizient zu ermöglichen.

Diese Struktur entspricht dem Prinzip der dritten Normalform (3NF), bei der funktionale Abhängigkeiten so getrennt werden, dass Redundanzen minimiert und Datenkonsistenz sichergestellt wird. Dadurch werden identische Permutationen nicht mehrfach gespeichert. Die eindeutige Identifikation jeder Konfiguration erfolgt über einen Primärschlüssel in `public.config`, der sich aus der Kombination aus Walzenstellung und Walzenlage ergibt. Die Permutationsdaten in `permutation_counts` werden dabei nur einmal gespeichert und können von mehreren Konfigurationen referenziert werden.

Zur Erzeugung dieser finalen Datenbankstruktur wurde ein separates Konsolidierungsskript entwickelt, das aus der Tabelle `catalogue` eindeutige Permutationsdatensätze extrahiert und mit den jeweiligen Konfigurationen verknüpft.

Datenbankschema:

Tabelle: `public.config`

Spalte	Beschreibung
`config_id`	Primärschlüssel (serial), nach `rotor_order` sortiert
`rotor_order`	Integer-Array, Walzenlage (z. B. `{3,2,1}`)
`rotor_positions`	Integer-Array, Walzenstellung (z. B. `{2,1,0}`)
`permut_id`	Fremdschlüssel auf `permutation_counts`
`position_id`	Sortierschlüssel nach `rotor_positions` sortiert

Tabelle: `public.permutation_counts`

Spalte	Beschreibung
`permut_id`	Primärschlüssel (serial)
`one_to_four`	Integer-Array der Zyklen
`two_to_five`	Integer-Array der Zyklen
`three_to_six`	Integer-Array der Zyklen
`one_1` bis `one_7`	Integer-Werte: Häufigkeiten spezifischer Zyklenlängen
`two_1` bis `two_7`	Integer-Werte: Häufigkeiten spezifischer Zyklenlängen
`three_1` bis `three_7`	Integer-Werte: Häufigkeiten spezifischer Zyklenlängen

8.3 Optimierungsstrategien für Abfrageperformance

Zur Steigerung der Abfrageeffizienz wurden mehrere komplementäre Maßnahmen implementiert:

- **Effiziente Filterung:** Die initiale Filterung auf den Array-Feldern `xxx_permut` erfolgt durch den Einsatz eines GIN-Index in Kombination mit dem Containment-Operator `@>`. Diese Synergie ermöglicht eine grobe, aber sehr performante Einschränkung der Ergebnismenge, indem schnell überprüft wird, ob die Daten die gesuchten Elemente enthalten.

- **Mehrstufiges Filterkonzept:** Das Filterverfahren ist in mehrere Stufen unterteilt: Zunächst erfolgt die grobe Filterung durch den GIN-Index und den Containment-Operator. Anschließend wird eine präzisere Filterung anhand der Zählspalten vorgenommen. Abschließend wird ein Join mit der Tabelle `public.config` über den Fremdschlüssel `permut_id` durchgeführt. Diese Reihenfolge minimiert unnötige Joins mit großen Datensätzen und reduziert somit die Gesamtverarbeitungszeit signifikant.

- **Zählspalten:** Zur effizienten Umsetzung häufiger Filterbedingungen (z. B. „mindestens zwei Einsen im Zyklus“) wurden die Felder `one_1` bis `one_7`, `two_1` bis `two_7` sowie `three_1` bis `three_7` eingeführt. Komplexe Ausdrücke wie

```
AND cardinality(array_positions(p.three_to_six_permut, 1))
    ↪ >= 2

```

werden dadurch vereinfacht und performant ausgeführt.

Die vorgelagerte Filterung mittels `@>`-Operator stellt sicher, dass ein Wert mindestens einmal im Array enthalten ist, wodurch eine effiziente erste Eingrenzung der Ergebnismenge erfolgt. Die Zählspalten ermöglichen eine präzise und performante Filterung bei Anforderungen auf mehrfache Vorkommen (Kardinalität > 1).

Da die Summe der Zyklenlängen pro Konfiguration maximal 13 beträgt, können Werte ab 7 höchstens einmal vorkommen; Zählspalten für diese Fälle wurden daher nicht erstellt.

- **Stabilisierung des Query-Plans:** Der PostgreSQL-Query-Planner konnte die Selektivität von Funktionen wie `array_positions` nur unzureichend abschätzen, was häufig zu ineffizienten Plänen und vollständigen Tabellenscans führte. Die Einführung von Zählspalten verbesserte die Kostenabschätzung und führte zu stabileren, schnellen Ausführungsplänen.

Da die Datenbank statisch blieb und keine weiteren Inserts oder Updates erfolgten, konnten diese Optimierungen ohne Kompromisse bei den Schreiboperationen durchgeführt werden. Die Maßnahmen resultierten in einer hochperformanten und planungssicheren Abfrageinfrastruktur.

8.4 Performance-Optimierungen bei der Datenbereitstellung

Erste Lasttests mit dem Tool `Postman` zeigten, dass bei Abfragen mit wenigen oder keinen Zyklusfiltern die Antwortgrößen die Limits erheblich überschritten. Eine Beispielabfrage ohne Zyklusfilter ergab eine unkomprimierte Antwortgröße von 172,24 MB und lag somit deutlich über dem Postman-Limit von 50 MB.

Durch Aktivierung der HTTP-Kompression in den `application.properties` reduzierte sich die Antwortgröße auf 8,37 MB – weiterhin zu groß für effiziente Datenübertragung.

Zur Bewältigung dieser Herausforderungen wurden folgende Maßnahmen eingeführt:

- **Pagination:** Einführung einer Seitennummerierung mit 100 Einträgen pro Seite zur Begrenzung der Datenmenge pro Anfrage.
- **Gesamtanzahl der Treffer:** Übermittlung der Gesamtanzahl der Suchergebnisse zur Verbesserung der Nutzerfreundlichkeit und zur Ermöglichung effizienter Seitennavigation.
- **HTTP-Kompression:** Aktivierung der Gzip-Kompression für HTTP-Antworten, um die Datenübertragung weiter zu reduzieren.
- **Caching:** Implementierung eines Zwischenspeichers für häufig abgefragte Datensätze zur Minimierung wiederholter Datenbankzugriffe und zur Beschleunigung der Antwortzeiten.

Die Kombination aus Pagination und Trefferanzahl ermöglichte sowohl die Begrenzung der Datenmenge pro Anfrage als auch eine verbesserte Übersichtlichkeit über die Gesamtergebnisse. Dadurch sank die durchschnittliche Antwortgröße auf 1,18 KB – eine Reduktion um den Faktor 7.093 gegenüber der komprimierten sowie um den Faktor 150.008 gegenüber der unkomprimierten Antwort.

8.5 Implementierung der Datenbankabfrage in Java

Die Datenbankabfrage erfolgt paginiert und ist in zwei separate SQL-Statements unterteilt:

- **Count-Query** zur Ermittlung der Gesamtanzahl übereinstimmender Datensätze, also einer Abfrage, die nur die Anzahl der Treffer zurückgibt, ohne die eigentlichen Daten zu laden.
- **Daten-Query** zur Abfrage der Inhalte einer spezifischen Seite, welche die tatsächlichen Datensätze für die gewünschte Seitennummer liefert.

Diese Trennung ermöglicht den Einsatz unterschiedlicher **Cache-Keys**, das sind eindeutige Schlüssel zur Identifikation von zwischengespeicherten Ergebnissen. Besonders die Count-Query profitiert davon, da sie unabhängig von Sortierkriterien und Seitennummer ist und somit effizient wiederverwendet werden kann.

Service-Schicht: Die **Service-Schicht**, welche die Geschäftslogik kapselt, erzeugt aus den Eingabeparametern ein `Pageable`-Objekt. Dieses Objekt fasst alle nötigen Angaben zur Seitennummer, Seitengröße und Sortierung zusammen und vereinfacht so die Handhabung der Paginierung.

Für das **Caching** kommen zwei spezialisierte Klassen zum Einsatz:

- `RotorCharacteristicCacheKey` für die paginierte Datenabfrage.
- `RotorCharacteristicCountCacheKey` für die pagination-unabhängige Zählabfrage.

Im Anschluss werden zwei Repository-Methoden aufgerufen:
`countRotorCharacteristics(...)` zur Ermittlung der Trefferanzahl
`findRotorCharacteristic(...)` zur Datenabfrage mit Pagination

Caching-Strategie: Das Caching basiert auf `Caffeine`, einer Java-Bibliothek zur effizienten Umsetzung von Caches mit festgelegtem Speicherlimit von 10.000 Einträgen.

