Examensarbete, Högskolan på Åland, Utbildningsprogrammet för Informationsteknik

MIGRERING AV EN BEFINTLIG WEBBAPPLIKATION

- från funktionsorienterad till objektorienterad kod i PHP

Kjell Hansen

2022:38

Datum för godkännande: 30.9.2022
Handledare: Björn-Erik Zetterman

EXAMENSARBETE
Högskolan på Åland

Utbildningsprogram:	Informationsteknik
Författare:	Kjell Hansen
Arbetets namn:	Migrering av en befintlig webbapplikation - från funktionsorienterad till objektorienterad kod i PHP
Handledare:	Björn-Erik Zetterman
Uppdragsgivare:	Kjell Hansen

Abstrakt

Mitt examensarbete beskriver en migrering av en webbplats i PHP konverteras från att vara funktionsorienterad till en modernare design i objektorienterad programmering.

Bakgrunden och syftet ska ge en inblick i hur man kan skapa en objektorienterad databasdriven webbplats i PHP med stöd av färdiga komponenter utan att använda något ramverk.

Nyckelord (sökord)

PHP, objektorienterad programmering, sql, komponentbibliotek, designmönster, migrering

Högskolans serienummer:	ISSN:	Språk:	Sidantal:
2022:38	1458-1531	Svenska	62

Inlämningsdatum:	Presentationsdatum:	Datum för godkännande:
8.6.2022	30.9.2022	30.9.2022

DEGREE THESIS
Åland University of Applied Sciences

Degree Programme:	Information Technology
Author:	Kjell Hansen
Title:	Migration of an Existing Web Application - from functions oriented design to object oriented code in PHP
Academic Supervisor:	Björn-Erik Zetterman
Commissioned by:	Kjell Hansen

Abstract

In my thesis I write about how I migrated a website in PHP from a functions oriented design to a modern object oriented design.

The background and my purpose is to show how you can create an object oriented database driven website in PHP without resorting to any frameworks but still use ready-to-use components.

Keywords
PHP, object oriented programming, sql, component library, design pattern, migration

Serial number:	**ISSN:**	**Language:**	**Number of pages:**
2022:38	1458-1531	Swedish	62 pages

Handed in:	**Date of presentation:**	**Approved:**
8.6.2022	30.9.2022	30.9.2022

INNEHÅLLSFÖRTECKNING

1 INTRODUKTION

1.1 Bakgrund

Under åren 2004-2006 utvecklade jag en webbsite i PHP/mySQL för att hålla reda på de åländska simresultaten. Från början var tanken bara att ha med de aktuella resultaten, men eftersom tiden gick blev de historiska resultaten också intressanta och lades in.Webbsajten skapades så att simtränarna kunde få aktuell information om personliga rekord, åländska rekord, personlig resultatutveckling mm. Sajten utvecklades funktionsorienterat dels för att jag inte behärskade objektorienterad programmering och dels för att PHP:s stöd för objektorienterad programmering var dåligt utvecklat vid den tiden. 2009-12 gjordes en smärre omarbetning av sajten och en migrering gjordes till PHP5. I samband med det lades det till lite fler funktioner och fler statistikuppgifter gick att få fram via sidorna. Allting var dock fortfarande funktionsorienterat.

I samband med att PHP5 nådde sitt End-of-Life i slutet av 2018 började jag fundera på hur jag skulle bygga sajten objektorienterat i PHP7. Många böcker och online-resurser senare stötte jag på boken Professional PHP av Patrick Louys som fick mig att få upp ögonen för hur man kunde utveckla specialanpassade webbsajter med hjälp av PHP och färdiga komponentbibliotek tillgängliga via pakethanteraren Composer.

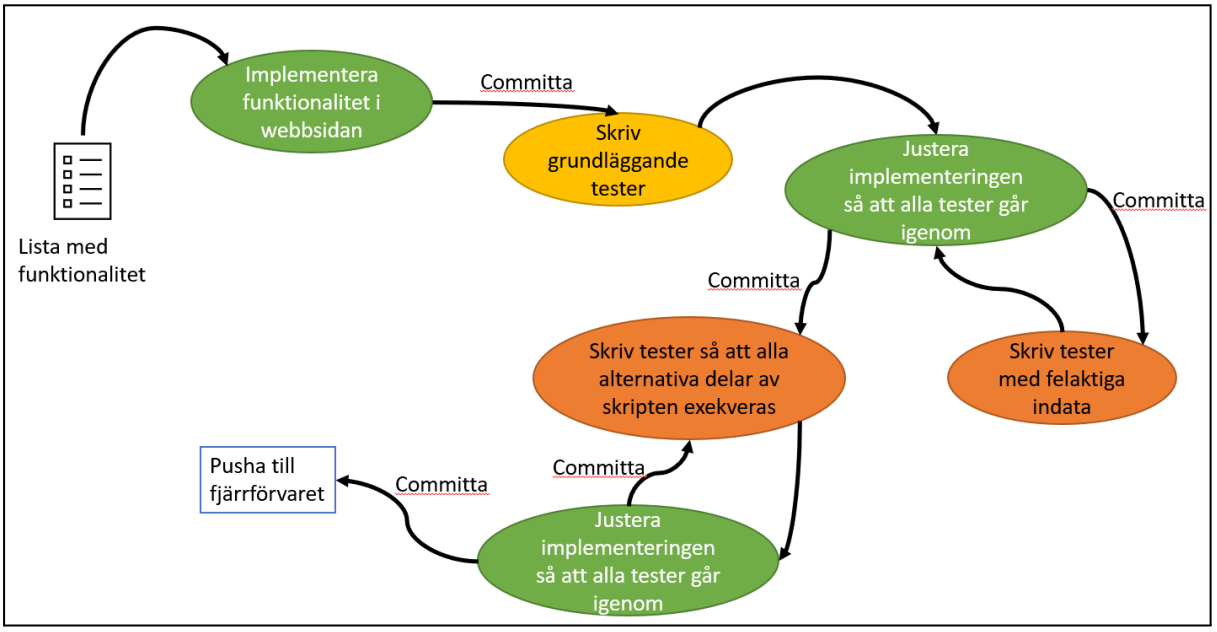
Även databasen behövde göras om, den skapades i en version av mySQL som inte stödde främmande nycklar och all kontroll av referensintegritet sköttes manuellt. Dessutom finns en hel del dubblerad data och fält som bara används ibland.

1.2 Syfte

Syftet med uppsatsen är att beskriva hur man kan bygga en lättunderhållen, dynamisk, databasdriven site i PHP utan att använda något ramverk men ändå utnyttja färdiga komponenter.

1.3 Metod

Den metod som jag använder mig av vid utvecklingen kan väl närmast beskrivas som *evolutionary prototyping(Floyd, 1984)*. Jag har en lista med önskvärd funktionalitet som jag har sorterat i prioritetsordning. Jag tar den funktionaliteten som ligger högst på prioritetslistan och gör en implementering av den så att den fungerar för de mest grundläggande fallen via användargränssnittet, därefter skriver jag tester som testar den funktionaliteten som jag har sett fungerar i användargränssnittet. När de testerna är skrivna och klara skapar jag nya testfall där jag skickar in felaktig parameterdata och ser att de testerna ändå ger förväntat resultat antingen direkt eller efter att jag justerat koden. Till sist skriver jag tester där jag försöker få alla alternativa delar av skripten för att exekveras. Den här proceduren gör jag för alla klasser som är inblandade i den funktionaliteten som jag utvecklar. När allt gått igenom snyggar jag upp koden och ser till att få den kommenterad i de delar där det inte är helt klart vad som sker. När det är klart är det dags att pusha alla commits och påbörja nästa funktionalitet från prioritetslistan. Se figur 1.



Figur 1. Översiktlig metod

1.4 Avgränsningar

Sajten kommer inte att byggas klart under arbetet med denna uppsats, det är alltför omfattande arbete som återstår för att det ska vara möjligt. Uppsatsen beskriver strategier och

metoder för sajtens uppbyggnad och hur arbetet framskridit så här långt samt en del beskrivningar av hur några vitala delar av framtida funktionalitet planeras att implementeras.

1.5 Definitioner

Nedanstående lista ger en kort beskrivning av termer och förkortningar som används i denna uppsats:

- **Ajax**: **(***Asynchronous JavaScript and XML*) är en teknik för att utnyttja Javascripts möjligheter till asynkron exekvering för att hämta serverdata utan att behöva ladda om sidan. Ursprungligen hämtades endast XML-dokument men numera kan all form av data hämtas av klienten (Wikipedia contributors, u.å.).
- **API**: (*Application Programming Interface*) är ett sätt för en server att tillhandahålla tjänster för att komma åt data. Servern definierar hur man ska anropa och vilket svar som erhålls och sen kan olika klienter komma åt data via gränssnittet (Wikipedia contributors, u.å.-a).
- **Backend**: Inom mjukvaruutveckling betecknar *backend* mjukvara som exekveras på servern och som alltså inte är tillgänglig på klienten.
- **Composer**: Composer är en pakethanterare på applikationsnivå för programmeringsspråket PHP som ger ett standardiserad hantering av beroenden mellan PHP och nödvändiga bibliotek (Wikipedia contributors, 2022a).
- **CSS**: (*Cascading Style Sheets***)** stilmallar för att definiera utseendet på olika delar av en webbsida. Stilmallarna har strikta regler för när de enskilda stilarna övertrumfar tidigare direktiv därav beteckningen *cascading*, överlappande på svenska (Wikipedia contributors, u.å.-b).
- ***Depency injection*** *(relevant svensk beteckning saknas)***:** ett objekt kan vara beroende av att andra objekt redan finns och man kan skicka in dessa objekt till det objektet som ska skapas. Det finns olika strategier för att vara säker på att objekten skapas i rätt ordning för att sen i sin tur kunna användas för att skapa nya objekt.
- **Enhetstestning**: (*Unit testing*) Ett standardiserat, automatiserat sätt att testa klasser och funktioner. Som utvecklare skriver man tester och kan sen välja att köra alla eller bara vissa. När man lagt till funktionalitet kan man köra samma tester igen för att se att den nya funktionaliteten inte förstört någon tidigare funktionalitet. Målet med

enhetstestning är att varje isolerad del av programmet ska fungera enligt specifikation (Wikipedia contributors, 2022c).

