Westfälische Hochschule
Gelsenkirchen Bocholt Recklinghausen
University of Applied Sciences

IHK Nord Westfalen

Friends and Places Server - Ausarbeitung

Simon, Nick, Johannes und Jan

Inhaltsverzeichnis

Einleitung

Dieses Dokument ist die Ausarbeitung für das Softwareprojekt des FAP-Servers, welches im Rahmen der Veranstaltung “Fortgeschrittene Internet Technologien” durchgeführt wurde. Zur Auswahl standen zwei Aufgaben: Implementation eines Frontends für den bereits bestehenden Tomcat-Server und Reimplementation des Backends in einer Programmiersprache bzw. mit einem Framework unserer Wahl. Wir haben uns für die Reimplementation entschieden. In den folgenden Kapiteln werden daher die angewandten Technologien für die Implementation des Backends sowie unsere gewählte Architektur und Besonderheiten, Schwierigkeiten und wesentliche Teile des Projekts dargestellt. Nach der Beschreibung des technischen Teils reflektieren wir noch einmal auf das Projekt und gehen zum Abschluss auf mögliche Geschäftsmodelle für den FAP-Server ein.

Architektur

REST

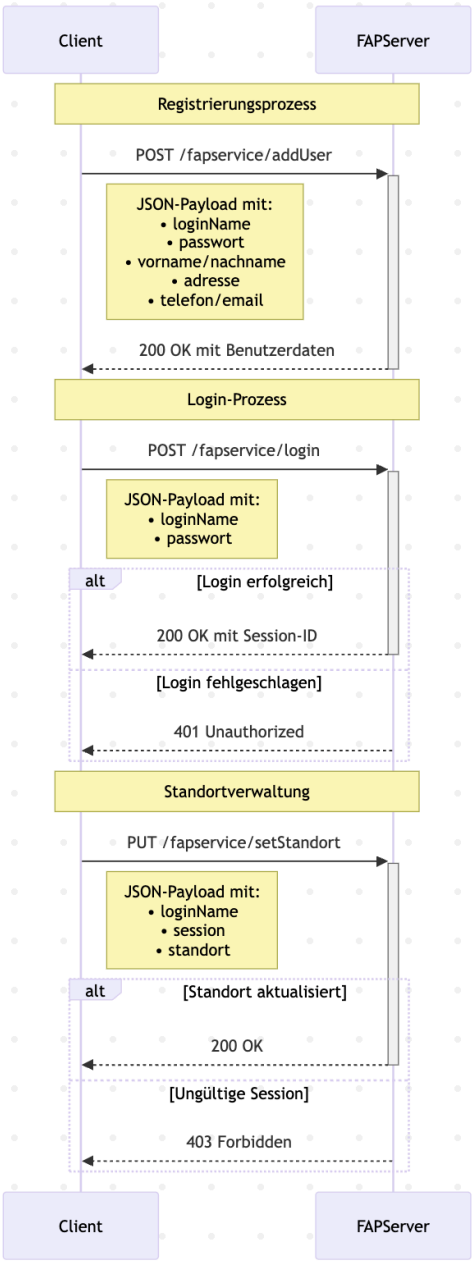
REST (Representational State Transfer) ist ein Architekturstil für verteilte Systeme. Es orientiert sich an den Prinzipien des World Wide Web und nutzt standardisierte HTTP-Methoden wie GET, POST, PUT und DELETE zur Interaktion mit klar definierten Ressourcen, die über URIs identifiziert werden. GET ist zum Abruf von Informationen. POST ist aufzurufen, um Ressourcen zu erstellen. PUT ist dann zu nutzen, wenn diese aktualisiert werden sollten. Zuletzt ist DELETE zur Löschung von Ressourcen zu nutzen.

Im Vergleich zu älteren Protokollen wie SOAP, das auf XML-Nachrichten und einheitliche Schnittstellen setzt, ist REST leichter gewichtet, flexibler und besser in bestehende Web-Infrastrukturen integrierbar. Es verzichtet auf komplexe Standards wie WSDL und setzt stattdessen auf einfache, sprechende URLs sowie eine konsistente Nutzung von HTTP.