- Der Cache-Key für die Datenabfrage berücksichtigt sowohl Filterkriterien als auch Pagination (Seite, Sortierung).
- Der Cache-Key für die Count-Abfrage ignoriert Pagination und basiert ausschließlich auf den Filterparametern.

Diese Trennung erlaubt es z. B., mehrere Seiten mit identischem Filter ohne erneute Zählabfrage abzurufen.

Repository-Schicht: Die **Repository-Schicht** übernimmt den direkten Datenzugriff auf die Datenbank und generiert native SQL-Statements. Dabei werden, abhängig von den übergebenen Parametern, `WHERE`- und `ORDER BY`-Klauseln dynamisch zur Laufzeit ergänzt oder weggelassen.

```
@Cacheable(value = "rotorCharacteristics", key = "#cacheKey")
public Page<RotorCharacteristic> findRotorCharacteristic(...) {
  String sql = buildDynamicQuery(...);
  ...
  return new PageImpl<>(results, pageable, totalCount);
}
```

Listing 16: Datenabfrage-Methode (vereinfacht)

Dynamischer SQL-Aufbau: Die SQL-Abfrage wird modular und zur Laufzeit erzeugt. Nur aktiv gesetzte Filterparameter (z. B. `firstCycle`) führen zur Generierung entsprechender SQL-Klauseln. Dies minimiert die Komplexität und erhöht die Effizienz des Query-Plans, da nur notwendige Filterbedingungen eingebunden werden.

```
StringBuilder sql = new StringBuilder();
sql.append("WITH rough_filtered AS (SELECT * FROM
    ↪ permutation_counts p ");

if (!firstCycle.isEmpty()) {
  sql.append("WHERE p.one_to_four_permut @> cast(:firstCycle AS
    ↪ integer[]) ");
  ...
}
sql.append("), exact_filtered AS (SELECT * FROM rough_filtered "
    ↪ );
if (hasExactFilter) sql.append("WHERE ...");
sql.append(") SELECT ef.*, c.* FROM exact_filtered ef JOIN
    ↪ config c ...");
sql.append("ORDER BY c.rotor_id ASC LIMIT :limit OFFSET :offset"
    ↪ );
```

Listing 17: Query-Zusammensetzung (Pseudocode)

9 Entwicklung des Frontends

Das Frontend der Anwendung wurde mit dem modernen JavaScript-Framework Vue.js (Version 3) entwickelt. Ziel war es, eine performante und benutzerfreundliche Oberfläche zu gestalten, welche die komplexen Backend-Funktionalitäten abstrahiert und eine intuitive Bedienung ermöglicht. Die Entwicklung folgt einem komponentenbasierten Ansatz, der Modularität und Wiederverwendbarkeit fördert. Die Kommunikation mit dem Spring Boot Backend erfolgt asynchron über HTTP unter Verwendung des Axios-Clients [11].

9.1 Projektinitialisierung

Die Initialisierung des Projekts erfolgt mithilfe der offiziellen Vue CLI, wie aus der Datei `package.json` hervorgeht. Die Anwendung verwendet das Single-File-Component-Prinzip, bei dem HTML, CSS und JavaScript in einer Datei gekapselt sind.

9.2 CORS und Proxy-Konfiguration

CORS (Cross-Origin Resource Sharing) ist eine Sicherheitseinschränkung moderner Browser, die verhindert, dass Webanwendungen unerlaubt auf Ressourcen anderer Domains, Ports oder Protokolle zugreifen. Dabei blockiert der Browser standardmäßig API-Anfragen an eine andere Herkunft, sofern diese nicht vom Zielserver explizit erlaubt werden.

Da das Frontend (typischerweise auf `localhost:8080` während der Entwicklung) und das Backend (z. B. auf `localhost:8081`) auf unterschiedlichen Ports laufen, würde der Browser bei direkten API-Aufrufen CORS-Fehler auslösen.

Um diese Problematik zu umgehen, werden zwei Maßnahmen ergriffen:

- **Proxy-Konfiguration im Entwicklungsserver:** Der Vue-Entwicklungsserver leitet API-Anfragen über eine Proxy-Konfiguration (`vue.config.js`) an das Backend weiter. Dadurch erscheinen die Anfragen für den Browser, als kämen sie vom gleichen Ursprung, was CORS-Fehler während der lokalen Entwicklung vermeidet.

- **CORS-Konfiguration im Backend:** Im produktiven Betrieb oder bei direktem Zugriff auf das Backend müssen die HTTP-Antworten entsprechende der enthalten, die den Zugriff vom Frontend erlauben.

Preflight-Anfragen (OPTIONS) und HTTP-Methoden: Besondere Bedeutung haben sogenannte „Preflight“-Anfragen, die der Browser automatisch vor der eigentlichen API-Anfrage ausführt, wenn spezielle Header oder HTTP-Methoden verwendet werden. Zum Beispiel sendet der Browser vor einer `POST`-Anfrage mit dem Header `Content-Type: application/json` eine `OPTIONS`-Anfrage, um die Zustimmung des Servers einzuholen.

Der Server muss diese `OPTIONS` -Anfrage akzeptieren und bestätigen, welche Methoden und Header erlaubt sind. Dies ist notwendig, weil der `Content-Type application/json` nicht zu den sogenannten „einfachen“ Headern zählt.

Beispielhafte CORS-Konfiguration im Spring Boot Backend: Die folgende Methode zeigt die Konfiguration, welche `POST` und `OPTIONS`-Anfragen an alle Pfade mit dem Präfix `/api` erlaubt. Zudem wird der Header `Content-Type` als erlaubter Header spezifiziert:

```
1 public WebMvcConfigurer corsConfigurer() {
2   return registry -> registry.addMapping("/api/**")
3   .allowedOrigins("https://enigma-zyklometer.rwu.de")  //
     ↪ Produktionsdomain
4   .allowedMethods("POST", "OPTIONS")
5   .allowedHeaders("Content-Type");
6 }
```

Listing 18: Spring Boot CORS-Konfiguration

Während der Entwicklung war zusätzlich `http://localhost:8080` als erlaubte Herkunft konfiguriert, um lokale Tests zu ermöglichen.

nginx-Server: CORS-Header für Produktion: Auf Serverseite sorgt der `nginx` -Server für das Ausliefern der statischen Frontend-Dateien und die Weiterleitung der API-Anfragen an das Backend. Dabei werden ebenfalls notwendige CORS-Header für die Produktion gesetzt um den Zugriff vom Frontend auf die API zu erlauben:

```
server {
  listen 80;
  server_name enigma-zyklometer.rwu.de;
  root /var/www/enigma;
  index index.html;

  location / {
    try_files $uri $uri/ /index.html;
  }
  location /api/ {
    proxy_pass http://localhost:8081/;

    # CORS-Header für reguläre und OPTIONS-Anfragen
    add_header Access-Control-Allow-Origin "https://enigma-
    ↪ zyklometer.rwu.de" always;
    add_header Access-Control-Allow-Methods "POST, OPTIONS"
    ↪ always;
    add_header Access-Control-Allow-Headers "Authorization,
    ↪ Content-Type" always;

    if ($request_method = OPTIONS) {
      return 204;
    }
  }
}
```

Listing 19: Konfiguration des `nginx`-Servers

Diese Konfiguration erlaubt dem Browser, `POST`-Anfragen mit `Content-Type: application/json` an die API zu senden ohne CORS-Fehler zu verursachen.

9.3 Entwicklungsworkflow, Build und Deployment

Für die lokale Entwicklung wird der Befehl `npm run serve` verwendet, der einen Entwicklungsserver auf `localhost:8080` startet. Dieser Entwicklungsserver unterstützt Hot-Reload, sodass Änderungen am Quellcode beim Speichern sofort im Browser sichtbar sind.

Sobald die Entwicklung abgeschlossen ist, wird das Projekt mit dem Befehl `npm run build` für die Produktionsumgebung gebaut. Dabei werden die Anwendung optimiert, Dateien minifiziert und in einem statischen HTML/CSS/JS-Bundle im Ordner `dist/` zusammengefasst. Dieses Bundle kann anschließend auf einem Webserver, beispielsweise einem `nginx`-Server, bereitgestellt werden.