- **Frontend**: Inom mjukvaruutveckling betecknar *frontend*-programmering kod som exekveras på klienten, dvs i användarens webbläsare när det handlar om webbprogrammering. Användaren kan både se och stega igenom koden som finns i klartext i webbläsaren. Det dominerande frontend-språket är Javascript.
- **HTML**: (*HyperText Markup Language*) är ett uppmärkningsspråk för att definiera olika delar av ett dokument med hjälp av taggar, ord eller förkortningar inom <> som definierar olika delar av dokumentet..
- **IndexedDB**: IndexedDB är en standardiserad databashanterare för JSON-objektdatabaser som finns inbyggd i moderna webbläsare sen ungefär 2012 men blev standardiserat först 2015 (Wikipedia contributors, 2022m).
- **IoC (*Inversion of Control*)**: Är en programmeringsprincip som bygger på att klasser ska inte vara beroende av andra klasser utan beroenden ska vara gränssnitt istället. Det gör att man enklare kan byta ut delar av implementeringen utan att andra delar av källkoden påverkas (Wikipedia contributors, 2022d).
- **Glass box-testing**: Testning av programkod där man kan se koden som ska testas och därför lättare hitta vilka tester som behöver göras för att säkerställa att så mycket kod som möjligt blir exekverad genom olika testfall (Wikipedia contributors, 2021).
- **Javascript**: (I marknadföringssyfte ofta JavaScript) är ett skriptspråk som exekveras i webbläsaren. Eftersom det exekveras på användarens dator är det läsbart och möjligt för användaren att stoppa exekveringen eller stega sig fram genom källkoden. Det finns strikta regler för vad på klienten som Javascript kan komma åt och därmed kan det inte ställa till problem på användarens dator (Wikipedia contributors, u.å.-c).
- **JSON**: (*JavaScript Object Notation*) är ett textbaserat format för att skicka data mellan servern och klienten. Det finns endast ett fåtal olika datatyper och dessa är universella och finns i alla programmeringsspråk (Wikipedia contributors, u.å.-e).
- **Jquery**: Ett Javascript-bibliotek som förenklar bland annat hantering av en webbsidas uppmärkningsspråk, CSS, ajax-anrop (Wikipedia contributors, u.å.-d)**.**
- **mySQL**: En relationsdatabashanterare som är en av de mest populära databashanterarna för internetapplikationer. Den skapades i mitten av 1990-talet och är licensierad som fri programvara. I samband med version 5.5 introducerades

InnoDB som en databasmotor vilket innebar att man kunde skapa relationer med främmande nycklar och att databasen kunde upprätthålla referensintegritet för relaterade tabeller (Wikipedia contributors, 2022j).

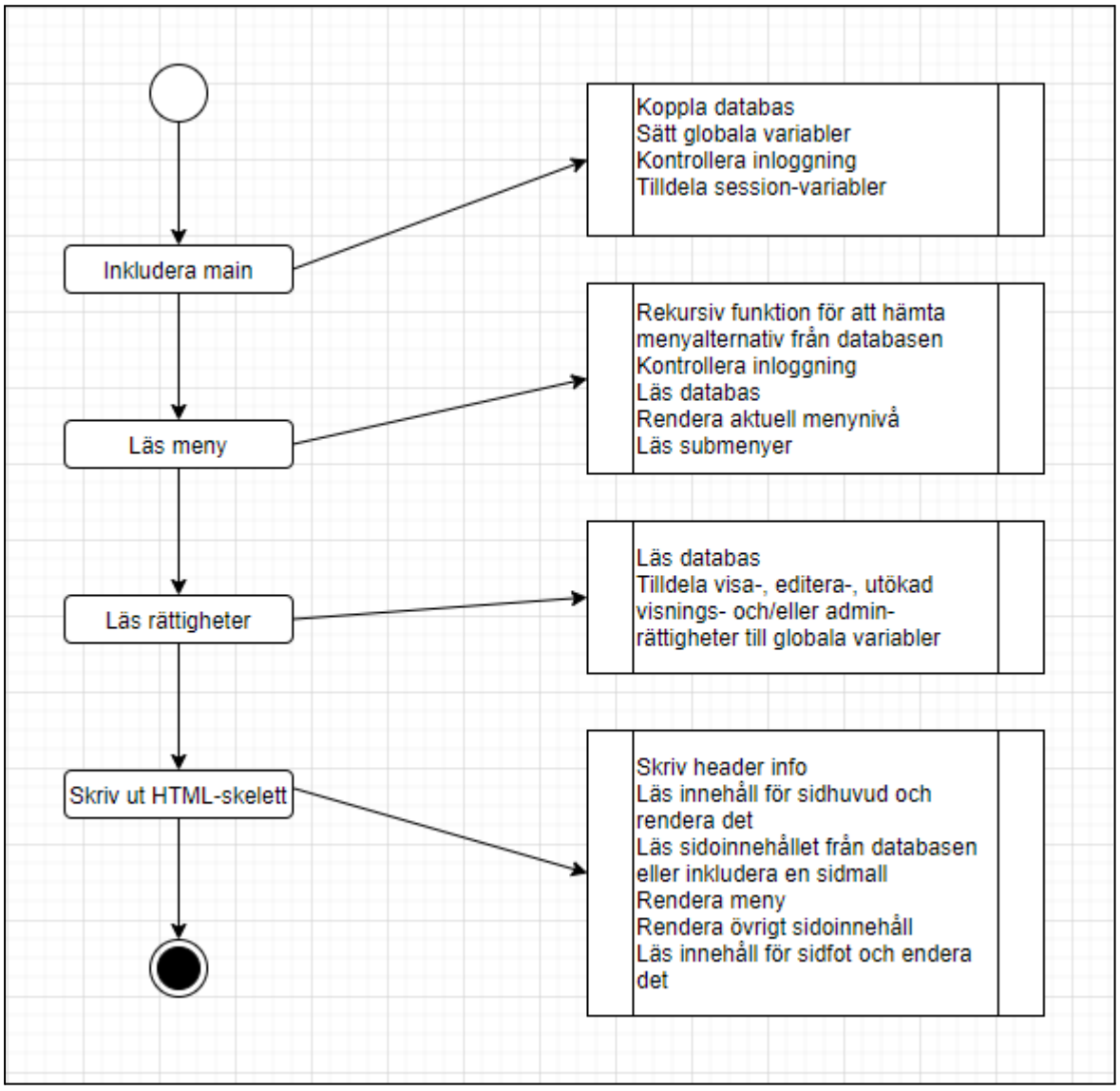
- **MVC**: (*Model-View-Controller*) är ett välanvänt och väldefinierat arkitekturmönster för mjukvaruutveckling. Det bygger på *separation of concerns* alltså att alla delar bara ska göra sin del (Wikipedia contributors, u.å.-f).
- **PHP**: (*PHP Hypertext Preprocessor*) är ett skriptspråk som körs via en webbserver för att visa dynamiskt skapade webbsidor (Wikipedia contributors, u.å.-g).
- **PHPDoc**: Ett sätt att automatiskt dokumentera klasser och funktioner i php. Det är egentligen en speciell form av kommentar där utvecklingsmiljön hjälper till på basen av deklarationer skapa dokumentation av källkoden (Wikipedia contributors, 2022b).
- **POJO**: (*Plain old java objects*) ursprungligen endast för java-objekt men används i överförd betydelse även för andra programmeringsspråk för att visa att man använder enkla klasser utan några beroenden ofta för att läsa och skriva databasposter (Wikipedia contributors, 2022i).
- **PWA**: (*Progressive Web Application*) en webbapplikation som uppför sig och fungerar till största delen som en "riktig" app i en telefon eller surfplatta. Det är egentligen bara en webbsida men den har funktioner för *off line*-laddning och användning (Wikipedia contributors, 2022k).
- **Sass**: (*Syntactically awesome style sheets*) är en preprocessor för att att skapa css-filer. Man skriver vanliga css-direktiv, men det finns möjlighet att nästla dem på ett mer intuitivt sätt än hur det ska göras i ren css. Det finns också möjligheter att använda funktioner och variabler i sass-filen (Wikipedia contributors, 2022g).
- **Sessionsvariabel**: Ett namn/värde-par som lagras i serverns RAM-minne och som kan nås via PHP-skript från olika anrop från samma klient.
- **Stub**: En klass eller funktion som implementerar samma interface som den riktiga klassen, men som alltid returnerar samma värde. Används i testsammanhang för att kunna testa komplexa anrop med hjälp av kända och mindre komplexa funktioner (Wikipedia contributors, 2022h).
- **Twig**: Ett mallhanterare för PHP-skript. Med hjälp av Twig kan man på ett säkert sätt generera webbsidor utifrån en mall. Det finns ett antal enkla funktioner i Twig som

gör att man kan loopa igenom variabler och formatera värden på olika sätt (Wikipedia contributors, 2022e).

- **UTF-8**: En Unicode teckenkodning där olika tecken har olika längd. Unicode-tecken kan använda upp till 21 bitar för att definiera ett tecken och det ryms inte i en vanlig byte (8 bitar) därför kommer vissa tecken att bestå av tre oktetter, andra av två och ytterligare andra av en byte. UTF-8 är ett vanligt sätt att koda webbsidor eftersom all världens tecken kan visas (Wikipedia contributors, u.å.-h).
- **UUID**: (*Universally unique identifier*) Ett 128 bitars tal som visas med hexadecimala tal i grupper åtskilda med bindestreck. 4-2-2-2-6 hexadecimala tal finns i vardera gruppen. Version 4 som jag använder i mitt projekt har 5.3×10^{36} olika kombinationer (Wikipedia contributors, 2022f).
- **WYSIWYG-redigering**: (*What you see is what you get*) Redigering av text så att man ser det färdiga utseendet redan när man redigerar texten.
- **XML**: (*Extensible Markup Language*) Ett uppmärkningsspråk, och filformat, för att lagra, ta emot, skicka och rekonstruera komplexa objekt i textformat. Språket definierar ett antal regler för att koda dokument som är läsbara både för maskiner och människor (Wikipedia contributors, 2022l).

2 TIDIGARE STRUKTUR

Den tidigare sajten hanterade sidförfrågningar via id-nummer. Alla sidor laddades via index.php, och beroende på vilket id som skickades till sidan inkluderades olika “sidmallar”. Mallarna var halvfärdiga HTML-dokument, som innehöll själva sidans innehåll. Allt som finns runt om som logo, sidrubriker, menyer, innehåll i sidopaneler och i sidfoten renderades från funktioner som anropades från index-sidan. Via index-sidan sköttes också rättigheter till olika sidor, innehållet i sidopaneler och till viss del lite av vilket innehåll som presenterades på själva sidan. Efter inloggning lagrades användar-ID:et i en sessionsvariabel och med dess hjälp lästes rättigheter från databasen och olika innehåll presenterades för användaren. Figur 2 visar flödet i den tidigare sajten.



Figur 2. Tidigare arkitektur

En typisk sida innehöll ett antal kontroller av inskickad data, eventuell Javascript för just den sidan, och sedan innehållet på sidan. Rent fysiskt låg alla php-filerna i olika mappar i webbroten och var alltså i princip nåbara för vem som helst som visste vad filen hette.