Ein zentraler Vorteil von REST ist die Nutzung bestehender Web-Technologien: HTTP-Clients, Caching, Authentifizierungsmechanismen (wie OAuth oder Basic

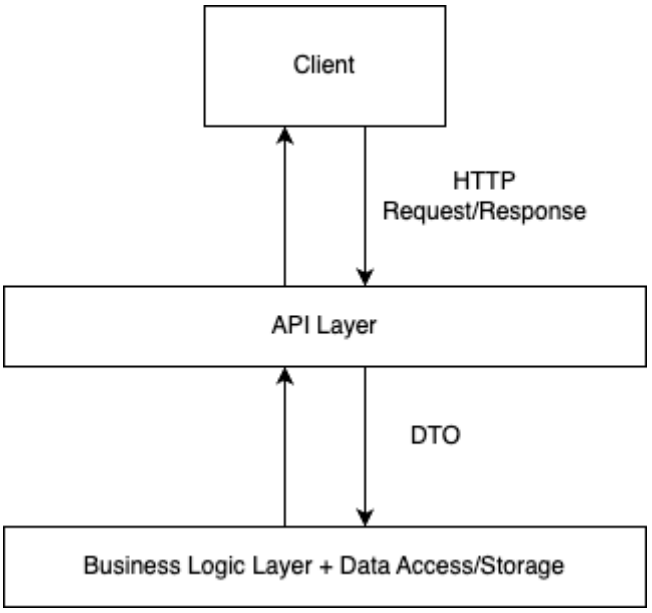
Auth) und standardisierte Datenformate wie JSON oder XML stehen bereits zur Verfügung. Dies erleichtert die Entwicklung, Skalierung und Wartung moderner Webanwendungen.

Die entwickelte Anwendung in GO basiert auf der REST-Architektur. Das folgende Sequenzdiagramm zeigt die Nutzung des API-Servers anhand eines Beispielhaften Logins und Zugriff auf den *setStandort* Endpunkt:



Layered Architecture

Der Server ist generell sehr simpel aufgebaut, daher können wir das Layered Architecture Modell nicht voll ausnutzen, man kann aber schon gut zwei getrennte Layers identifizieren. Die erste Layer bilden die Handler für die REST-Endpunkte im *handlers* package. Sie nehmen Requests entgegen, validieren deren Korrektheit und geben die relevanten Daten weiter an die anderen Teile des Programms. Diese Layer ist vollständig und alleinig zuständig für den Umgang mit HTTP-bezogener Logik. Die zweite Layer besteht dann aus dem User-Service und allen anderen relevanten Funktionen zur Umsetzung der Geschäftslogik. In unserem Fall beinhaltet die Layer für Geschäftslogik ebenfalls Data Access und Data Storage, da dieser über eine In-Memory-Lösung implementiert wurde. Die Interaktion der Schichten sieht wie folgt aus:



Auf die Implementierung der Datenspeicherung sowie die DTOs wird in späteren Kapiteln noch genauer eingegangen.

Golang als Programmiersprache

Bei der Auswahl der Programmiersprache ist unsere Wahl relativ schnell in Richtung Golang gegangen. Golang ist eine statisch typisierte und kompilierte Programmiersprache, die ab 2009 von Google entwickelt wurde. Sie wurde mit dem

Ziel entworfen, eine einfache, effiziente und gut wartbare Sprache für moderne Softwareentwicklung zu bieten. Go kombiniert die Performance und Sicherheit von kompilierter Software mit einer klaren, minimalistischen Syntax. Dabei kann man besonders hervorheben, dass es eingebaute Unterstützung für Multithreading durch Goroutines und eine umfangreiche Standardbibliothek gibt. Go eignet sich dank dieser Eigenschaften besonders gut für performante Backend-Systeme, Microservices und REST-APIs.

Auch wenn Go viele Vorteile bietet, gibt es ein paar Nachteile. Da wäre zum Einen das verbose und teilweise als unnötig empfundene Error-Handling zu nennen. In Go ist es extrem selten, dass eine Funktion durch einen Fehler das Programm beendet. Stattdessen geben fast alle Funktionen neben ihrem eigentlichen Rückgabewert zusätzlich eine Fehler-Variable zurück. Das kann zwar aufwändig sein, führt aber dazu, dass Go-Anwendungen By-Design sehr fehlerresistent sind.