9.4 Projektstruktur

Die Projektstruktur folgt einer gängigen Vue 3 Konvention und ist wie folgt gegliedert:

```
src/
|-- App.vue
|-- assets/
|-- components/
|   |-- LabeledPlugboard.vue
|   |-- MultiSelect.vue
|   |-- MultiSelectWithCheckbox.vue
|   |-- Plugboard.vue
|   |-- ReverseMultiSelect.vue
|   |-- SubmitButton.vue
|   |-- ToggleSwitch.vue
|   \-- TooltipLabel.vue
|-- main.js
|-- router/
|   \-- index.js
|-- services/
|   \-- Enigma/
|       |-- BackendEnigma.js
|       \-- Enigma.js
\-- views/
|-- CyclometerView.vue
|-- EnigmaRequestView.vue
\-- InstructionsView.vue
```

Listing 20: Projektstruktur

Die Einstiegskomponente ist `App.vue`, welche das grundlegende Layout definiert. Der Einstiegspunkt `main.js` bindet Vue, den Router und die Root-Komponente zusammen. Jede Hauptfunktionalität (z. B. Enigma-Anfrage, Zyklometer-Abfrage) ist als eigene View unter `src/views` realisiert.

9.5 Verwendung der Composition API und Komponentenentwicklung

Zur Strukturierung der Komponentenlogik kommt in diesem Projekt die Composition API von Vue 3 zum Einsatz. Sie erlaubt es, die Logik nach Funktionen zu gruppieren und somit besser wiederverwendbar und testbar zu gestalten. Im Gegensatz zur älteren Options API, bei der verschiedene Optionen (Daten, Methoden, Computed Properties etc.) in einem Objekt zusammengefasst werden, können mit der Composition API verwandte Funktionen und Zustände enger zusammengehalten werden.

Dies führt zu einer klareren Trennung von Anliegen und einer übersichtlicheren

Codebasis, insbesondere bei komplexeren Komponenten mit mehreren Funktionalitäten.

Ein zentrales Prinzip der Frontendentwicklung ist die Verwendung wiederverwendbarer und modularer Komponenten. Vue.js ermöglicht mit seinem komponentenbasierten Ansatz, einzelne Funktionseinheiten zu kapseln und flexibel in verschiedenen Kontexten einzusetzen. So stellt beispielsweise die Komponente `ReverseMultiSelect.vue`, kombiniert mit `TooltipLabel.vue` und einer internen `ToggleSwitch`-Logik, eine vielseitige Lösung für Benutzereingaben der Enigma-Konfiguration dar.

9.5.1 Designprinzipien: Wiederverwendbarkeit und Modularität

Die Logik, Darstellung und Interaktion werden klar voneinander getrennt. Die Logik der Mehrfachauswahl ist beispielsweise in einer eigenen `ReverseMultiSelect` -Struktur gekapselt, sodass diese unabhängig von der erweiterten Checkbox-Logik wiederverwendet werden kann. Die Komponente `TooltipLabel.vue` ist als eigenständige Benutzeroberflächen-Komponente für beschriftete Tooltips ausgelegt.

Die Abbildung 7 zeigt exemplarisch, wie die Komponenten- und View-Hierarchie der Anwendung in der Praxis zusammenspielen kann.

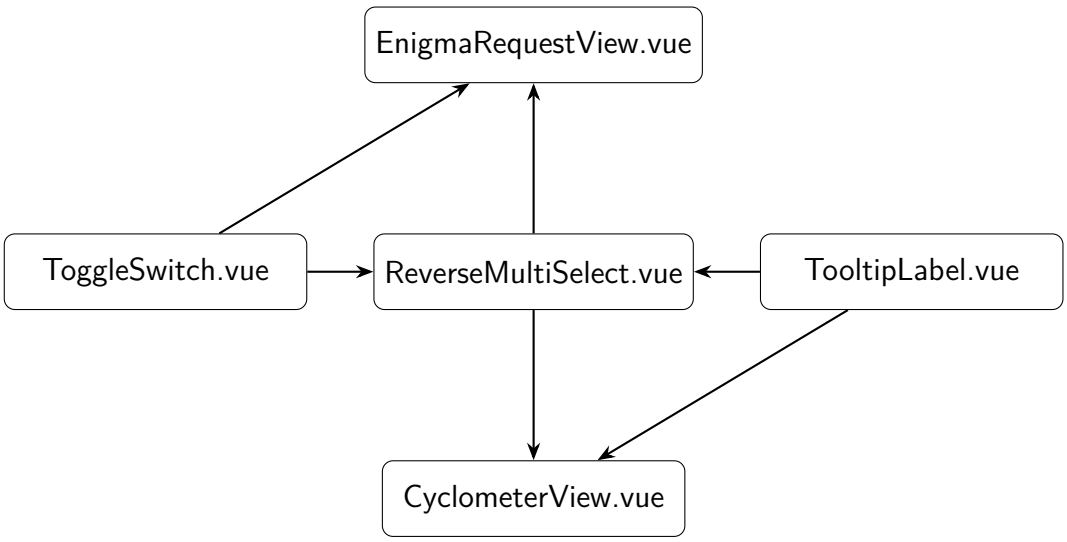


Abbildung 7: Die Komponenten befinden sich mittig, oben bzw. unten sind die jeweiligen Views.

Durch solche Strukturen, welche hier exemplarisch gezeigt werden, entsteht eine hohe Wiederverwendbarkeit und Erweiterbarkeit. Neue Anwendungsfälle lassen sich mit minimalem Anpassungsaufwand umsetzen.

9.5.2 Beispiel: `ReverseMultiSelect.vue`

Die Komponente `ReverseMultiSelect.vue` ermöglicht es, eine oder mehrere Auswahlfelder (Dropdowns) dynamisch zu rendern. Dabei berücksichtigt sie unterschiedliche Anforderungen:

- dynamische Anzahl von `<select>`-Feldern über `selectCount`,
- `<select>`-Felder werden von rechts nach links aufgereiht, um die physische Enigma-Maschine zu imitieren.
- gemeinsame oder individuelle Optionslisten,
- gezielte Deaktivierung einzelner Felder über `disabledIndexes`,
- globale Deaktivierung durch das Attribut `disabled`,
- optionaler Aktivierungs-/Deaktivierungs-Schalter (Checkbox) über `showCheckbox`.

Zudem wird durch die Einbindung der Komponente `TooltipLabel.vue` eine benutzerfreundliche Hilfsanzeige über Tooltips realisiert. Dabei kommen zwei zentrale Konzepte des Vue-Frameworks zum Einsatz:

Reaktive Referenzen mittels **`ref()`**[6] ermöglichen eine bidirektionale Bindung zwischen Datenmodell und Benutzeroberfläche, sodass Änderungen in beide Richtungen automatisch erkannt und übernommen werden.

Ergänzend dazu liefern berechnete Eigenschaften über **`computed`** abgeleitete Werte, die stets aktuell gehalten werden, sobald sich eine ihrer abhängigen Datenquellen ändert.

Der übergebene Wert `arrayOptions` wird intern als Array behandelt, um Einzelauswahlen und Mehrfachauswahlen einheitlich und flexibel verarbeiten zu können.

[6] `ref()` stammt aus der Composition API und erzeugt eine sogenannte reaktive Referenz, die Vue automatisch mit dem DOM synchronisiert

Die folgende Konfiguration demonstriert die Vielseitigkeit der Komponente `ReverseMultiSelect`:

```
<ReverseMultiSelect
v-model:single="settings.enigma.reflector"
:singleOptions="reflectors"
v-model:array="settings.enigma.rotors"
:arrayOptions="rotorOptions"
label="Walzenlage:"
info="Hier wird die Reihenfolge..."
:isSingleEnabled="false"
/>
```

Listing 21: Verwendung von ReverseMultiSelect in Vue

9.6 Routing mit Vue Router

Der Router, definiert in `router/index.js`, nutzt das History-API[7], um die Navigation innerhalb der Single-Page Application (SPA) zu steuern. Er sorgt dafür, dass beim Wechsel zwischen verschiedenen Ansichten keine vollständige Seitenaktualisierung erfolgt, sondern nur die entsprechenden Komponenten dynamisch geladen und angezeigt werden. So entsteht schnelles Benutzererlebnis mit sauberen URLs.

Die wichtigsten Routen sind:

```
import { createRouter, createWebHistory } from "vue-router";
import EnigmaRequestView from "@/views/EnigmaRequestView.vue";
import CyclometerView from "@/views/CyclometerView.vue";

const routes = [
{ path: "/", name: "enigma", component: EnigmaRequestView },
{ path: "/enigma", redirect: "/" },
{ path: "/cyclometer", name: "cyclometer", component:
    ↪ CyclometerView },
];

const router = createRouter({
  history: createWebHistory(process.env.BASE_URL),
  routes,
});

export default router;
```

Listing 22: Vue Router Konfiguration

[7]Ein Web-API zur Steuerung des Browserverlaufs, das saubere URLs ohne #-Zeichen erlaubt und clientseitige Navigation ohne Seitenreload ermöglicht

9.7 Kommunikation mit dem Backend

Für die Kommunikation zwischen Frontend und Backend wird die JavaScript-Bibliothek `axios` verwendet. Eine zentrale Instanz wird in `Enigma.js` mit dem Basis-URL-Präfix `/api` erstellt.