3 ARKITEKTUR OCH ANVÄNDA DESIGNMÖNSTER

3.1 Designmönster

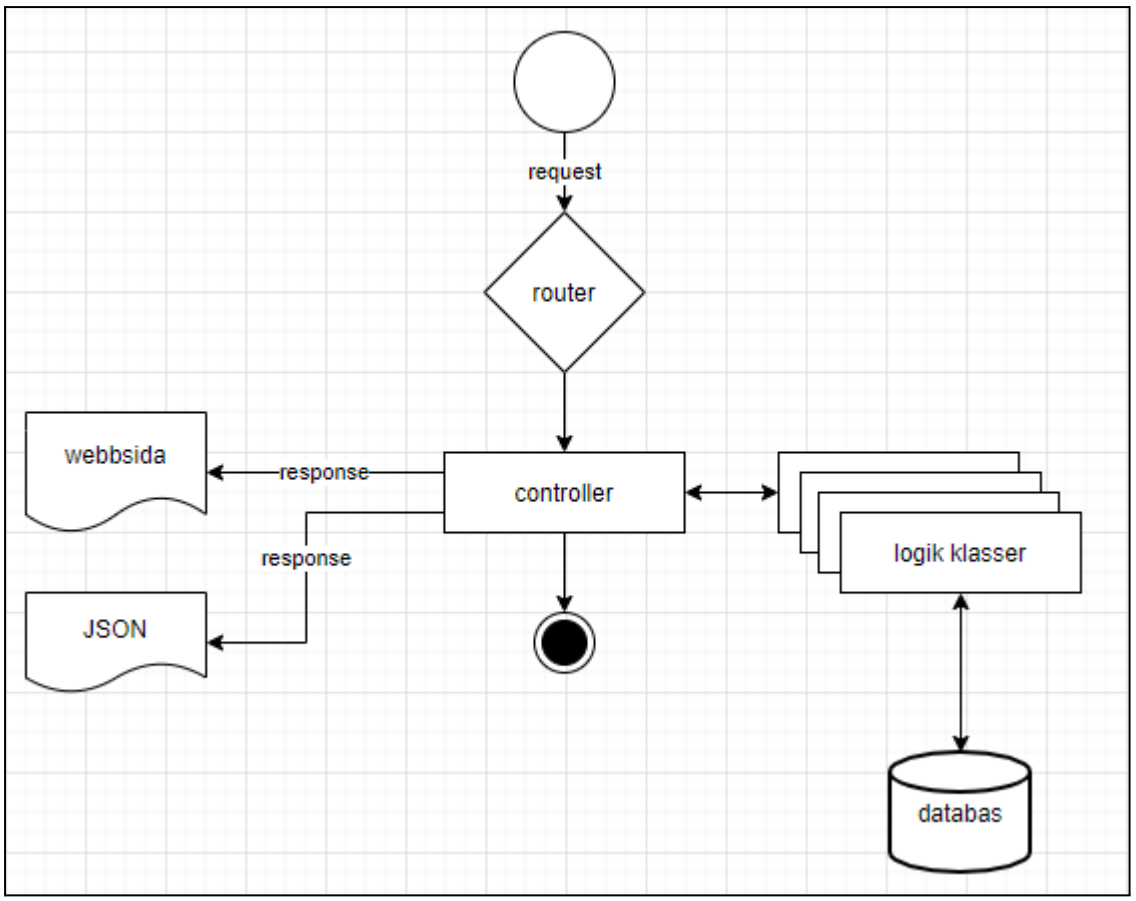
Ett designmönster är en beskrivning av hur man rent designmässigt löser ett känt problem på ett genomtänkt och välbeprövat sätt. Termen designmönster kommer ursprungligen från arkitekturen men började i slutet på 1990-talet tillämpas i överförd mening på programmeringsproblem. 1994 skrevs standardverket “Design Patterns, Elements of Reusable Object-Oriented Software” skrevs av fyra mjukvaruutvecklare som kom att kallas “The Gang of Four”(Gamma m.fl., 2016). I mitt projekt har jag använt mig av 6 av de 23 designmönster som finns i boken.

3.2 MVC

MVC (*Model-View-Controller*) I många system avser *model* datalagringen och mjukvarukomponenter som interagerar med databasen, *view* avser användargränssnittet och *controller* är de delar av mjukvaran som bearbetar och svarar på händelser som sker i användargränssnittet och som påverkar innehållet i både *view*- och *model*-delarna av mjukvaran

I min jakt på en lämplig arkitektur läste jag en hel del böcker och online-resurser om MVC-mönstret och jag tyckte att det lät som ett bra angreppssätt även om jag inte riktigt tyckte att det passade in. I Louys bok avfärdar han det designmönstret med hänvisning till att designmönstret inte är skapat för en request/response-miljö utan för användargränssnitt i en desktop-applikation med kontinuerlig koppling till en server. Därmed har utvecklare förändrat och anpassat MVC-konceptet efter egna önskemål till en sån nivå att det är svårt att hitta två utvecklare som är överens om exakt vad MVC-mönstret innebär(Louys, 2018, s. 77). Den arkitekturen som min implementation bygger på är inspirerad av MVC-mönstret men jag skulle inte säga att det är en MVC-arkitektur. Istället skulle jag kalla det en arkitektur med en tydlig separering av ansvar (*separation of concerns*). Controllers tar emot förfrågningar (requests) och skickar svar. Vad som händer med förfrågan hanteras av logiken i olika objekt, det finns dataklasser som motsvarar en post i en tabell och det finns dataklasser som motsvarar innehåll i formulär på webbsidorna. Dessutom finns det databasklasser som

hanterar läsning och skrivning till databasen. Själva sidan renderas av en mall. Figur 3 visar hur den nya sajten hanterar förfrågningar.



Figur 3. Ny arkitektur

3.3 Single point of entry

Det finns många goda anledningar till att se till att webbplatser har en enda startpunkt (*Why Should MVC for Websites Require a Single Point of Entry?*, u.å.).

- Det förenklar hanteringen av inkluderade filer. Man har en enda fil där man ser till att inkludera alla filer som behövs oavsett vilket script som anropas.
- Det förenklar också kontrollen av behörigheter eftersom de kan skapas i början av den enda anslutningspunkten.
- Man kan få "snyggare" url:er som är lättare att komma ihåg och som är mera sökmotorvänliga
- All kod som alla sidor behöver anropa skapas och anropas på ett enda ställe.

Alla dessa anledningar kokas ner till några tydliga fördelar, det blir mindre kod som finns på flera ställen vilket i sin tur minskar risken för att man glömmer att ändra på något ställe. Det

gör också att det blir enklare att testa funktionaliteten eftersom den bara finns på ett enda ställe och alltså inte behöver testas i olika versioner. Skulle jag vilja lägga till någon central funktion, t.ex. loggning av anrop kan jag göra det på ett enda ställe och veta att det kommer att göras oavsett vilken sida som efterfrågas.

3.4 IoC och dependency injection

*Inversion of Contro*l är en designprincip som säger att objekt ska vara beroende av abstraktioner snarare än av verkliga objekt (Hasan, 2021). Dependency injection är ett designmönster som implementerar IoC, en klass är beroende av ett visst interface snarare än ett visst objekt. En klass har ett antal interface i sin konstruktor och vilket objekt som i verkligheten skickas in till det nya objektet kan styras på olika sätt i olika implementationer.

I min implementation har jag ett Dependency-objekt som har till uppgift att skapa alla objekt som behövs för instansieringen av ett objekt av önskad typ. Dependency-klassen är ett via Composer importerat bibliotek som heter Auryn\Injector. Fördelen med att använda IoC är att det blir mycket lättare att utföra tester av klasser eftersom man kan skapa stubs som implementerar gränssnittet och i testklasserna använda mina stubs istället för att använda de "riktiga" implementationerna av gränssnittet.

3.5 Singleton och globala objekt

Designmönstret singleton är ett sätt att använda globala objekt, och globala objekt/variabler ska man undvika att använda när det är möjligt. En del objekt vill man dock av "kostnadsskäl" endast ha ett av, och i mitt fall vill jag bara ha ett objekt som kopplar till databasen för att kunna utnyttja transaktioner i databasen. Auryn\Injector-biblioteket hanterar även denna typ av objekt genom att skapa ett delat objekt (*instance sharing*) (Lowrey, u.å.). Det betyder att Dependency-klassen skapar ett databaskopplingsobjekt och att samma objekt används i alla klassers konstruktorer som är beroende av kopplingsobjektet. Databaskopplingen är det enda delade objektet i min implementering.

3.6 Factory

En factory är en klass som har till uppgift att skapa instanser av en annan klass. I de flesta fallen i min implementering skapas factory-objektet som en beroende till controller-klassen och factory-objektet skapar ett annat objekt (oftast ett form-objekt) genom att använda en del av controller-objektets fält.

3.7 Chain of Responsibility

I designmönstret Chain of Responsibility låter jag olika nivåer av validering utföras i en kedja, om någon validering fallerar bryts kedjan och valideringen returnerar ett falskt resultat och ett felmeddelande når användaren. Går alla valideringar igenom returneras ett sant resultat och hanteringen av indata kan fortsätta.

För framtida moduler planerar jag att använda en modifierad version av Chain of Command där enstaka valideringar kan misslyckas men att alla valideringar ändå görs så att användaren får reda på alla fel som fanns i indata från formuläret.

3.8 Composite

Designmönstret Composite använder jag för att skapa objekt som har underobjekt i flera nivåer i samband med inläsning av resultatfiler. Genom att låta subklasserna implementera ett interface kan jag låta varje klass läsa den “egna” XML-taggen och sen skicka inläsningen av underliggande tagg till en egen klass. Omvandlingen till en objekthierarki sköts genom att varje klass låter sin/sina underklass(er) skapa ett objekt och returnera det/dem till ovanför liggande objekt för vidare hantering.

3.9 Decorator

Designmönstret Decorator använder jag i samband med mallarna. Composer-biblioteket Twig har en render-metod men genom att skapa ett interface med en render-metod kan jag på ett mycket enkelt sätt testa att det är rätt mall som renderas och att rätt data skickas till mallen.

3.10 Ajax

Ajax är en teknik att skicka data mellan klienten och servern via Javascript och (ursprungligen) XML som informationsbärare. Eftersom det är ganska krångligt att skapa XML-dokument i PHP och ganska svårt att läsa dem med Javascript i jämförelse med hanteringen av JSON använder jag mig av JSON som informationsbärare mellan server och klient. Anropen görs med traditionella POST/GET anrop och svaren är i JSON-format. Till en början använde jag mig av PUT och DELETE anrop också och lät klasserna ta hand om det via PHP's php://stdin. Problemet är att man inte så enkelt kan testa indata via stdin som det är att testa via request-objektet, så jag övergav PUT/DELETE anropen och gjorde om dem till vanliga POST-anrop.