Was auch noch als Nachteil von Go angeführt werden kann ist, dass die Sprache relativ zu anderen Sprachen eher jung ist, wodurch sich das Community-getriebene Ökosystem noch nicht auf den Stand anderer Sprachen entwickeln konnte. Im Rahmen dieses Projekts sind jedoch nur Standardfunktionen gefordert, die sogar fast komplett durch die Standardbibliothek abgedeckt werden können.

Abschließend haben wir entschieden, dass Go unsere Anforderungen für das Projekt erfüllt und in Anbetracht des geringen Umfangs des Projekts eine gute Möglichkeit ist, unser Wissen um Programmiersprachen zu erweitern.

Framework (Gin) vs. kein Framework

Zu Beginn der Entwicklung stand die Frage im Raum, ob der Einsatz eines Frameworks wie *Gin* für den Server sinnvoll ist. Dabei haben wir die jeweiligen Vor- und Nachteile analysiert und in Relation zur Komplexität unserer Anwendung gesetzt.

Die Verwendung von Go ohne zusätzliches Framework ermöglicht vollständige Kontrolle über den Request-Handling-Flow. Das *net/http*-Paket stellt eine solide Basis dar, auf der sich eigene Routing-Mechanismen, Middleware-Chains oder Logging-Lösungen gezielt und schlank implementieren lassen. Diese Herangehensweise fördert ein tiefes Verständnis für HTTP, Server-Design und Go-spezifische Konzepte wie Handler-Funktionen oder *http.ServeMux*. Darüber hinaus entfällt jegliche externe Abhängigkeit, was langfristig die Wartbarkeit erhöht und das Risiko von Breaking Changes oder Sicherheitslücken durch Drittanbieter reduziert. Insbesondere bei kleinen Projekten sind viele Features eines Frameworks – etwa komplexes Routing, Binding, Dependency Injection oder Plug-in-Systeme – schlicht überdimensioniert und würden zusätzlichen Overhead verursachen.

Auf der anderen Seite bietet ein Framework wie *Gin* zahlreiche Komfortfunktionen: deklaratives Routing, Middleware-Stacking, integriertes JSON-Binding, Error-Handling sowie standardisierte Strukturen für größere Projekte. Diese abstrahieren repetitive Aufgaben, reduzieren Boilerplate-Code und erhöhen dadurch die Entwicklungsproduktivität. Zudem ist Gin auf Performance optimiert und zählt zu den schnellsten Go-Frameworks. Der Performance-Impact gegenüber *net/http* ist in realistischen Szenarien meist vernachlässigbar.

Trotz dieser Vorteile haben wir uns bewusst gegen den Einsatz eines Frameworks entschieden. Aufgrund des geringen Funktionsumfangs und der überschaubaren Architektur unserer Anwendung überwiegen die Vorteile einer leichtgewichtigen, modularen Lösung auf Basis der Standardbibliothek. Dadurch bleiben wir flexibel, halten die Größe des finalen Programms gering und vermeiden unnötige Komplexität durch externe Abhängigkeiten.

Sollte die Anwendung zukünftig an Komplexität gewinnen (z. B. durch Authentifizierung, Versionierung von APIs oder modulare Erweiterbarkeit), kann der Wechsel zu einem Framework wie Gin oder Fiber jederzeit evaluiert werden.

Datenspeicherung

Für die Speicherung der Sessions und User haben wir uns für ein In-Memory Modell entschieden, bei dem der User- und Sessionspeicher jeweils ein Dictionary bzw. eine Map sind. Dabei sind entsprechend der login Name und die Session ID die Schlüssel und das korrespondierende Objekt jeweils eine Datenstruktur Namens User und Session:

```
type Session struct {
	ID        string
	UserID    string
	ExpiresAt time.Time
}

type User struct {
	LoginName string    `json:"loginName" validate:"required"`
	Password  Password  `json:"passwort" validate:"required"`
	FirstName string    `json:"vorname" validate:"required"`
	LastName  string    `json:"nachname" validate:"required"`
	Street    *string   `json:"strasse,omitempty"`
	ZipCode   *string   `json:"plz,omitempty"`
	City      *string   `json:"ort,omitempty"`
	Country   *string   `json:"land,omitempty"`
	Phone     *string   `json:"telefon,omitempty"`
	Email     *Email    `json:"email,omitempty"`
	Location  *Location `json:"-"`
}

type UserService struct {
	users    map[string]models.User    // In-memory store
	sessions map[string]models.Session // In-memory session store
	mu       sync.RWMutex              // For thread safety
}
```