Die Datei `BackendEnigma.js` kapselt die Logik für konkrete API-Aufrufe. Dort wird auf die zentrale `axios`-Instanz zurückgegriffen.

Das folgende Beispiel zeigt die Methode zur Verschlüsselung einer Eingabe über den Endpunkt `/enigma`:

```
import axios from "axios";
  // zentrale Axios-Instanz
const api = axios.create({ baseURL: '/api' });
  // Methode zur Verschlüsselung
export function getEncryption(data) {
  return api.post("/enigma", data, {
    headers: { 'Content-Type': 'application/json' }
  });
}
```

Listing 23: API-Aufruf in `BackendEnigma.js`

9.7.1 Fehlerbehandlung Kommunikation mit dem Backend

Für die Fehlerbehandlung im Frontend wird die Bibliothek `toast` verwendet, um Benutzern eine unmittelbare Rückmeldung über Fehlerzustände zu geben. Insbesondere werden HTTP-Fehlercodes 400 und 500 abgefangen.

Bei einem Fehler 400 gibt die Enigma-API hilfreiche Fehlermeldungen zurück, die direkt an den Nutzer weitergeleitet werden. Dies verbessert die Benutzererfahrung, da präzise Hinweise zum Problem angezeigt werden.

Das folgende Beispiel zeigt den Umgang mit Fehlern beim asynchronen Aufruf einer Verschlüsselungsfunktion:

```
const Encrypt = async (data) => {
  try {
    // Anfrage ans Backend
    const response = await BackendEnigma.getEncryption(data);
    // --- gekürzt: Ausgabe-Verarbeitung entfernt ---
  } catch (error) {
    // Fehlerobjekt aus der Antwort auslesen (falls vorhanden)
    const errors = error?.response?.data?.errors;
    if (errors) {
      // Alle Fehlernachrichten anzeigen
      Object.values(errors).forEach(msg => toast.error(msg));
    } else {
      // Allgemeiner Fehler-Fallback
      toast.error(error.message || "Unbekannter Fehler");
    }
  }
};
```

Listing 24: Fehlerbehandlung mit Toast bei Backend-Anfragen in Vue 3

Dieses Muster ermöglicht es, Fehler strukturiert und nutzerfreundlich anzuzeigen und erleichtert die Fehlerdiagnose während der Nutzung der Anwendung.

10 Beschreibung der Webanwendung

Die vorliegende Webanwendung dient der Simulation, Analyse und Verschlüsselung historischer Enigma-Maschinen. Ziel ist es, Nutzern sowohl eine intuitive Oberfläche zur Konfiguration und Verschlüsselung als auch eine analytische Komponente zur Untersuchung der charakteristischen Zyklen der Enigma zu bieten. Dabei werden komplexe kryptografische Vorgänge durch eine klare und benutzerfreundliche Gestaltung zugänglich gemacht.[12]

Darstellung der Enigma-Konfiguration in der Weboberfläche

Die Enigma-API nutzt für Walzen- und Ringstellungen intern einen numerischen Wertebereich von 0 bis 25, wobei **A** dem Wert 0 und **Z** dem Wert 25 entspricht. Auf der Benutzeroberfläche der Webanwendung wird dieser Bereich hingegen um eins erhöht dargestellt, sodass beispielsweise **A** als 1 und **Z** als 26 erscheint.

Historisch wurde bei der Enigma die Walzenstellung ausschließlich durch Buchstaben und die Ringstellung ausschließlich durch Zahlen angegeben. Die Webanwendung zeigt jedoch bewusst beide Darstellungen kombiniert an (z. B. **G** (7)), um die Beziehung zwischen den beiden Positionierungen anschaulich zu machen.

So wird etwa die Kombination Walzenstellung **G** mit Ringstellung **G** intern genauso interpretiert wie Walzenstellung **A** mit Ringstellung **A**, solange keine Einkerbung überlaufen wurde. Die durch die Ringstellung bewirkte Verschiebung der Einkerbung bleibt dabei zentral für das Verhalten der Maschine.

Da Begriffe wie Walzenstellung, Ringstellung und Einkerbung bereits ausführlich in einer früheren Arbeit erläutert wurden, wird hier lediglich auf deren Relevanz im Kontext der Benutzeroberfläche Bezug genommen.

Die dargestellte Walzenlage folgt der historischen Reihenfolge von rechts (schnell) nach links (langsam) und wird in der UI (User Interface) entsprechend durch die `ReverseMultiSelect.vue`-Komponente abgebildet; links davon ist, wie beim Original, die Umkehrwalze leicht abgesetzt dargestellt.

10.1 Enigma-Seite

Die Enigma-Seite bildet den ersten zentralen Bereich der Webanwendung und dient gleichzeitig als Startansicht. Sie wird sowohl beim Aufruf der Root-URL (`/`) als auch unter `/enigma` geladen. Damit stellt sie die Standardseite dar, auf die Benutzer automatisch gelangen, sofern keine spezifischere Route aufgerufen wird.

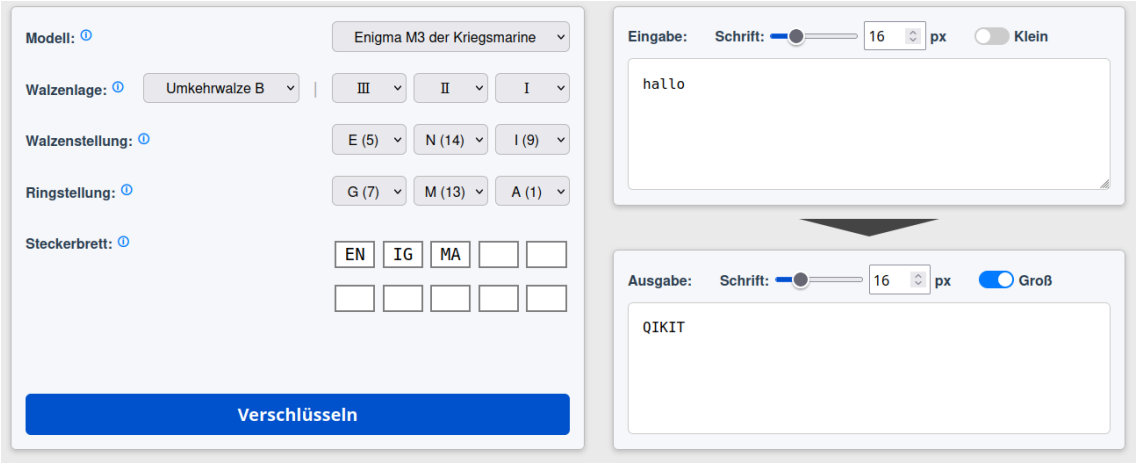


Abbildung 8: Screenshot der EnigmaView mit Beispielverschlüsselung in der eigenen Webanwendung

Die Seite ermöglicht die Konfiguration einer historischen Enigma-Chiffriermaschine sowie die anschließende Verschlüsselung eines eingegebenen Textes über einen API-Aufruf.

10.1.1 Benutzeroberfläche

Die Benutzeroberfläche ist in zwei Hauptbereiche unterteilt: den linken Konfigurationsbereich und den rechten Textbereich. Im linken Bereich kann der Nutzer alle für die Enigma relevanten Parameter einstellen:

- **Modell**: Auswahl des verwendeten Enigma-Modells (z. B. Wehrmacht M3 oder M4).
- **Umkehrwalze**: Festlegung der Umkehrwalze.
- **Walzenlage**: Auswahl und Anordnung der drei bzw. vier Walzen.
- **Walzenstellung**: Anfangspositionen der Walzen.
- **Ringstellung**: Mechanische Verschiebung des Ringkerns jeder Walze.
- **Steckerbrett**: Buchstabenpaare zur Substitution, einzugeben als zwei Zeichen pro Feld.

Im rechten Bereich befinden sich zwei untereinander angeordnete Textfelder. Oben wird der zu verschlüsselnde Text eingegeben, unten erscheint nach der Verschlüsselung die Ausgabe. Beide Felder sind durch das Layout optisch klar voneinander getrennt.

Dynamische Anpassung an das Enigma-Modell: Die Benutzeroberfläche passt sich, je nach ausgewähltem Enigma-Modell, automatisch an. Für die Modelle „I“ und „M3“ stehen jeweils drei Walzen zur Verfügung, wobei sich die Anzahl

der auswählbaren Walzen zwischen den Modellen unterscheidet: Modell „I“ bietet fünf, Modell „M3“ hingegen acht Walzen zur Auswahl. Intern wird in beiden Fällen jedoch dieselbe Modellnummer (`model = 3`) an die API übermittelt, da diese nur die Anzahl der eingesetzten Walzen berücksichtigt.