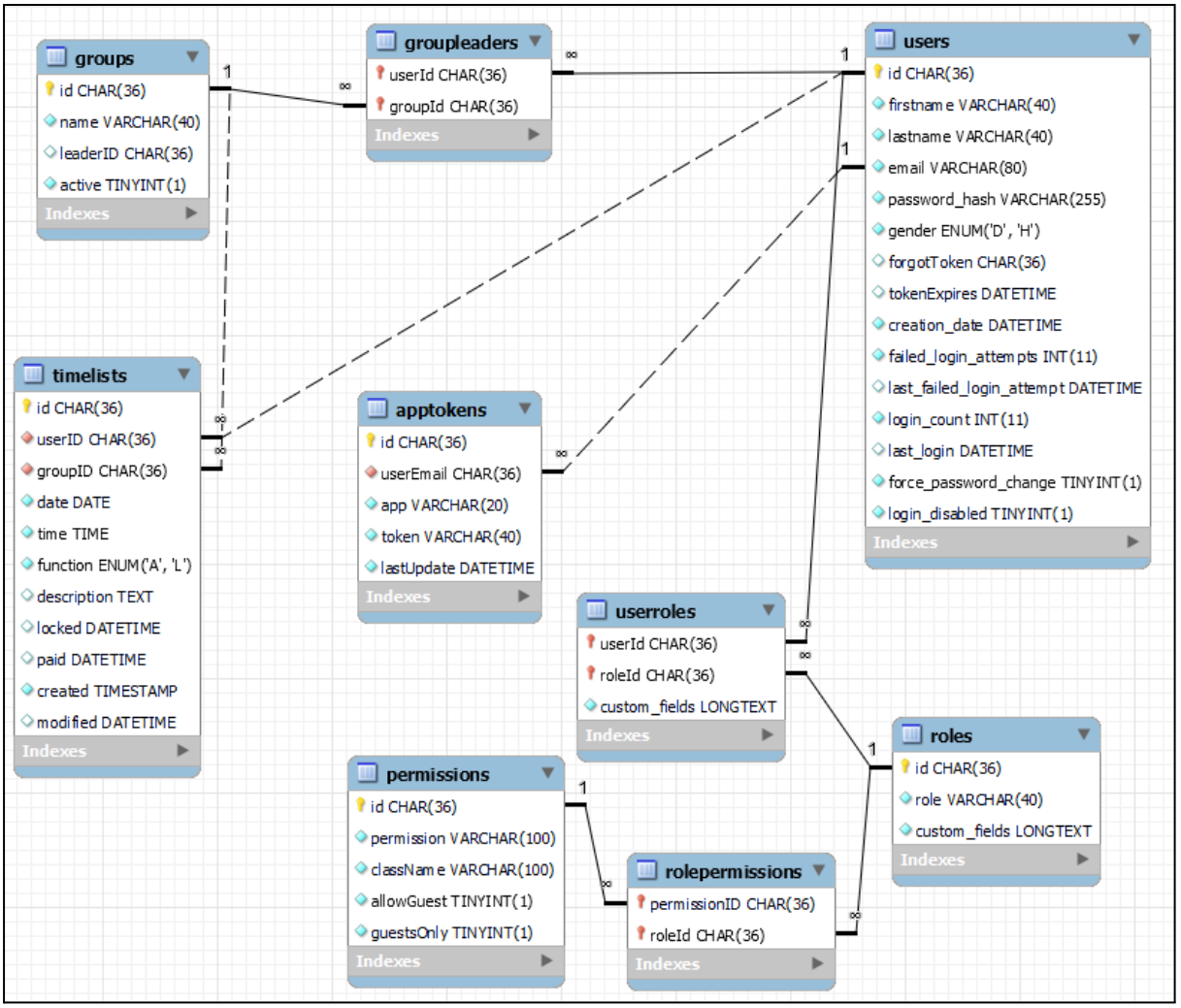
Eftersom sajten anpassas efter moderna webbläsare har jag genomgående använt mig av fetch-api:et som är implementerat i alla webbläsare sen 2015/2016 beroende på vilken läsare man använder (*Fetch API*, u.å.).

4 DATABASSTRUKTUR

Den nya sajtens nuvarande databasstruktur är inte funktionell utan ett resultat av två parallella utvecklingsspår. Det betyder att det finns en hel del dubblerad data och en hel del omstruktureringar som behöver göras innan nästa release av applikationen kan släppas.
Det första spåret (det ursprungliga) hanterar användare, roller, och en enkel tidsrapporteringsapp. Där finns också en början på en hantering av rättigheter via användarens olika roller. Figur 4 visar databasens tabeller och relationer i detta det första spåret.

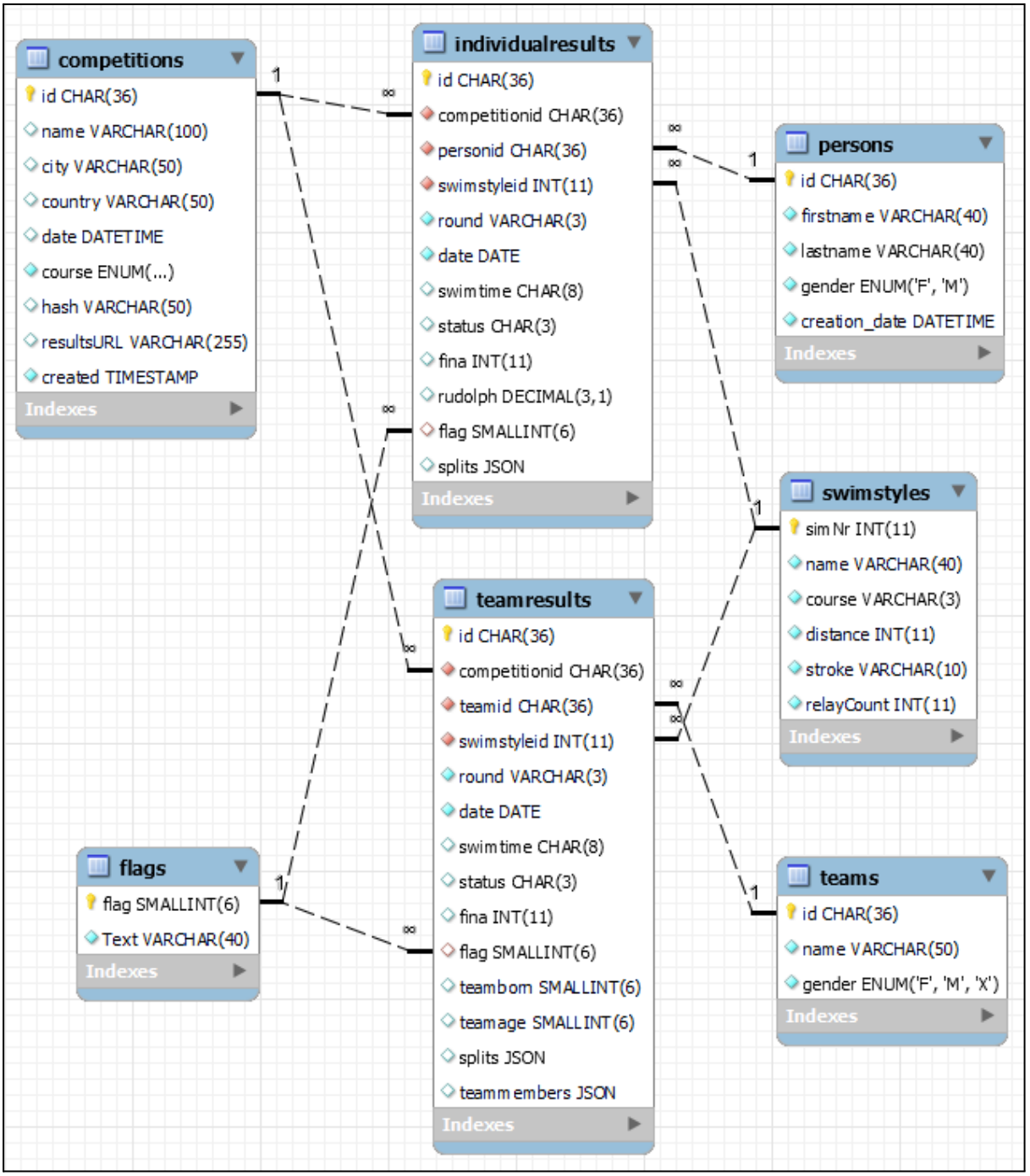
I den gamla databasen användes heltal (autoincrement) som primärnycklar, medan alla id-fält [1] i den nya databasen är Uuid-fält. Det gör att man inte kan kopiera över data från tidigare tabeller på ett enkelt sätt utan en konverteringsfunktion behöver byggas upp. Den mesta av databasen är inte konverterad utan data sätts in efter hand som den behövs.

[1] Utom SimNr som är ett heltal och nyckelfältet i tabellen Swimstyles



Figur 4. Nuvarande databasstruktur - användare och roller (och tidsredovisning)

Det andra spåret i utvecklingen är det som hanterar resultaten. Det spåret påbörjades i och med ett projekt för att skapa en ny inläsning av resultat-filer. Eftersom utvecklingen av inläsningen skedde separat från övrig utveckling optimerades databasen utan större hänsyn till befintliga eller tidigare tabeller utan så optimalt som möjligt för just den isolerade uppgiften. Den utvecklingen resulterade i databasstrukturen som visas i figur 5.



Figur 5. Nuvarande databasstruktur - resultathantering

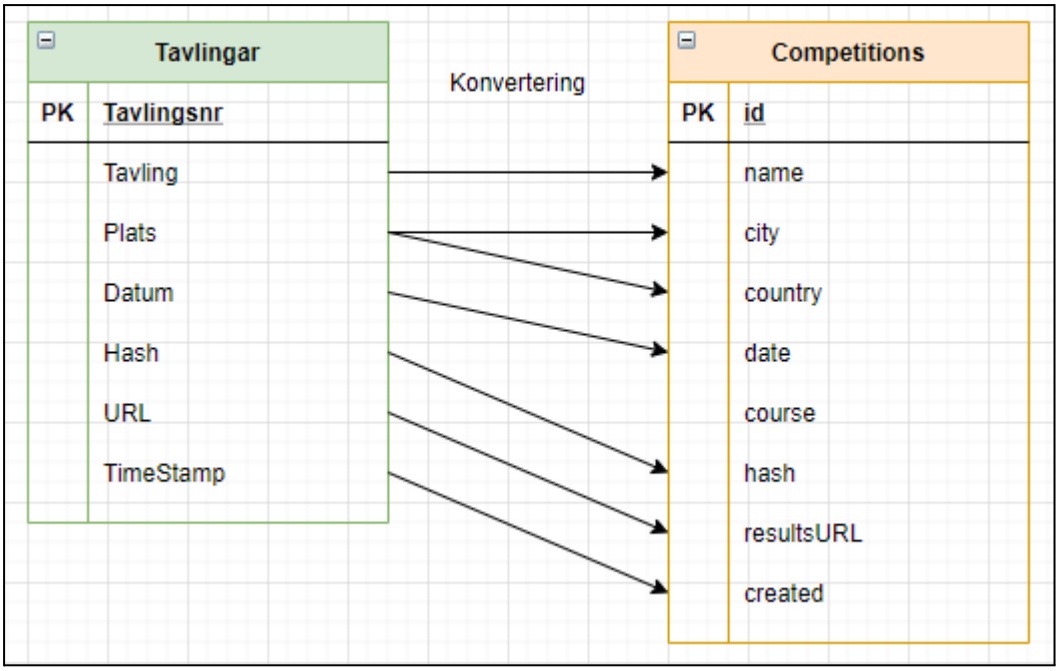
Det spåret fortsatte med import och konvertering av tidigare data till befintliga tabeller.