Wir haben uns bei der Datenspeicherung gegen einen Datenbankservice wie MongoDB entschieden. Die Anforderungen bieten unserer Meinung nach keine Notwendigkeit für einen persistenten Speicher. Zudem ist es deutlich einfacher, eine In-Memory Map zu nutzen, als einen Speicherservice anzubinden.

Beim Aufbau der structs haben wir uns an den JSON Rückgabewerten der API orientiert. So konnten wir das Tag-Feature von Go benutzen, welches uns erlaubt, Felder mit beliebigen Zeichenketten zu taggen. Diese können dann ausgelesen und z.B. vom *encoding/json* Package zum Serialisieren/Deserialisieren von structs zu JSON Strings genutzt werden. Dadurch wird der Umgang mit JSON in Go sehr einfach gestaltet. Neben dem *json* Tag können auch andere, wie hier zu sehen z.B. der *validate* Tag, genutzt werden. Auf die Validierung von structs wird in einem späteren Kapitel noch näher eingegangen.

Neben den struct Tags haben wir noch eine andere Technik verwendet. Und zwar Pointer-Typen in structs. In Go haben alle Datentypen, die kein Pointer sind, eine Zero-Value. Diese Zero-Value ist wie ein Default, der schon bei der Deklaration, also noch vor der Initialisierung, einer Variable zugewiesen wird. Bei Ganzzahlen ist das *0*, bei Strings ist es *""* und bei Boolschen Werten *false*. Die Zero-Value eines structs ist einfach ein struct bei dem jedes Feld den Wert seiner jeweiligen Zero-Value hat. Dieses Design hat den Vorteil, dass jede Variable, die kein Pointer ist, immer einen Wert hat und so Fehlern vorgebeugt werden kann. Das Problem dabei ist, dass wenn man einen JSON String in ein Zero-Value struct deserialisieren möchte, man nicht zwischen ausgelassenen und leeren Variablen unterschieden kann. Wenn z.B. in meinem JSON String ein Feld den Wert *0* hat, kann dieser Wert im struct nicht mehr von der Zero-Value eines Integers (*0*) unterschieden werden. In diesem Fall kann man das Feld im struct als Pointer definieren. Das hat im restlichen Code, aufgrund der einfachen Funktionsweise von Go, wenig Auswirkungen. Jedoch ist die Zero-Value eines Pointers immer *nil*. Ein ausgelassenes Feld würde somit den Wert *nil* statt z.B. *0* haben. Diese Technik wird oft bei optionalen Feldern verwendet.

Anfragen an den API-Server kommen nicht immer nacheinander, sondern manchmal auch gleichzeitig an. In so einem Fall ist es wichtig, dass zwei schreibende Funktionen in verschiedenen/parallelen Threads nicht gleichzeitig den User- oder Sessionspeicher beschreiben. Um dieses Problem zu lösen, benutzen wir den *RWMutex* aus dem *sync* Package. Mit diesem können die beiden Maps gelocked werden, wenn sie beschrieben werden. Wenn die Maps fürs Schreiben gelocked sind, können sie nicht gelesen werden, aber wenn sie nur zum Lesen gelocked sind, können andere Threads gleichzeitig auch den Wert lesen.