Beim Modell „M4“ wird zusätzlich eine vierte Walze aktiviert, was die Benutzeroberfläche um eine weitere Auswahlmöglichkeit für Walzenlage und Walzenstellung erweitert. Außerdem sind die Auswahlmöglichkeiten für die Umkehrwalze (UKW) und die vierte Walze in diesem Modus individuell konfigurierbar. In diesem Fall wird der Wert `model = 4` an die API gesendet, um die erweiterte Konfiguration korrekt abzubilden.

Durch diese reaktive Anpassung stellt die Benutzeroberfläche sicher, dass nur gültige Konfigurationen ausgewählt werden können, was die Bedienung vereinfacht und Fehlerquellen reduziert.

Reaktives Texteingabefeld: Das Eingabefeld für den Klartext reagiert unmittelbar auf jeden Tastendruck. Über einen sogenannten Debounce-Mechanismus wird die Verarbeitung der Eingabe leicht verzögert, um eine zu hohe Anzahl von API-Anfragen zu vermeiden. So wird die Verschlüsselung dynamisch und benutzerfreundlich aktualisiert, ohne dass ein zusätzlicher Bestätigungsbutton betätigt werden muss.

10.1.2 API-Kommunikation (/enigma)

Die Verschlüsselung wird entweder durch Betätigung des „Verschlüsseln“-Buttons oder durch das reaktive Texteingabefeld ausgelöst. Dabei werden die aktuell gesetzten Konfigurationsparameter zusammen mit dem Eingabetext über ein POST-Request an die `/enigma`-API gesendet.

Der Server verarbeitet die Eingabedaten mithilfe des Simulators der Enigma-Maschine und gibt den verschlüsselten Text als Antwort zurück. Diese Antwort wird anschließend direkt im Ausgabefeld der Oberfläche angezeigt.

10.1.3 Ablauf der Verschlüsselung

Der gesamte Ablauf erfolgt interaktiv im Browser. Die Vue-Komponente `EnigmaRequestView.vue` verwaltet dabei den Zustand der Benutzereingaben über das `settings.enigma`-Objekt. Die Texteingabe wird zusätzlich validiert, um sicherzustellen, dass nur vom Backend unterstützte Zeichen übermittelt werden.

10.2 Zyklometer-Seite

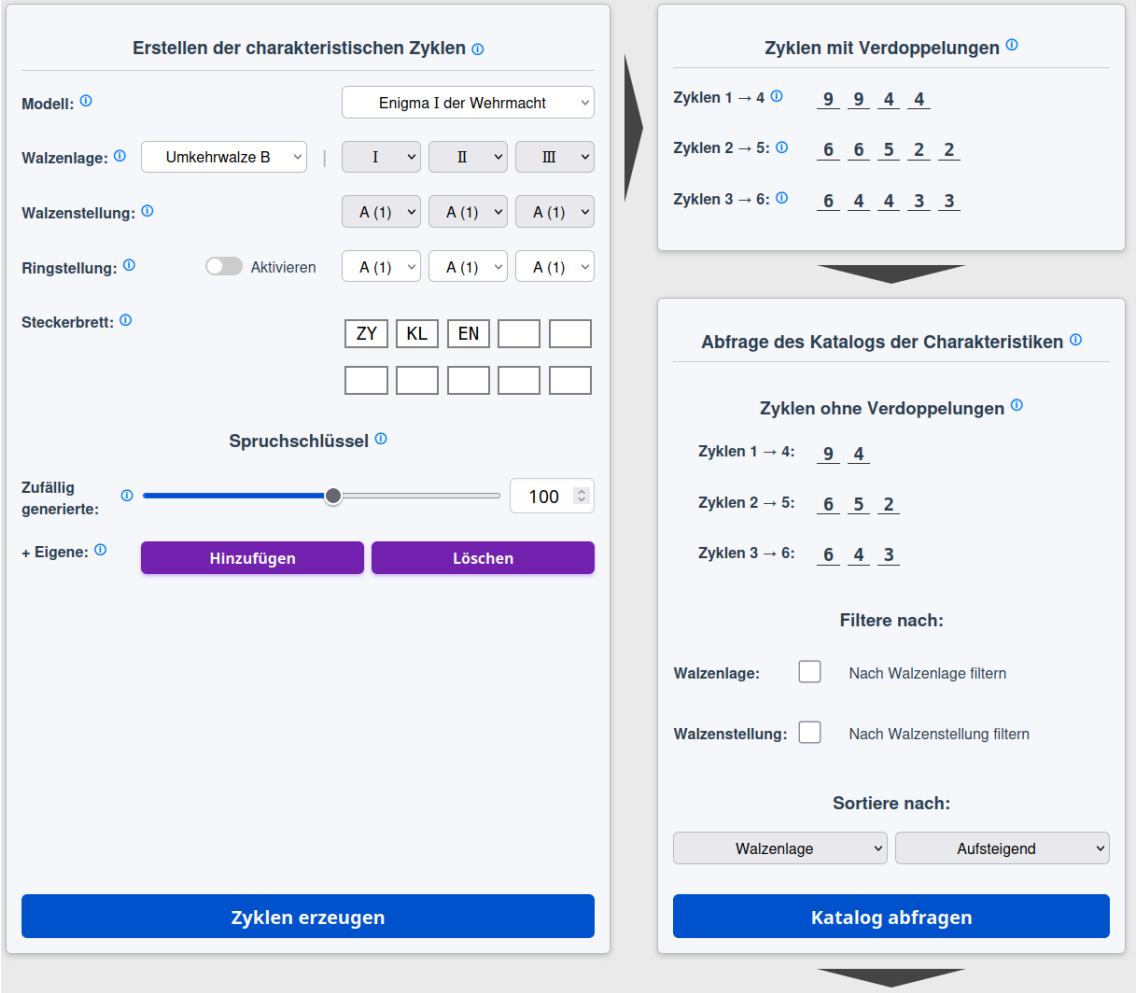


Abbildung 9: Screenshot der CyclometerView mit Darstellung der charakteristischen Zyklen

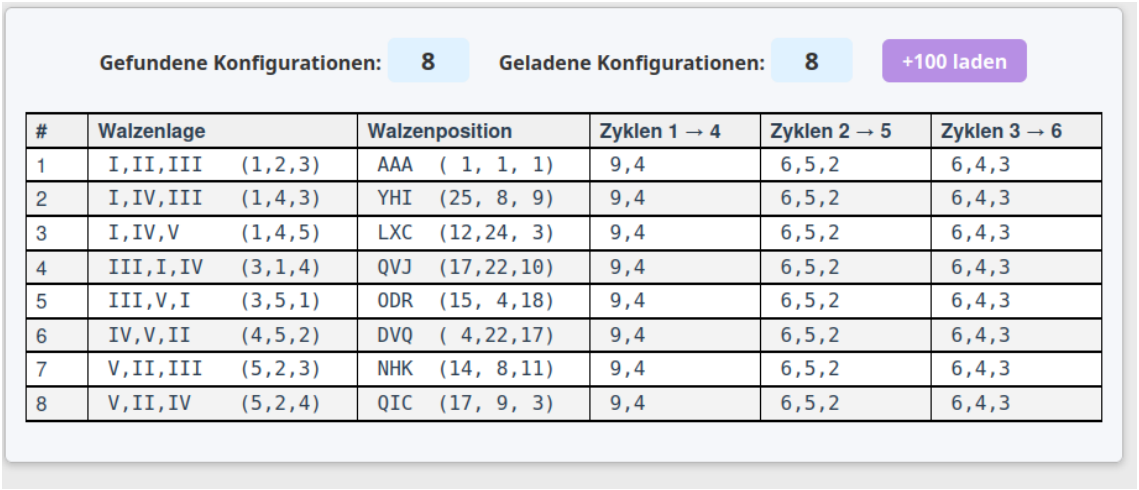
Die Zyklometer-Seite ist in drei Hauptbereiche gegliedert und dient der Erzeugung sowie Analyse charakteristischer Zyklen der Enigma-Maschine. Sie ist unter der Route `/cyclometer` erreichbar.

10.2.1 Benutzeroberfläche

Die Seite gliedert sich in folgende Bereiche:

- **Linker Bereich (Eingabeformulare):** Hier kann der Benutzer die Enigma-Konfiguration einstellen, inklusive Walzenlage, Walzenstellung, Ringstellung (optional aktivierbar), Steckerbrett sowie Anzahl der zufälligen und manuellen Spruchschlüssel.