4.1. Resultatkonverteringen

I konverteringen av resultaten konverterades 1 300 tävlingar, 42 000 resultat och 24 000 mellantider för drygt 550 simmare.

Steg 1: Konvertera tävlingar

Första steget i konverteringen handlade om att kopiera över all information från tävlingstabellen till den nya competition-tabellen, se figur 6.



Figur 6. Konvertering av databasstruktur - tävlingsdata

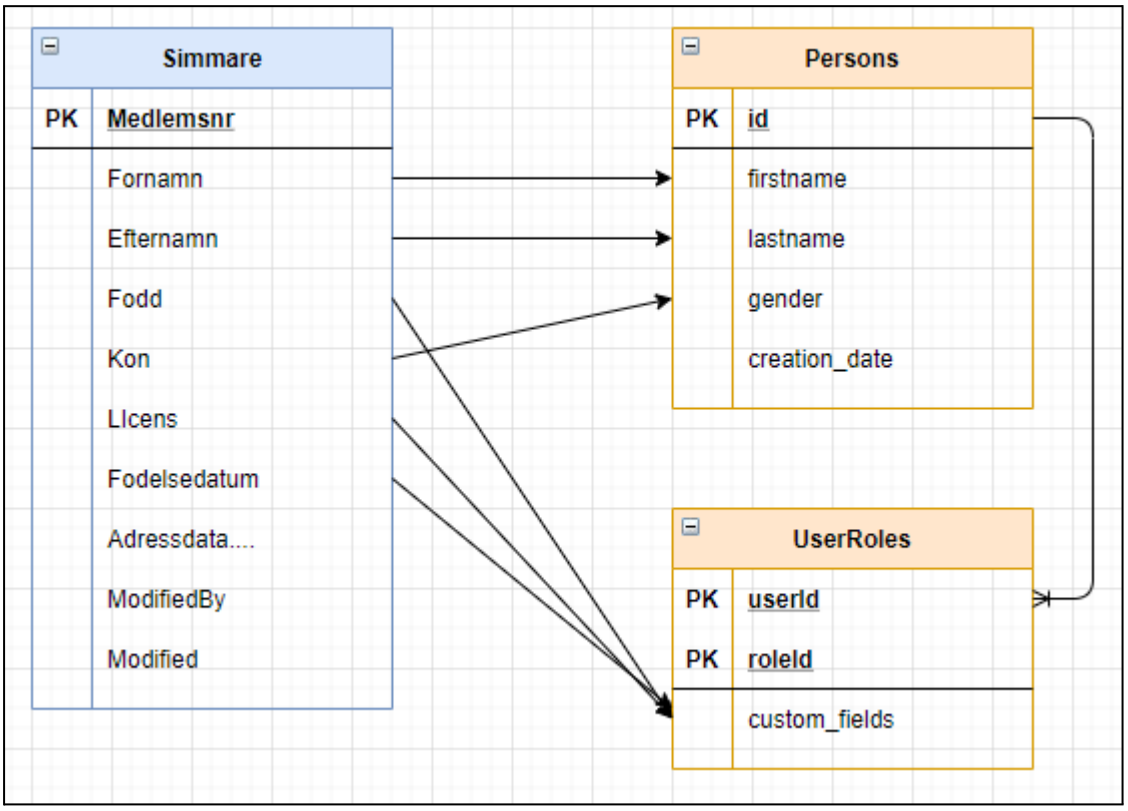
En utmaning i konverteringen är att data i den gamla tabellen var sparad med charset Latin1 och alla specialtecken (åääöé osv) var lagrade som html-entities (å ä ö é osv), i den nya databasen sparas all data med charset utf8.

En annan utmaning var att kopiera över Plats-data till city/country fälten i den nya tabellen. De flesta platserna hade bara ett ortsnamn och då kopierades det över till city-fältet. Enstaka poster hade också ett komma (,) som separerade platsen från landet. I de fallen kopierades informationen före kommat till city-fältet och resten till country-fältet..
Orsaken till att jag vill ha separation av stad och land är för att kunna tillhandahålla statistik över antal tävlingar i olika länder, framförallt Finland, Sverige och Åland.
Före publiceringen av nästa version kommer jag att behöva göra en genomgång av alla platser och sätta rätt land mer eller mindre manuellt.

I samband med inläsningen skapades en array där Tavlingsnr från den gamla tabellen var nyckel och id från den nya var värde. Den arrayen behövs för att kunna hålla reda på sambandet mellan de gamla och nya primärnycklarna för tävlingstabellerna.

Steg 2: Konvertera simmare

Nästa steg i konverteringen var att konvertera simmarnas personinformation till en ny person-tabell. Med tanke på att alla personer, oavsett roller, ska ligga i samma tabell (för att undvika redundans) har persontabellen ganska få fält, de simmarspecifika fälten för licens, födelseår och födelsedatum slogs ihop till ett JSON-fält i tabellen för användarroller (user roles), figur 7 visar vilka fält i den tidigare (blå) tabellen som mappades mot vilka fält i de nya (rosa) tabellerna.



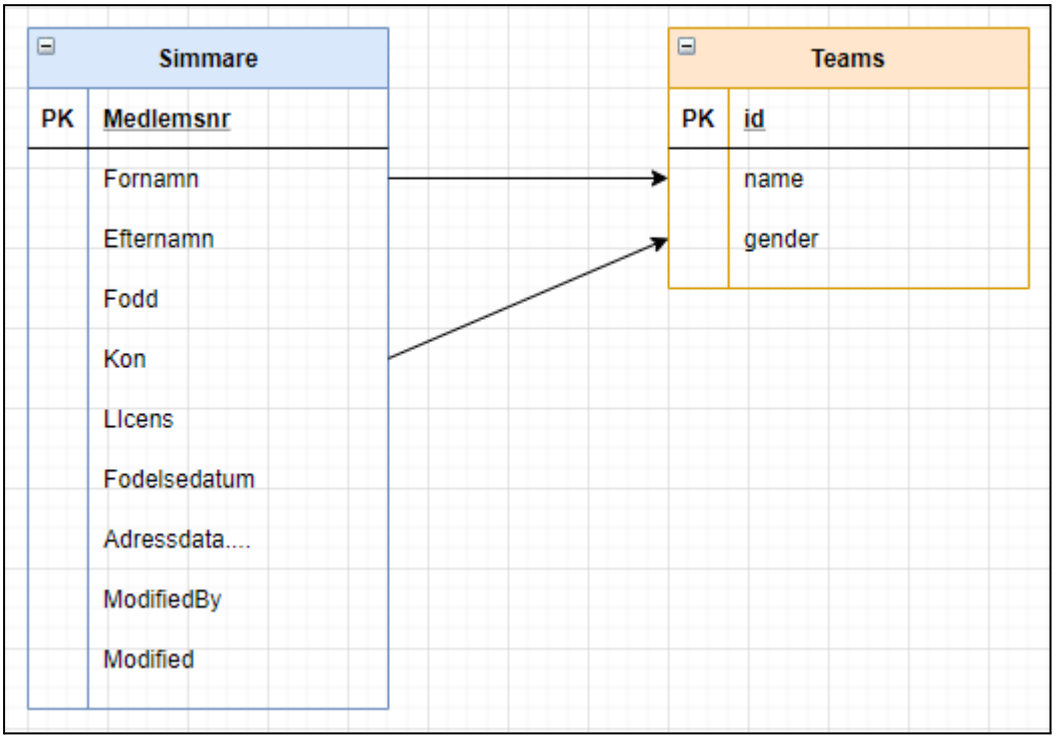
Figur 7. Konvertering av databasstruktur - simmardata

Även funktionen för att konvertera simmarna resulterade i en array med den gamla Medlemsnr-informationen som nyckel och det nya id som värde för att användas vid konverteringen av resultatdatan.

Informationen om kön är olika i den gamla och nya tabellen. I den gamla lagrades kön som char(2) (‘H’ för herr och ‘D’ för dam) men i den nya tabellen är det en enum (‘F’ för dam och ‘M’ för herr). Orsaken till att jag har olika enumerationer för kön i gamla och nya databasen är att jag när jag skapade den nya databasen helt enkelt inte tänkte på att jag hade de svenska förkortningarna i den gamla.

Även tabellen simmare i gamla databasen användes charset Latin1 så både förnamn och efternamnen fick göras om från html-entiteter till vanliga tecken eftersom persons-tabellen lagras i utf8.

Den tidigare simmartabellen innehöll även stafettlag, men tanken var att stafettlag skulle lagras i en separat tabell (teams) så konverteringen av simmartabellen fick göras i två steg, först ett för personer och sen ett andra för stafettlag, figur 8 visar mappningen mellan den tidigare tabellen och den nya.