Nutzung von DTOs

Wir haben uns für den Umgang mit JSON Nachrichten für das DTO Entwurfsmuster entschieden. Das hat zum einen den Vorteil, dass die Daten gekapselt werden und so eine saubere Trennung zwischen interner Logik und externer Datenrepräsentation möglich ist. Zum Anderen ist es in Go, im Vergleich zu anderen Sprachen wie z.B. TypeScript, sehr schwer mit arbiträren unstrukturierten JSON Daten zu arbeiten, da Go statisch typisiert ist und so vor jedem Zugriff auf ein Feld erst geprüft werden müsste ob das Feld wirklich existiert und ob es den erwarteten Datentyp hat. DTOs ermöglichen uns daher einen sinnvollen Umgang mit JSON Daten.

```
type LoginRequest struct {
	LoginName string `json:"loginName" validate:"required"`
	Password  struct {
		Password string `json:"passwort" validate:"required"`
	} `json:"passwort" validate:"required"`
}

type LoginResponse struct {
	SessionID string `json:"sessionID"`
}
```

Um die Datenstrukturen (DTOs) korrekt zu JSON zu serialisieren bzw. JSON Strings korrekt in DTOs zu deserialisieren, machen wir hier Gebrauch von Feldtags in Go. So können wir durch den *json*-Tag definieren, was ein gegebenes Feld in serialisierten JSON heißt. Dadurch kann später der JSON Encoder/Decoder die korrekte Zuweisung von JSON-Feldern zu Datenstruktur-Feldern erkennen.

Request Validation (go-playground/validator)

Für die Validierung von eingehenden Requests nutzen wir das Paket github.com/go-playground/validator/v10, das eine deklarative Möglichkeit bietet, Validierungsregeln direkt in den Struct-Tags unserer DTOs (Data Transfer Objects) zu definieren.

Validierung mittels Go Struct Tags

Die Validierung in Go erfolgt elegant durch Struct-Tags, die direkt an den Feldern unserer Datenstrukturen definiert werden. Hier ein Beispiel aus unserem Projekt:

```
type User struct {
	LoginName string    `json:"loginName" validate:"required"`
	Password  Password  `json:"passwort" validate:"required"`
	FirstName string    `json:"vorname" validate:"required"`
	LastName  string    `json:"nachname" validate:"required"`
	Street    *string   `json:"strasse,omitempty"`
	ZipCode   *string   `json:"plz,omitempty"`
	City      *string   `json:"ort,omitempty"`
	Country   *string   `json:"land,omitempty"`
	Phone     *string   `json:"telefon,omitempty"`
	Email     *Email    `json:"email,omitempty"`
	Location  *Location `json:"-"`
}

type Password struct {
	Password string `json:"passwort" validate:"required,min=6"`
}

type Email struct {
	Address string `json:"adresse" validate:"email"`
}
```

Die wichtigsten Validierung Tags, die wir verwendet haben, sind:

- required: Feld darf nicht leer sein
- min: Längenbegrenzung für Strings
- email: Validiert die Email-Adresse

Vorteile der Implementierung

Die Validierung mit go-playground/validator ist besonders einfach umzusetzen. Nach der Initialisierung des Validators können Requests mit einem einzigen Aufruf geprüft werden:

```
if err := validator.Struct(user); err != nil {
	fmt.Println(err)
	pkg.JsonError(w, pkg.GenericResponseJson("Fehler", "Invalid
body"), http.StatusBadRequest)
```

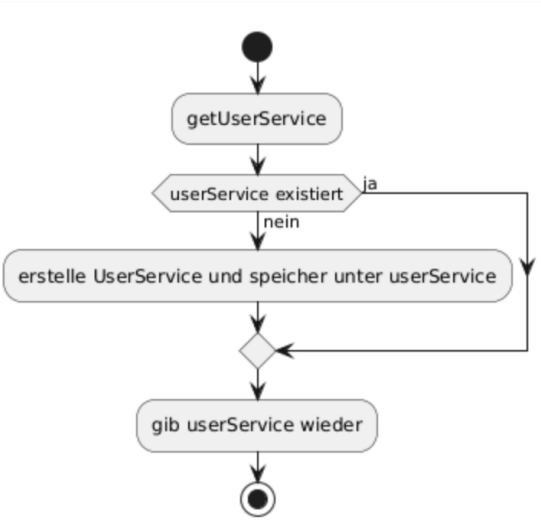
```
        return
}
```

Ein großer Vorteil dieses Ansatzes ist, dass die Validierungsregeln direkt bei den Datenstrukturen definiert sind. Wenn Änderungen an den DTOs vorgenommen werden müssen, können die Validierungsregeln unmittelbar angepasst werden, was die Wartbarkeit des Codes deutlich verbessert. Diese Kopplung von Datenstruktur und Validierungslogik an einer Stelle reduziert die Fehleranfälligkeit und erleichtert zukünftige Erweiterungen.