- **Rechter Bereich (Ausgabe der Zyklen und Katalogabfrage):** Im oberen Teil des rechten Bereichs werden die erzeugten charakteristischen Zyklen für die Permutationen 1:4, 2:5 und 3:6 angezeigt. Darunter befindet sich ein Formular zur Abfrage des Katalogs mit Filter- und Sortiermöglichkeiten basierend auf Walzenlage, Walzenstellung und Zyklendaten.
- **Unterer Bereich (Tabellarische Darstellung des Katalogs):** Sobald eine Katalogantwort vom Backend vorliegt, werden die Ergebnisse in einer Tabelle am unteren Seitenbereich dargestellt. Diese Tabelle zeigt detaillierte Konfigurationen mit Walzenlage, Walzenstellung sowie die jeweiligen Zyklendaten. Zusätzlich sind Steuerungen zum Nachladen weiterer Einträge vorhanden, um die Übersichtlichkeit zu wahren.



#	Walzenlage	Walzenposition	Zyklen 1 → 4	Zyklen 2 → 5	Zyklen 3 → 6
1	I,II,III (1,2,3)	AAA (1, 1, 1)	9,4	6,5,2	6,4,3
2	I,IV,III (1,4,3)	YHI (25, 8, 9)	9,4	6,5,2	6,4,3
3	I,IV,V (1,4,5)	LXC (12,24, 3)	9,4	6,5,2	6,4,3
4	III,I,IV (3,1,4)	QVJ (17,22,10)	9,4	6,5,2	6,4,3
5	III,V,I (3,5,1)	ODR (15, 4,18)	9,4	6,5,2	6,4,3
6	IV,V,II (4,5,2)	DVQ (4,22,17)	9,4	6,5,2	6,4,3
7	V,II,III (5,2,3)	NHK (14, 8,11)	9,4	6,5,2	6,4,3
8	V,II,IV (5,2,4)	QIC (17, 9, 3)	9,4	6,5,2	6,4,3

Abbildung 10: Screenshot der CyclometerView mit der Tabelle möglicher Enigma-Konfigurationen

Die tabellarische Darstellung der möglichen Enigma-Konfigurationen ist in Abbildung 10 zu sehen und befindet sich im unteren Bereich der Zyklometer-Seite.

10.2.2 API-Kommunikation

Die Kommunikation erfolgt über zwei API-Endpunkte:

- **/cyclometer**: Beim Absenden des Formulars im linken Bereich werden die Enigma-Einstellungen an diesen Endpunkt gesendet. Die Antwort enthält die berechneten charakteristischen Zyklen, die im rechten oberen Bereich der Benutzeroberfläche angezeigt werden – inklusive der Zyklen mit Verdoppelungen.
- **/catalogue**: Die Katalogabfrage nutzt die zuvor berechneten Zyklen aus `/cyclometer` als Grundlage. Dabei werden Verdoppelungen in den Zyklenlängen herausgerechnet, indem die Anzahl ihrer Vorkommen halbiert und

aufgerundet wird. Beispielsweise bleibt aus zwei Zyklen der Länge 3 ein Zyklus der Länge drei übrig. Oder bei fünf Zyklen der Länge 2 bleiben drei erhalten. Die so transformierten Zyklenlängen dienen anschließend als Filterkriterien zur Auswahl passender Enigma-Konfigurationen aus dem Katalog der Charakteristiken.

10.2.3 Besondere Funktionen

- Die Ringstellung kann über eine Checkbox aktiviert oder deaktiviert werden und beeinflusst die Zykluserzeugung.
- Manuell hinzugefügte Spruchschlüssel ermöglichen individuelle Anpassungen.
- Sortier- und Filteroptionen für Walzenlage und Walzenstellung etc. erlauben gezielte Suchen im Katalog.
- Die Ergebnisliste kann in Seiten von jeweils 100 Einträgen nachgeladen werden, um Performance und Übersichtlichkeit zu optimieren.

10.3 Benutzerführung und Darstellung

Die Webanwendung ist so gestaltet, dass Nutzer ohne Vorkenntnisse die Verschlüsselung mit der Enigma intuitiv bedienen können. Die Bedienung des Zyklometers hingegen ist deutlich anspruchsvoller und richtet sich an Anwender mit vertieftem Verständnis der Materie. Um die Nutzung zu erleichtern, sind zahlreiche Tooltips integriert, die wichtige Hinweise und Erklärungen liefern.

Darüber hinaus steht eine separate Seite mit einer ausführlichen Bedienungsanleitung zur Verfügung.

Die klare Aufteilung in Eingabe-, Ausgabebereiche und Ergebnislisten unterstützt eine logische Abfolge der Arbeitsschritte. Hinweise und Validierungen erleichtern die korrekte Eingabe kryptografischer Parameter und verhindern Fehler frühzeitig.

Die interaktive Darstellung der Zyklen und der Enigma-Konfigurationen erfolgt in übersichtlichen Tabellen und visualisierten Komponenten, die auf Benutzeraktionen reagieren. Zudem werden Lade- und Bearbeitungszustände visuell kommuniziert, um Transparenz im Ablauf zu schaffen.

11 Inbetriebnahme und Nutzung der Quellcodebasis

Der vollständige Quellcode dieser Arbeit ist in vier öffentlich zugänglichen Repositories auf `GitHub` verfügbar. Jede Komponente wird mit einer eigenen `README.md`-Datei dokumentiert, die alle relevanten Installations- und Konfigurationshinweise enthält. Neben den bereits beschriebenen Projekten steht auch das Repository der Bachelorarbeit selbst zur Verfügung, welches die gesamte Dokumentation und die Codebasis bündelt [13].

In diesem Kapitel wird die Vorgehensweise zur Installation, Konfiguration und Ausführung der Softwarekomponenten beschrieben.

11.1 Quellcodeübersicht

- **Native Enigma-Simulation**
  `https://github.com/UPEV1sion/Enigma/tree/server`

  Beinhaltet den Quellcode der Enigma-Simulation in `C` sowie das Buildsystem zur Erzeugung der `libenigma.so`.

- **Backend REST API**
  `https://github.com/UPEV1sion/Enigma-API`

  Enthält die Spring Boot Anwendung, welche über die
  `FFM API API` mit dem nativen C-Backend kommuniziert.

- **Webfrontend (Vue 3)**
  `https://github.com/Bibble-code/EnigmaSite`

  Implementiert das interaktive Interface zur Enigma und zum Zyklometer. Die Anwendung kann lokal über `npm` gestartet oder als statische Website gebaut werden.

Installation und Start

Für die vollständige Inbetriebnahme sind drei separat zu konfigurierende Komponenten erforderlich: die native Bibliothek, das Java-Backend und das Webfrontend. Die folgende Übersicht beschreibt die zentralen Schritte.

1. Native Bibliothek (`libenigma.so`)

Die Enigma-Simulation ist in `C` implementiert und muss manuell aus dem separaten Repository `Enigma/tree/server` kompiliert werden. Dabei ist wie folgt vorzugehen:

- Repository klonen und gemäß der dortigen `README.md` einrichten.
- `cmake` und `make` ausführen, um die Datei `libenigma.so` zu erzeugen.
- Die erzeugte `.so`-Datei nach `src/main/enigma_c/` im Backend-Projekt kopieren.

Hinweis: Die Datei wird absichtlich nicht versioniert und muss lokal erzeugt werden. Dabei ist darauf zu achten, dass die Kompilierung **für das Zielsystem** erfolgt (z. B. passende Architektur und `glibc`-Version), da unter Linux systemnahe Inkompatibilitäten auftreten können — selbst zwischen gängigen Distributionen wie Ubuntu und Arch Linux.

2. Backend (Spring Boot)

Das Backend ist eine Java-Anwendung, die über die FFM API auf die native Bibliothek zugreift. Zur Vorbereitung:

- `Java 22+` und `Maven 3.9+` installieren.
- Datenbank initialisieren:
  - PostgreSQL 17+ installieren.
  - Datenbank und Benutzer gemäß `application.properties` anlegen.
  - SQL-Dump entpacken: `unzip db/init.sql.zip`
  - Dump importieren: `psql -U <user> -d <datenbankname> < init.sql`
- Weitere Details enthält `db/README.md`.
- Backend starten:

```
mvn clean install java --enable-native-access=ALL-UNNAMED -jar target/Enigma_API.jar
```

3. Frontend (Vue 3)

Die grafische Benutzeroberfläche basiert auf Vue 3 und kommuniziert über HTTP mit dem Backend.