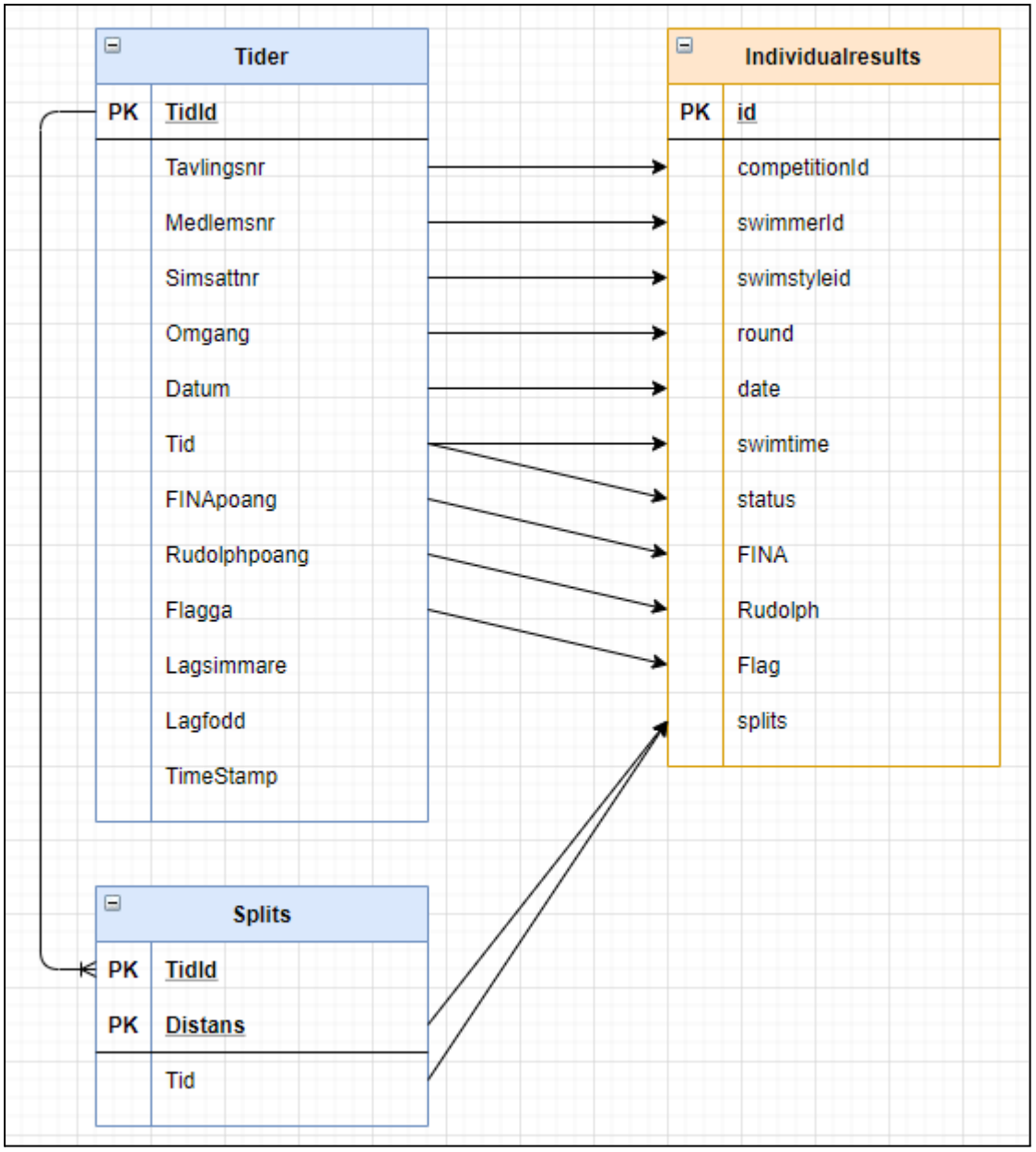
Figur 8. Konvertering av databasstruktur - stafettlag

Informationen om kön var i den gamla tabellen char(2) (‘LH’ för herr, ‘LD’ för dam och ‘LX’ för mixed) men i nya tabellen är det en enumeration (‘F’ för dam, ‘M’ för herr och ‘X’ för mixed). Stafettlagen saknade efternamn så förnamnet var det enda som kopierades över till den nya tabellen.

Steg 3: Konvertera resultat och mellantider

Resultaten konverterades i två omgångar eftersom de skulle in i två olika tabeller, en för individuella resultat och en för stafettresultat. För att hålla reda på att rätt tävlingsnr blev rätt competitionid användes arrayen som skapades i samband med att tävlingsdatan konverterades, samma gäller för medlemsnr och swimmerId. Simsattsnr och swimstyleid är samma värde från en tabell med alla simsätt.

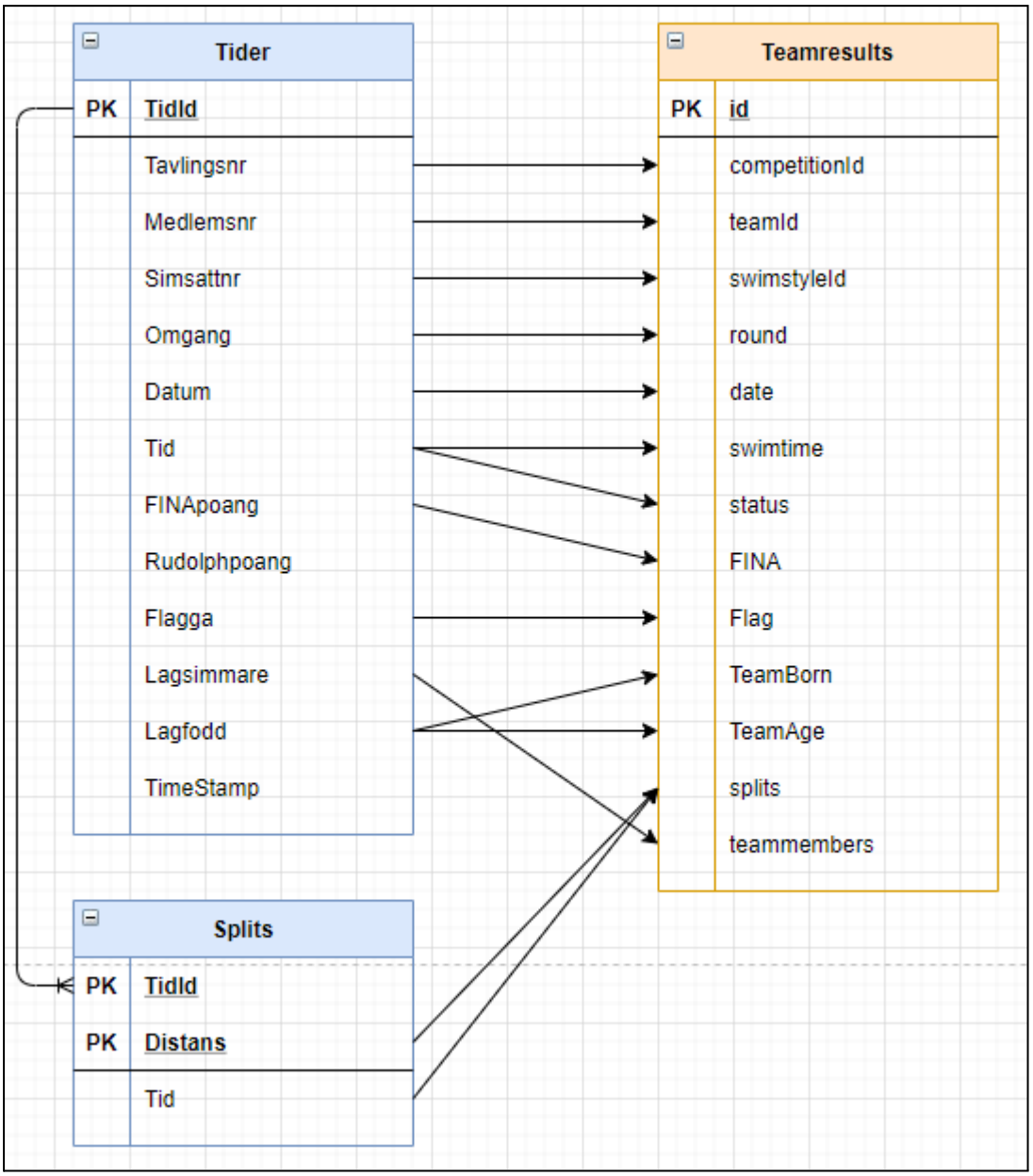
I den tidigare databasens tid-tabell lagrades DNF (Did Not Finish), diskvalifikation mm i tid-fältet, men i den nya tabellen är det två skilda fält, ett för tid (null om det inte är ett giltigt resultat) och status (oftast null om det är ett giltigt resultat). Mellantider flyttades från en egen tabell till ett JSON-fält i resultattabellen. Figur 9 visar hur mappningen gick till mellan de tidigare tabellerna och den nya tabellen för individuella resultat.



Figur9. Konvertering av databasstruktur - Individuella resultat

Konverteringen av stafettresultaten gick till på samma sätt, med det tillägget att fältet Lagsimmare i den tidigare tabellen nu delades upp i ett antal olika fält som kombinerades ihop till ett JSON-fält i tabellen Teamresults.

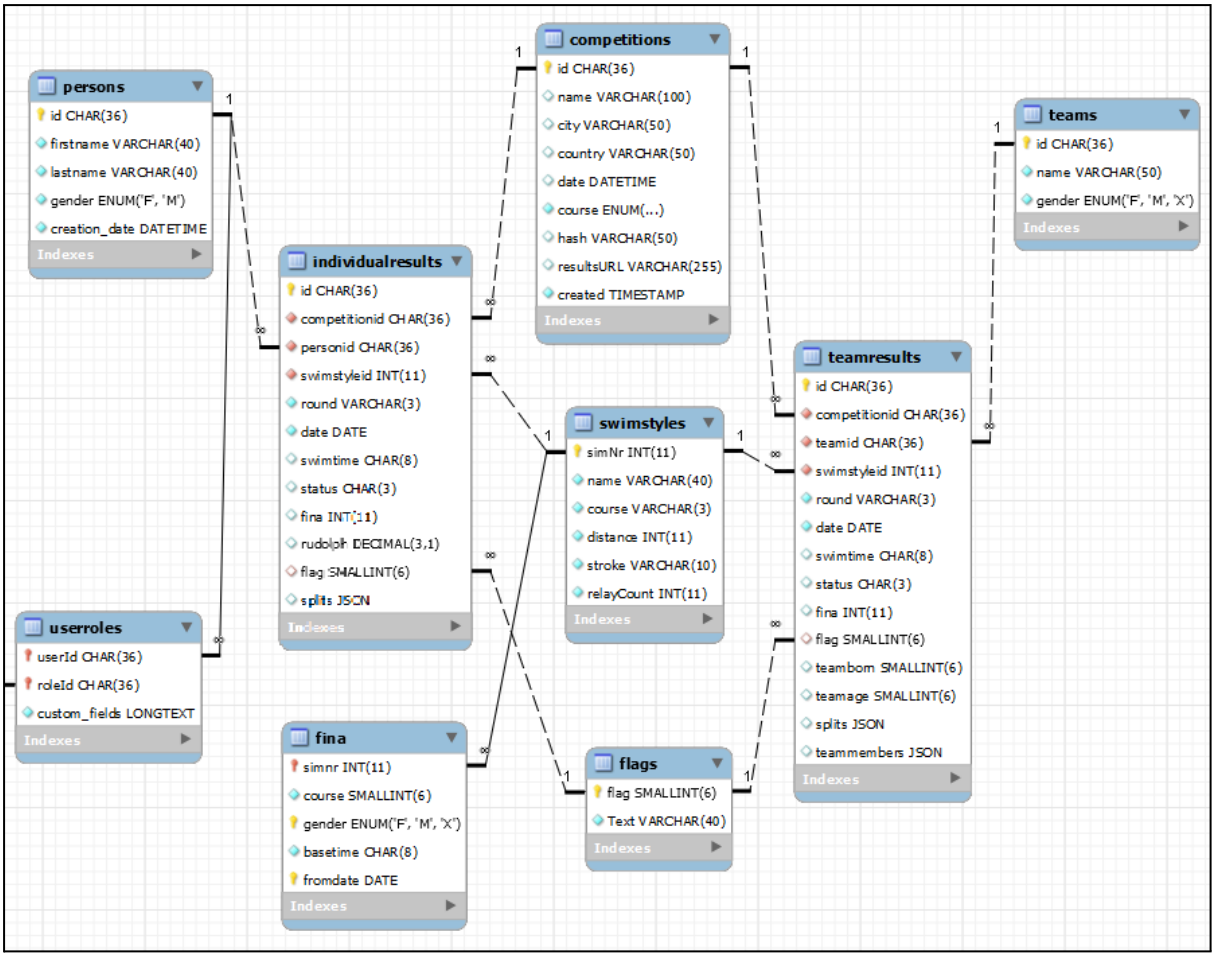
I den tidigare tabellen användes fältet Lagfodd för att lagra antingen äldsta lagmedlemmens födelseår (för normala resultat) *eller* sammanlagda åldern på alla simmare i laget (för Masters-resultat). Den informationen delades upp i antingen äldsta lagmedlemmens födelseår (fältet *TeamBorn*) och sammanlagda åldern på simmarna i laget (fältet *TeamAge*). Figur 10 visar mappningen av stafettresultat i gamla (blå) och nya (rosa) databasen.