Der Validator bietet zudem die Möglichkeit, benutzerdefinierte Validierung Funktionen zu registrieren, was es uns erlaubt, spezifische Geschäftsregeln für den FAP-Server zu implementieren.

Nutzung des Singleton Entwurfsmusters

Das Herzstück der Implementation bildet der *UserService*. Dieser bietet eine strukturierte Schnittstelle zum Datenspeicher und ist mit gezielt gewählten Abstraktionen ausgestattet, um die Zugriffe auf den Datenspeicher zu vereinheitlichen. Für die Implementation des *UserService* wurde sich am Singleton Pattern bedient. Neben einem reduzierten Speicherverbrauch sind, besonders für den Zugriff auf Datenspeicher, die Vorteile von gezielter Zugriffskontrolle und eines eindeutigen Zugriffspunkts entscheidend. Die Zugriffskontrolle ist dahingehend relevant, dass durch die Nutzung spezialisierter Methoden, Validierungsmechanismen einmalig implementiert und bei weiteren Zugriffen auf die Daten nicht vergessen werden. Dazu kommt durch den eindeutigen Zugriffspunkt, dass Anpassungen nur an einer Stelle erfolgen und ein klarer Kommunikationsweg zu den Daten besteht, was die Wartbarkeit des Systems verbessert. Umgesetzt wird das Singleton-Pattern, indem beim ersten Aufruf eine Instanz erstellt wird. Nachfolgende Aufrufe bedienen sich dann der schon erstellten Instanz, anstelle neue Instanzen zu erstellen. Abstrakt lässt sich dieser Ablauf wie folgt definieren.



Von einer externen Stelle wird eine Funktion aufgerufen. Als erstes wird in der Funktion geprüft, ob schon eine Instanz besteht. Ist dies nicht der Fall wird ein Instanzierungsprozess durchlaufen, welcher spezifisch für das zu erstellende Konstrukt ist. Das Ergebnis wird dann abgespeichert und wiedergegeben. In einem nächsten Aufruf ist das Objekt schon vorhanden und kann direkt wiedergegeben werden. Wichtig dabei ist, alle anderen Wege, um das Objekt zu erstellen, zu blockieren. Andernfalls könnte der Prozess ungewollt ausgehebelt und die Vorteile negiert werden. Eine mögliche Implementation in Go sieht wie folgt aus.

```
func NewUserService() *UserService {
   return &UserService{
      users:    make(map[string]models.User),
      sessions: make(map[string]models.Session),
   }
}
```

Die Funktion *NewUserService* erstellt einen *UserService* mit *users* und *sessions*.

```
var userService = services.NewUserService()
```

Danach wird diese Funktion einmalig aufgerufen und das Ergebnis in der Variable *userService* gespeichert. Die Variable *userService* erlaubt nun den Zugriff auf den *UserService.*

```
func AddUserHandler(w http.ResponseWriter, r *http.Request) {
   ...
   userExists := userService.UserExists(user.LoginName)
   if userExists {
      ...
   }

   _ = userService.AddUser(user)

   ...
}
```

Das wurde ebenfalls für den zuvor definierten *Validator* gemacht und so ein klarer Datenfluss durch die Applikation definiert und garantiert.

Besondere Teile der Implementierung

Der Großteil der Anwendung konnte mit Standardfunktionen des *net/http* packages gut abgedeckt werden und beinhaltet sonst wenig bemerkenswerten Code. Trotzdem möchten wir an dieser Stelle den Mechanismus zum Aufräumen der abgelaufenen User Sessions anbringen. Dieser verwendet nämlich gleich mehrere Besonderheiten der Programmiersprache Go:

```
func init() {
      go func() {
            ticker := time.NewTicker(1 * time.Hour)
            for range ticker.C {
                  userService.CleanupSessions()
            }
      }()
}
```