- `Node.js` und `npm` installieren.
- Abhängigkeiten mit `npm install` installieren.
- Start im Entwicklungsmodus: `npm run serve`
- Optional: `npm run build` für Produktionsbuild

11.2 Veröffentlichte Anwendung

Die im Rahmen dieser Arbeit entwickelte Webanwendung ist öffentlich zugänglich unter: `https://enigma-zyklometer.rwu.de`

11.3 Detaillierte Anleitungen in den README-Dateien

Die vollständigen Anleitungen zur Installation, Konfiguration und Nutzung der einzelnen Komponenten befinden sich jeweils in den `README.md`-Dateien der zugehörigen Repositories. Sie enthalten unter anderem Hinweise zur Datenbankeinrichtung, zu Lizenzinformationen sowie zu Build- und Startprozessen.

12 Evaluation und Reflexion

12.1 Selbstständige Konzeption und Zielerreichung

Die Aufgabe, eine bestehende C-Implementierung der Enigma-Maschine als Webanwendung öffentlich zugänglich zu machen, bot großen gestalterischen Spielraum und stellte dadurch hohe Anforderungen an die eigenständige Konzeption und Umsetzung. Rückblickend wurde das übergeordnete Ziel erreicht: Die Anwendung ist online verfügbar, verbindet moderne Webtechnologien mit systemnaher Programmierung und erfüllt sowohl funktionale als auch technische Anforderungen.

12.2 Herausforderungen in der technischen Umsetzung

Die Anbindung der bestehenden C-Implementierung der Enigma-Maschine an die Java-basierte Webarchitektur stellte erhebliche technische Herausforderungen dar. Zwar gestaltete sich die Umsetzung der FFM API zunächst als schwierig und zeitaufwändig, dennoch erwies sich der Einsatz von FFM letztlich als die richtige Entscheidung. Die API ermöglichte eine stabile Schnittstelle, die den Datenaustausch zwischen C und Java zuverlässig realisierte.

Auch die Anbindung der PostgreSQL-Datenbank erwies sich als anspruchsvoll. Die Entwicklung performanter und komplexer Abfragen im Java-Backend erforderte umfangreiche Optimierungen und ein tiefgehendes Verständnis der Datenbank- und Framework-Mechanismen. Besonders die Performanceoptimierung des Katalogsystems erwies sich als deutlich aufwändiger als zunächst erwartet und erforderte wiederholte Anpassungen, um eine flüssige Benutzererfahrung sicherzustellen.

12.3 Komplexität und Schnittstellenproblematik

Das Projekt erforderte die Koordination eines umfangreichen Technologie-Stacks, von C über Java mit Spring Boot und PostgreSQL bis zum Vue3-Frontend. Änderungen an einer Komponente hatten häufig Auswirkungen auf andere Bereiche, was die Komplexität erhöhte und die Fehleranfälligkeit steigerte.

13 Fazit und Ausblick

Die Arbeit zeigt die erfolgreiche Umsetzung einer komplexen technischen Aufgabe: Die Überführung einer systemnahen C-Implementierung der Enigma-Maschine und insbesondere des Zyklometers in eine moderne, öffentlich zugängliche Webanwendung.

Dabei ist es gelungen, erstmals eine funktionierende Simulation des Zyklometers online zur Verfügung zu stellen. So kann die Funktionsweise des Zyklometers besser verstanden und die Analyse der Zyklen praktisch nachvollzogen werden.

Für die Zukunft bieten sich ein paar größere Erweiterungen an, die das Potenzial der Anwendung deutlich steigern könnten:

- **Aufteilung der Zyklometer-Abfrage in zwei Stufen:** Die derzeitige Zyklometer-Abfrage kombiniert derzeit zwei Funktionalitäten: Die Verschlüsselung der Spruchschlüssel und die Generierung der Zyklen aus diesen. Eine Aufteilung könnte wie folgt aussehen:

  Die erste Stufe erzeugt aus unverschlüsselten Spruchschlüsseln verschlüsselte Spruchschlüssel basierend auf einer Enigma-Konfiguration. In der zweiten Stufe werden anschließend aus diesen verschlüsselten Schlüsseln die charakteristischen Zyklen gebildet. Diese Zweistufigkeit würde die Simulation der sogenannten „deutschen“ und „polnischen“ Seite des Zyklometers realistischer abbilden und die Analyse präzisieren.

- **Erweiterung um die Entzifferung der Spruchschlüssel:** Derzeit können mit dem Zyklometer nur Walzenlage und Walzenstellung bestimmt werden, nicht jedoch die Steckerbrett-Verbindungen. Eine zukünftige Erweiterung könnte eine neue Abfrage implementieren, die eine Entzifferung der Steckerbrett-Konfiguration erlaubt.

- **Integration eines mit dem Spruchschlüssel verschlüsselten Texts:** Zusätzlich zum verschlüsselten Spruchschlüssel könnte ein kurzer Nachrichtentext übermittelt werden, der mit dem jeweiligen Spruchschlüssel verschlüsselt wird. Nach Rekonstruktion sämtlicher Enigma-Parameter (einschließlich Steckerbrett) könnte zunächst der Spruchschlüssel und anschließend mit diesem der Text entschlüsselt werden. Dies würde eine realitätsnähere Simulation des vollständigen kryptographischen Ablaufs ermöglichen.

- **Skalierbarkeit und Deployment:** Bisher wurde auf eine containerisierte Bereitstellung (z. B. mit Docker) verzichtet. Eine solche Maßnahme könnte die Skalierbarkeit und Performance der Webanwendung verbessern und den Betrieb vereinfachen.

Diese Weiterentwicklungen würden nicht nur die Funktionalität vertiefen, sondern auch den praktischen Nutzen der Anwendung steigern.

Glossar

Frontend-Technologien

Axios Eine JavaScript-Bibliothek zum einfachen Senden von HTTP-Anfragen, besonders für die Kommunikation mit REST-APIs.

Composition API Ein modernes Programmiermodell in Vue 3 zur besseren Strukturierung von Komponentenlogik, besonders nützlich für komplexe Anwendungen.

CORS (Cross-Origin Resource Sharing) Ein Sicherheitsmechanismus, der steuert, welche Ressourcen von einer Webdomain aus auf eine andere zugreifen dürfen.

DOM Das Document Object Model ist eine standardisierte, baumartige Darstellung von HTML- und XML-Dokumenten, die es Skriptsprachen wie JavaScript ermöglicht, Inhalte, Struktur und Stil einer Webseite dynamisch zu verändern.

Framework Eine wiederverwendbare Softwarestruktur, die als Gerüst für die Entwicklung von Anwendungen dient. Frameworks stellen vorgefertigte Funktionen und Konventionen bereit, um typische Programmieraufgaben zu vereinfachen und zu standardisieren. Beispiele sind *Spring Boot* für Java oder *Vue.js* für das Web-Frontend.

History API Browser-API zur Manipulation des Verlaufs und der URL, wichtig für Navigation in SPAs ohne Seitenreload.

Preflight-Anfrage Eine automatische OPTIONS-Anfrage des Browsers zur Überprüfung von CORS-Richtlinien vor der eigentlichen HTTP-Anfrage.

Single-File Component (SFC) Dateien, die Template, Logik und Styles in einer einzigen .vue-Datei bündeln und so modularen und übersichtlichen Code ermöglichen.

Single-Page Application (SPA) Webanwendungen, die in einer einzelnen HTML-Seite laufen und Inhalte dynamisch nachladen, um schnelle und flüssige Nutzererlebnisse zu ermöglichen.

Vue 3 Die aktuelle Hauptversion von Vue.js, die neue Features wie die Composition API für bessere Code-Organisation und höhere Leistung bietet.

Vue CLI Ein Kommandozeilen-Werkzeug zum schnellen Erstellen und Verwalten von Vue-Projekten mit integrierten Build-Tools und Plugins.

Vue.js Ein progressives JavaScript-Framework zur Erstellung interaktiver Benutzeroberflächen. Es ähnelt AngularJS und React, ist aber besonders leichtgewichtig und flexibel.

Backend-Technologien und Java

API (Application Programming Interface) Eine Schnittstelle, die es ermöglicht, dass verschiedene Softwarekomponenten miteinander kommunizieren. APIs definieren, wie Funktionen, Daten und Dienste von einem Programm für andere Programme zugänglich gemacht werden. Beispiele in diesem Projekt sind die *FFM API* zur Anbindung nativer Funktionen und die *Enigma-API* als REST-Schnittstelle zur Enigma-Verschlüsselung.

Backend Der Teil einer Anwendung, der serverseitig Daten verarbeitet, Logik ausführt und Schnittstellen zu Datenbanken bereitstellt.

DTO (Data Transfer Object) Einfache Datenstruktur zur Übertragung von Daten zwischen verschiedenen Schichten einer Anwendung, z. B. zwischen Controller und Service.