Figur 10. Konvertering av databasstruktur - Stafettesultat

4.1.2 Ny resultat-struktur

Den nya databasens struktur gällande resultathanteringen ser alltså ut enligt figur 11.



Figur 11. Ny databasstruktur för resultatdelen

5 BACKENDDESIGN OCH KOMPONENTBIBLIOTEK

5.1 Övergripande design

Via en .htaccess-fil i webbroten dirigeras alla anrop till index.php utom de som pekar på en existerande fil som t.ex. en Javascript-fil eller bild.

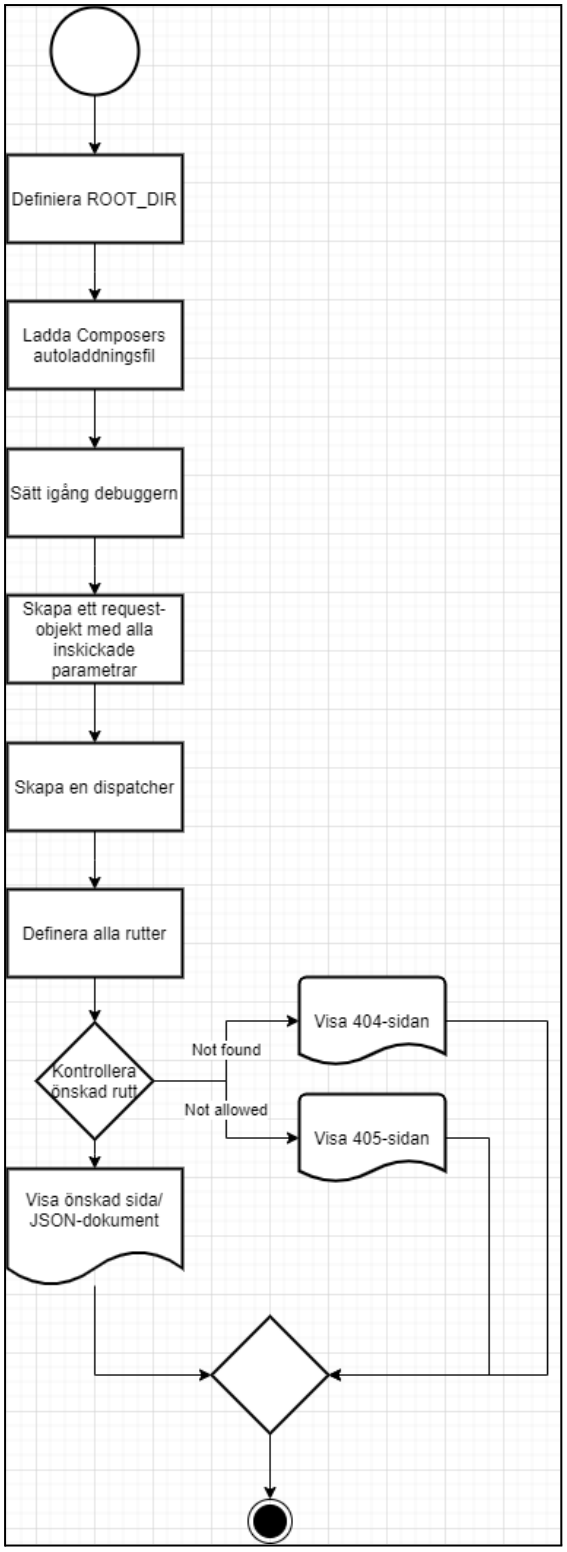
```php
<?php

declare(strict_types=1);

// Se till att alla fel rapporteras
ini_set('display_errors', '1');
error_reporting(E_ALL);
require __DIR__ . '/../src/Bootstrap.php';
```

Figur 12 Fullständig kod för index.php

Koden i filen är som synes i figur 12 minimal. Både rad 7 och 8 kommer (tillsammans med kommentaren) att raderas i den publicerade versionen. Detta är den enda PHP-filen som är nåbar via internet. Alla andra filer ligger utanför webbroten och är alltså inte nåbara annat än via denna enda fil. Man kan alltså inte ange rätt sökväg till en fil och då exekvera den som man kunde i den tidigare versionen, de url:er som man anger har ingenting med sökvägen till någon fysisk fil att göra utan är bara rutter som sedan översätts till en metod i en controller-klass. Bootstrap-filen är aningen mer omfattande, figur 13 visar ett flödesschema över filen.



Figur 13. Flödesschema för bootstrap.php

Redan i bootstrap-filen råkar vi ut för de fyra första komponentbiblioteken: symfony/http-foundation (för hantering av bl.a. request och response), tracy (för att underlätta felsökning), nikic/fast-route (för rutt-hanteringen) och rdlowrey/auryn (för hantering av beroenden mellan klasser).

Alla rutter i rutt-filen pekar på en metod i någon av alla controller-klasserna, och den metoden gör sedan anrop till olika objekt som skapats i samband med att controller-klassen skapades genom dependency injection från ett objekt som kan liknas vid en service locator som skapar de nödvändiga objekten som konstruktorn i controller-klassen behöver.

5.2 Controller-metoderna

Ganska många av controller-metoderna har samma uppbyggnad, först kontrolleras behörigheten. Än så länge genom att det är hårdkodat att man antingen är en authenticatedUser eller en Admin beroende på behörighetsnivån för metoden som anropas. Därefter läses Request-objektet av och tilldelas lokala variabler, behöver inparametrarna valideras skapas ett form-objekt med hjälp av en factory-metod i ett formfactory-objekt. Form-objektet innehåller valideringar och en command-metod som skapar ett objekt där alla databasens fält finns som egenskaper. Detta objekt skicka via ett handler-objekt för skrivning till databasen. Samma objekt används också vid radering av posten i databasen. Därefter skickas resultatet av databashanteringen tillbaka till användaren, antingen via ett html-dokument, eller som ett JSON-dokument.

Controller-metoderna, liksom de flesta andra metoder i andra klasser är relativt korta, sällan mer än 20 rader kod (plus kommentarer och blankrader för läsbarhetens skull). Jag ser också till att hoppa ur alla metoder så snart som möjligt för att slippa onödiga kontroller och else-block (Louys, 2018, s. 25f). Figur 13 visar ett exempel på en funktion i en controller-klass.

```php
    function addGroup(Request $request): JsonResponse {
        // Bara admin
        if (!is_a($this->rbacUser, Admin::class)) {
            $me = new stdClass();
            $me->error[] = "Unauthorized";
            return new JsonResponse($me, 401);
        }

        $form = $this->groupFormFactory->createFromRequest($request);
        if ($form->hasValidationErrors()) {
            $error = new stdClass;
            $error->error = $form->getValidationErrors();
            $json = json_encode($error, JSON_PARTIAL_OUTPUT_ON_ERROR + JSON_PRETTY_PRINT);
            return new JsonResponse($json, 400, [], true);
        }
        $groupID = $this->groupHandler->add($form->toCommand());

        $result = new stdClass;
        $result->message[] = "Spara lyckades";
        $result->id = $groupID->toString();
        $json = json_encode($result, JSON_PRETTY_PRINT + JSON_PARTIAL_OUTPUT_ON_ERROR);
        return new JsonResponse($json, 200, [], true);
    }
```

Figur 13. Typisk Controller-metod - Skapa ny grupp

5.3 Typade argument och returer

PHP är ett otypat språk, dvs variabler har ingen definierad typ och kan ändra typ under sin livstid, men i samband med PHP 7.0 implementerades möjligheterna att ange vilka datatyper funktioner förväntar sig som parametrar och ger som returvärden (*PHP 7 ChangeLog*, u.å.). Det utnyttjar jag så långt det är möjligt, och i kombination med att först i alla filer ange `declare (strict_types=1);` gör det att om jag skulle anropa en funktion med fel datatyp får jag ett kompileringsfel och koden kan alltså inte exekveras förrän typfelet åtgärdats.

Tyvärr kan man inte ange en array av en viss typ (t.ex. en array med heltal) som returtyp utan man kan bara ange att det är en array. Men min editor utnyttjar att om man gör PHPDoc-kommentarer med den "rätta" datatypen för både parametrar och returtyper ändå kan få hjälp av Intellisense-funktionerna att veta vilka metoder och fält som finns i datatyperna.

5.4 Uppdelning av kommandon och frågor

En funktion ska antingen förändra ett objekts tillstånd, eller returnera ett värde som efterfrågas, aldrig både och (Louys, 2018, s. 28f). Det är en viktig uppdelning som jag håller

mig till i mesta möjliga mån i alla mina klassers funktioner. Funktioner som inte returnerar något värde har returtypen `void` för att indikera att de inte ger något returvärde.

5.5 Composer

Composer är ett verktyg för att hantera beroenden i PHP, du kan deklarera vilka komponenter ditt projekt behöver och Composer kommer att se till att de är installerade tillsammans med alla enskilda komponenters beroenden (*Introduction - Composer*, u.å.).