Der Mechanismus wird gestartet durch die *init* Funktion des *handlers* Packages. Eine *init* Funktion in Go muss nicht vom Entwickler aufgerufen werden, sondern wird ausgeführt, wenn das entsprechende Package importiert wird. Dabei ist es egal, wie oft das Package importiert wird, die *init* Funktion wird immer genau ein Mal ausgeführt. Das ist wichtig, denn sonst würden mehrere Cleanup-Loops gestartet

werden. Als nächstes wird eine Goroutine gestartet, die den Cleanup-Loop ausführen wird. Eine Goroutine kann als ein leichtgewichtiger und effizienter Thread verstanden werden und ist hier notwendig, da sonst der Cleanup-Loop den Haupt-Thread blockieren würde. Das letzte Puzzleteil ist die Nutzung von Channels. Ein Channel in Go ist ein eigener eingebauter Datentyp und repräsentiert, grob gesagt, eine Queue, die ausgelesen und beschrieben werden kann. Das *ticker*-Objekt im Code beinhaltet einen Channel, in dem in regelmäßigen Intervallen (hier eine Stunde) Werte geschrieben werden. Die *for*-Schleife blockiert die Goroutine solange bis ein nächster Wert verfügbar ist, und führt den Body der Schleife aus, wenn dies der Fall ist. So wird, solange das Programm läuft, jede Stunde die Methode aufgerufen, die abgelaufenen User Sessions invalidiert.

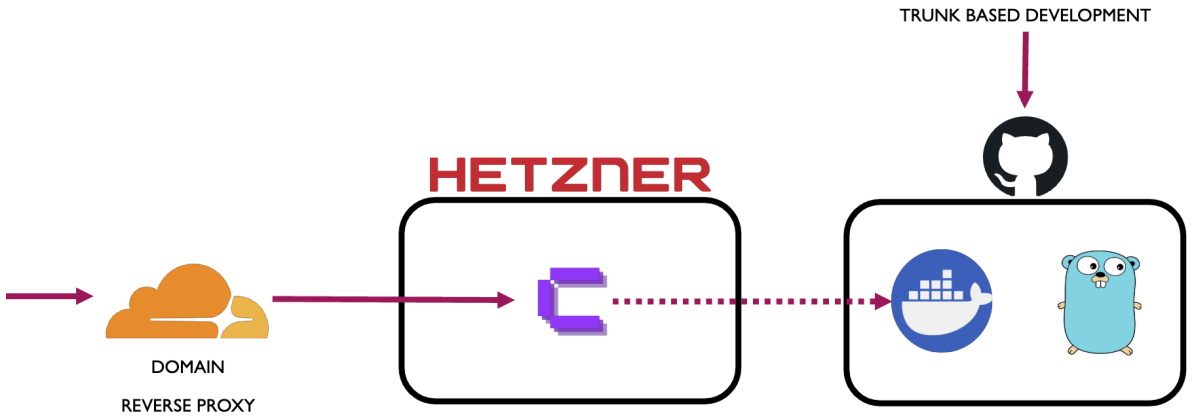
Eine weitere Besonderheit der Sprache Go, die wir bei unserer Implementation verwendet haben, ist das *defer* Keyword:

```
func (s *UserService) UserExists(username string) bool {
	s.mu.RLock()
	defer s.mu.RUnlock()
	_, ok := s.users[username]
	return ok
}
```

In dieser Funktion nutzen wir einen Mutex, der (vereinfacht) am Anfang der Funktion den Speicher sperren, und ihn am Ende der Funktion wieder freigeben soll. In Go haben wir es jedoch leicht und müssen nicht alle Funktionen, die vor dem return einer Funktion ausgeführt werden sollen, zu allen jeweiligen return Statements schreiben. Es kann dafür das defer Keyword verwendet werden, welches den angegebenen Funktionsaufruf verschiebt, bis die umschließende Funktion returned. Das führt im obigen Beispiel dazu, dass der Speicher automatisch wieder freigegeben wird, sobald die Funktion beendet ist. Dieser Mechanismus ist besonders praktisch, wenn mit Mutex, Dateien oder Streams gearbeitet wird und die Funktion mehrere verschiedene return-Punkte hat.

Deployment

Zu guter Letzt soll das Deployment der Applikation thematisiert werden.