Hibernate Java-Framework für Objekt-Relationale Abbildung (ORM), das SQL-Komplexität reduziert.

Java Foreign Function und Memory API (FFM API) Eine moderne Java-API zur effizienten und sicheren Anbindung nativer Bibliotheken (z. B. in C). Sie ermöglicht direkten Zugriff auf native Funktionen und Speicher, als Nachfolger des älteren JNI-Mechanismus, mit einfacherem und performanterem Code.

Java Native Interface (JNI) Ältere Schnittstelle zum Aufruf nativer Bibliotheken aus Java, komplexer als FFM API.

JDBC (Java Database Connectivity) Standard-API in Java zur Kommunikation mit relationalen Datenbanken.

Native Funktion Funktion, die außerhalb der Java Virtual Machine (JVM) in einer anderen Programmiersprache wie C oder C++ implementiert ist und über Schnittstellen wie JNI oder die Foreign Function & Memory API (FFM API) von Java aufgerufen wird.

ResponseEntity Klasse zur flexiblen Modellierung von HTTP-Antworten, inklusive Statuscode und Headern.

REST (Representational State Transfer) Ein Architekturstil für Webservices, der auf HTTP-Methoden (GET, POST, PUT, DELETE) basiert. REST definiert Prinzipien wie zustandslose Kommunikation, klare Ressourcennamen (URLs) und standardisierte Operationen. APIs, die diese Prinzipien befolgen, werden als *RESTful* bezeichnet und ermöglichen einfache, skalierbare und leicht verständliche Schnittstellen.

RESTful HTTP-API Eine Web-API, welche die Prinzipien von REST nutzt, um Ressourcen über HTTP-Methoden zugänglich zu machen.

Service-Schicht Softwarearchitektur-Schicht, die Geschäftslogik kapselt und von Controllern genutzt wird.

Spring Boot Ein Framework zur Entwicklung von Java-Anwendungen, das auf dem Spring-Framework aufbaut. Es ermöglicht durch Konventionen und automatische Konfiguration eine schnelle Einrichtung von produktionsreifen Webanwendungen. Spring Boot unterstützt die Entwicklung von REST-APIs, Validierung (z. B. mit `javax.validation`) sowie Dependency Injection und bietet umfangreiche Middleware-Integration.

Datenbanken und Indizes

Containment Operator (@>) Operator zur Abfrage, ob ein Array oder JSON ein bestimmtes Element enthält.

Fremdschlüssel Ein Attribut, das auf einen Primärschlüssel einer anderen Tabelle verweist und somit eine Beziehung zwischen Tabellen definiert. Fremdschlüssel sorgen für referenzielle Integrität und verhindern inkonsistente Daten.

GIN-Index (Generalized Inverted Index) Ein spezieller Index in PostgreSQL, der schnelle Volltextsuche und Abfragen auf komplexen Datentypen wie Arrays und JSONB ermöglicht. Er optimiert Suchanfragen durch Invertierung der Datenstruktur.

Normalisierung Ein Verfahren zur Strukturierung von Datenbanken, das Redundanzen vermeidet und die Datenintegrität durch die Einhaltung von Normalformen (z. B. dritte Normalform, 3NF) sicherstellt. Normalisierung verbessert die Wartbarkeit und Konsistenz der Daten.

Pagination Die Unterteilung großer Datenmengen in überschaubare Seiten (Pages) zur Verbesserung der Performance und Nutzerfreundlichkeit. Pagination reduziert die Netzwerklast und die Ladezeiten beim Abrufen von Datensätzen aus der Datenbank.

PostgreSQL Leistungsstarkes, objektrelationales Open-Source-Datenbanksystem mit umfangreichen Managementfunktionen.

Primärschlüssel Ein eindeutiges Attribut oder eine Attributkombination, die jeden Datensatz in einer Tabelle eindeutig identifiziert. Primärschlüssel sind fundamental für Datenintegrität und die Herstellung von Beziehungen in relationalen Datenbanken. Besteht der Primärschlüssel aus mehreren Attributen, so spricht man von einem zusammengesetzten Primärschlüssel.

Query-Planner Komponente einer Datenbank, die entscheidet, wie eine Abfrage ausgeführt wird, um sie möglichst effizient zu machen.

Server und Infrastruktur

Caching Zwischenspeicherung häufig genutzter Daten, um Ladezeiten und Serverlast zu verringern.

HTTP-Kompression Verfahren zur Reduktion der Größe von HTTP-Antworten zur Beschleunigung der Datenübertragung.

nginx Beliebter, performanter Webserver und Reverse-Proxy.

Reverse-Proxy Server, der Anfragen an Backend-Server weiterleitet, oft für Lastverteilung und Sicherheit genutzt.

systemd-Dienst Linux-Manager für Hintergrundprozesse (Dienste).

Speichertechnologienn

FunctionDescriptor Beschreibung der Signatur nativer Funktionen für die FFM API.

MemorySegment Speicherbereich in der Java FFM API für sicheren Zugriff auf nativen Speicher.

MethodHandle Java-Objekt zur dynamischen Methodenreferenzierung.

Speicher-Alignment Ausrichtung von Daten im Speicher nach bestimmten Grenzen für Performance und Kompatibilität.

Speicher-Layout (MemoryLayout) Anordnung von Daten im Speicher für Kompatibilität mit nativen Bibliotheken.

Literatur

[1] D. Palloks. „Enigma Simulator (Version 2.62)“. Zugriff am 30. Juni 2025. Adresse: https://people.physik.hu-berlin.de/~palloks/js/enigma/enigma-u_v262.html.

[2] Welt-Redaktion. „Enigma – der Kampf um den Code der Wehrmacht“. Welt.de, Artikel vom 15. Oktober 2014, besucht am 12. Juni 2025. Adresse: https://www.welt.de/geschichte/zweiter-weltkrieg/gallery136436121/Enigma-der-Kampf-um-den-Code-der-Wehrmacht.html.

[3] Wikipedia contributors. „Enigma-Walzen — Wikipedia, Die freie Enzyklopädie“, besucht am 12. Juni 2025. Adresse: https://de.wikipedia.org/wiki/Enigma-Walzen.

[4] E. Piotte. „Enigma Cipher Simulator“, besucht am 12. Juni 2025. Adresse: https://piotte13.github.io/enigma-cipher/.

[5] Wikipedia contributors. „Zyklometer (Kryptologie) — Wikipedia, Die freie Enzyklopädie“. Zugriff am 12. Juni 2025. Adresse: https://de.wikipedia.org/wiki/Zyklometer_(Kryptologie).

[6] M. Rejewski, „How Polish Mathematicians Deciphered the Enigma“, *Annals of the History of Computing*, Jg. 3, Nr. 3, S. 213–234, 1981. DOI: 10.1109/MAHC.1981.10016.

[7] A. Hasanic, *Kryptanalyse der Enigma durch Nachimplementierung des Zyklometers*, Informatikprojekt, RWU–University of Applied Sciences, Betreuer: Prof. Dipl.-Math. Ekkehard Löhmann, Projekt auf GitHub, 2024. Adresse: https://github.com/murderbaer/enigma.

[8] E. Schäffer, *Kryptoanalyse der Enigma-Maschine durch eine Software-Nachbildung der Turing-Welchman-Bombe*, Informatikprojekt, RWU–University of Applied Sciences, Betreuer: Prof. Dipl.-Math. Ekkehard Löhmann, 2025. Adresse: https://github.com/UPEV1sion/Turing-Bombe-Notes.

[9] E. Schäffer, *Enigma – Weiterentwicklung des Forks von Schäffers Enigma-Implementierung*, GitHub Repository unter dem Alias UPEV1sion, Fork und Weiterentwicklung des Projekts, 2025. Adresse: https://github.com/UPEV1sion/Enigma/tree/server.

[10] T. Steidle, *Enigma-API*, Repository gehostet von Emanuel Schäffer. Zugriff am 22. Juni 2025, 2025. Adresse: https://github.com/UPEV1sion/Enigma-API.

[11] T. Steidle, *EnigmaSite – Enigma- und Zyklometer-Simulation (Quellcode)*, Zugriff am 22. Juni 2025, 2025. Adresse: https://github.com/Bibblecode/EnigmaSite.

[12] T. Steidle, *Enigma- und Zyklometer-Simulation*, Zugriff am 22. Juni 2025, 2025. Adresse: https://enigma-zyklometer.rwu.de.

[13] T. Steidle, *Enigma-Zyklometer-Notes*, GitHub Repository, Zugriff am 1. Juli 2025, 2025. Adresse: https://github.com/Bibble-code/Enigma-Zyklometer-Notes.