Die Code-Änderungen werden durch die Code-Verwaltungsplattform Github gespeichert. Code-Änderungen werden durch das Branching-Modell *Trunk Based Development* synchronisiert. Dabei stehen viele kleine Änderungen, die direkt in den Main-Branch gelangen, im Vordergrund. Feature-Branches gibt es keine, was die Integration von Änderungen bei dieser Projektgröße beschleunigt hat. Die Go Applikation wird durch ein Docker Image deployed. Das Docker Image liegt ebenfalls in dem Github-Repository. Die Applikation läuft auf einem Hetzner Dedicated Server über Coolify. Coolify ist ein OpenSource Tool, welches die Verwaltung von Server Anwendungen vereinfacht. Die Github-Integration erlaubt es, Coolify mit Github zu verbinden und Anwendungen einfach bereitzustellen und bei Änderungen zu synchronisieren. Damit die IP-Adresse des Servers geheim bleibt und die Anwendung einfacher anzusprechen ist, ist ein Cloudflare DNS zwischengeschaltet. Die integrierte Reverse-Proxy-Funktion schützt die IP-Adresse und erlaubt den einfachen Zugriff durch einen Domainnamen. Für dieses Projekt wird *fap.krempin.cloud* verwendet.

Fazit

Kritische Bewertung des Projekts

Die Aufteilung der Gruppe hat gut funktioniert. Die Anforderungen wurden initial in mehrere Aufgaben aufgeteilt und in einem Backlog hinterlegt. Aus diesem Backlog wurden die Aufgaben eigenhändig abgearbeitet. Durch eine fehlende feste Zuordnung der Teilaufgaben hat sich die Last ungleichmäßig verteilt, was jedoch auf die unterschiedlichen Level an Expertise zurückzuführen ist und nicht fehlender Eigeninitiative. Trotz der asynchronen Arbeitsweise bestand ein umfangreicher Austausch bei Problemen und Fortschritt wurde, nicht zuletzt durch das Aufgaben-Backlog, regelmäßig kommuniziert. Dadurch konnte das Projektziel schnell und ohne weitere große Schwierigkeiten erreicht werden.

Im Hinblick auf die Implementation war es ungewohnt, eine deutschsprachige API zu schreiben, da in den Unternehmen und generell in der Informatik in Englisch gearbeitet wird. Neben einer Umstellung hat dieser Aspekt aber nicht weiter zu Problemen in der Umsetzung geführt.
Zu guter Letzt sind wir sehr zufrieden mit der Wahl der Programmiersprache Go. Wir konnten erste wichtige Einblicke erhalten und wurden mit neuen Denkweisen von einer Programmiersprache konfrontiert, die nicht im Unternehmenskontext genutzt wird. Die erlernten Go Paradigmen sind für unseren Arbeitsalltag relevant und somit bietet das Projekt, selbst nach dem Abschluss, noch einen nachhaltigen Mehrwert. Besonders an Go hat uns gefallen, dass die Nutzung sehr naheliegend und einfach ist, wodurch schnell guter und lesbarer Code geschrieben werden konnte. Was sich besonders als neue Programmiersprache für dieses Projekt als sehr wertvoll bewährt hat. Nachteile von Go konnten wir in dieser kurzen Zeit noch nicht herauskristallisieren. Schließlich besteht bei allen das Interesse, sich im privaten oder im beruflichen Kontext weiter mit Go zu beschäftigen.

Geschäftsmodelle

Basierend auf der finalen Applikation lassen sich mehrere Geschäftsmodelle aufbauen. Als erstes ist denkbar, ein ähnliches System wie “Find my Device” von

Apple / Google aufzubauen, nur für Familie und Freunde. Besonders relevant kann ein solches Feature für Familien mit Kindern sein. Damit Eltern im Notfall wissen, wo sich das Kind aufhält, teilt die Applikation alle 5 Minuten ihren Standort mit den Eltern. In diesem Fall ist es jedoch essentiell, weitere Sicherheitsmaßnahmen zu implementieren, um die kritischen Standortdaten ausreichend zu schützen. Weniger systemkritisch ist eine “Play with me” Applikation, mit der Fremde zu Freunden werden können. Beispielsweise ist die App “beerwithme” zu nennen, mit welcher der Standort geteilt wird und Nutzer in der Nähe somit eingeladen werden “zusammen Bier zu trinken”. Das Konzept lässt sich um beliebige Aktivitäten erweitern.