Composer hanterar komponentbibliotek på projekt-basis, olika projekt kan ha samma komponent i olika versioner utan att det blir någon konflikt mellan dem. Källkoden till komponenterna sparas i undermappar oftast kallad *vendor*-mappen som ligger i projektets rot-katalog.

Packagist.org är Composers basförvar, alltså den plats där Composer i huvudsak söker efter komponenter och beroenden. Alla komponenter som finns listade på Packagist kan man automatiskt hämta via Composer.

När man har hämtat en komponent via Composer skapas en autoload.php-fil i *vendor*-mappen där sökvägen till alla enskilda klasser i alla komponenter finns så att de klasserna laddas automatiskt när ditt projekt körs.

5.6 Komponentbibliotek

I mitt projekt har jag hittills använt 10 komponentbibliotek (listade nedan). När jag söker efter komponentbibliotek gör jag det oftast på Packagist (alltså via Composer) och jag söker i första hand efter komponenter som är relativt fristående från andra paket, jag vill ogärna ladda ner massor med extra filer för att få till en enkel funktionalitet.

5.6.1 tracy/tracy

Tracy (https://tracy.nette.org/) är ett verktyg för att underlätta debuggning av PHP-kod, det installeras enklast via Composer. Tracy visualiserar och spårar fel, både syntax-fel och rena skrivfel på ett lättförståeligt sätt. Man får se var felet upptäcktes och hela anropsstacken tillsammans med alla variabler i respektive metod så att det är enkelt att spåra felets ursprung

och rätta till det. Det är alltså ytterst sällan som man behöver lägga in debug-utskrifter av variabler eller stega igenom koden via något xdebug-verktyg.

5.6.2 rdlowrey/auryn

Auryn är en rekursiv dependecy injector för objektorienterade PHP-applikationer. Auryn instansierar klassberoenden som finns som parametrar i en klass constructor-metod. Auryn använder Reflection för att göra detta rekursivt så att alla beroenden blir skapade.
Via auryn kan man också definiera vilka klasser som ska användas för att implementera de interface som man har deklarerat i constructorerna. Normalt skapar auryn ett objekt per klass, men man kan också se till att det bara skapas ett delat, globalt tillgängligt objekt för till exempel databaskopplingar eller konfigurationsfiler.(Boggiano, u.å.-c)

5.6.3 nikic/fast-route

Fast-route är en komponent för att förenkla rutthanteringen och den har inga beroenden till andra komponenter.(Boggiano, u.å.-b) Hanteringen av rutter är enkel och (enligt utvecklaren) snabb.(*Fast request routing using regular expressions*, u.å.)
Min nuvarande implementering av komponenten bygger på en importerad array som finns i en routes-fil. Figur 14 visar en del av rutthanteringsfilen.

```
4
5   return [
6       [
7           'GET',
8           '/',
9           'asfStatistik\FrontPage\FrontPageController#showStatic'
10      ],
11      ['GET',
12          '/admin/staticText',
13          'asfStatistik\FrontPage\FrontPageController#show'
14      ],
15      ['GET',
16          '/admin/getAllTexts',
17          'asfStatistik\FrontPage\FrontPageController#getAllTexts'
18      ],
```

Figur 14. routes.php

Varje element består av en metod (GET/POST), en sökväg och en metod i en klass. Klassnamnet och metoden separeras med ett #-tecken. Komponenten klarar även av PUT/PATCH/DELETE och HEAD anrop, men eftersom dessa inte stöds av PHP:s request-metoder så använder jag bara GET/POST-anrop.

5.6.4 symfony/http-foundation

Symfony är en uppsättning PHP-komponenter, ett ramverk för att skapa webbapplikationer (Symfony, u.å.-b). Komponenterna som bygger upp ramverket kan emellertid användas fristående och det finns i dagsläget 130 komponenter.

http-foundation-komponenten hanterar alla http-request/respons-anrop i olika klasser vilket gör att man aldrig behöver hantera de superglobala variablerna ($_GET/$_POST/$_FILES/ $_SESSION osv) utan använder objekt istället. Det gör också att det är enkelt att skapa testobjekt eftersom man kan lägga till variabler till objekten. I min applikation använder jag request-objektet för att ta emot data från webbsidan och olika respons-klasser för att skicka svar till anropande sida. Om det är ett AJAX-anrop skickar jag en JSONResponse-svar och om det är ett "normalt" webbanrop är det ett vanligt Respons-objekt som returneras. När det är fråga om "normala" POST-anrop skickar jag en RedirectResponse till en annan sida för att undvika ompostning av formulär. Figur 15 visar en funktion med ett sådant RedirectResponse-svar.

```
public function register(Request $request): RedirectResponse {
    $response = new RedirectResponse('/register');
    $form = $this->registerUserFormFactory->createFromRequest($request);
    if ($form->hasValidationErrors()) {
        foreach ($form->getValidationErrors() as $errorMessage) {
            $this->session->getFlashBag()->add('errors', $errorMessage);
        }
        return $response;
    }

    $this->registerUserHandler->handle($form->toCommand());

    $this->session->getFlashBag()->add(
            'success',
            'Kontot är skapat, du kan nu logga in.'
    );
    return $response;
}
```

Figur 15. Exempel på RedirectResponse för att undvika ompostning av formulär

5.6.5 twig/twig

Twig är också en symfony-komponent för att skapa HTML-mallar som man kan fylla med inskickad data. Twig har ett enklare sätt att hantera loopar och andra förgreningar i koden och

att skriva ut variabelvärden. Twig har stöd för arv vilket gör att man kan bygga kraftfulla mallar och inkludera olika delar av en webbsida i olika filer (Symfony, u.å.-a).

I min applikation är mallhanteringen abstraherad så att man kan använda olika komponenter för att rendera sidorna, jag har t.ex. en testimplementering som inte returnerar någon HTML-kod utan en array bestående av anropad mall och inskickade parametrar. På så vis kan jag kontrollera att det som skickas in är rätt och om det inte ser rätt ut på sidan beror det på hanteringen av data i twig-mallen och inget annat.

Jag har också använt designmönstret Decorator för min "normala" formulär-rendering och lagt till ett antal funktioner till twig-klassen som gör att jag bland annat kan rendera en meny utan att behöva skicka menyn som en inparameter till själva mallen.

5.6.7 doctrine/dbal

Doctrine är ett annat stort ramverk med komponenter som framförallt fokuserar på lagring av data och mappning av dataobjekt. Därifrån har jag tagit komponenten som abstraherar databasanropen.("Introduction", 2014) Det betyder att jag behöver inte skriva SQL-satser utan bygger upp dem med olika metodanrop. Figur 16 visar en funktion som bygger upp en SQL-sats via doctrine/dbal.

```php
public function getIndividualSwimstyles(string $bassang): array {
    $qb = $this->connection->createQueryBuilder();
    $qb->addSelect('simNr')
            ->addSelect('name')
            ->addSelect('course')
            ->addSelect('distance')
            ->addSelect('stroke')
            ->addSelect('relayCount');
    $qb->from('swimstyles');
    $qb->where("course = {$qb->createNamedParameter($bassang)}")
            ->andwhere("relayCount=1");

    $stmt = $qb->execute();
    $rows = $stmt->fetchAll();
    $swimstyles = [];
    foreach ($rows as $row) {
        $swimstyles[] = Swimstyle::createFromRow($row);
    }

    return $swimstyles;
}
```

Figur 16. Exempel på hämtning av data via doctrine/dbal

Jag undviker också att konkatenera ihop villkor från inparametrar utan använder hela tiden namngivna parametrar som synes i figuren ovan. Connection-objektet får klassen i constructor-metoden och det är ett delat objekt som via injectorklassen och en factorymetod i ConnectionFactory-klassen skapas och delas mellan alla objekt med databaskoppling. Figur 17 visar hur man skapar delade objekt i injectorklassen via metoden *share*. Figur 18 visar hur factory-metoden i connection-klassen läser config-filen för att få sajtspecifik data till databaskopplingen.

```php
$injector->delegate(Connection::class, function () use ($injector): Connection {
    $factory = $injector->make(ConnectionFactory::class);
    return $factory->create();
});
$injector->share(Connection::class);
```

Figur 17. injector-klassen skapar ett delat objekt via en factory-metod

```
public function create(): Connection {
    return DriverManager::getConnection(
                    ['dbname' => $this->config->get("Database.db"),
                        'user' => $this->config->get("Database.user"),
                        'password' => $this->config->get("Database.password"),
                        'host' => $this->config->get("Database.host"),
                        'driver' => 'pdo_mysql'],
                    new Configuration()
    );
}
```

Figur 18. Factory-metoden i ConnectionFactory-klassen, informationen för att koppla till databasen ligger i en konfigurationsfil som läses via config-objektet.

5.6.8 robmorgan/phinx

Versionshantering av databasen är ett problem som jag inte stött på tidigare. Men i samband med detta mycket större projekt än mina tidigare behöver jag kunna hantera olika versioner av databasen och att bara dumpa ut ett SQL-script varje gång jag gör en ny release skulle inte fungera. Därför har jag använt mig av en komponent som sköter uppgradering av databasen via script som jag skapar. Det betyder att jag kan ge ett enkelt kommando "*phinx migrate*" för att uppdatera databasen till den senaste versionen både i min lokala utvecklingsmiljö och i produktionsmiljön (*Phinx Documentation - 0.12*, u.å.).

Komponenten som tillåter mig göra detta heter phinx och är en del av CakePHP-ramverket även om komponenten i sig är utvecklad av en oberoende utvecklare. Via phinx kan jag uppgradera men också nedgradera databasen eller lägga in poster i t.ex. uppslagstabeller. Kontrollen av vilka script som körts lagras i databasen i en särskild tabell som heter phinx.

5.6.9 ramsey/uuid

Den sista komponenten som är hämtad från Professional PHP är en komponent för att skapa uuid:n, d.v.s. Universal Unique Identifier (Wikipedia contributors, 2022f). Via komponenten kan man skapa ett antal olika versioner av uuid:n, den version som jag använder hela tiden är uuid4 (slumpmässigt uuid). Jag använder uuid för att lagra unika nycklar i databasen, orsaken till att jag använder uuid istället för autogenererade heltals databas-id är att jag då på förhand vet vilket id som den nya posten får och kan på så vis skapa objekt utan att egentligen spara dem i databasen. Det underlättar testning och gör att lagringen blir mer flexibel. Jag skulle kunna lagra poster i en JSON-fil istället för att använda databasen utan att det skulle påverka någonting.

5.6.10 hassankhan/config

PHP kan läsa ini-filer, men jag ville ha en mera flexibel konfigurationsfil och tyckte att yaml var ett lämpligt alternativ. För att underlätta läsning och parsning av yaml-filerna behövde jag hitta en komponent som gör det utan att ha en massa andra beroenden och jag fastnade till slut för hassankhans komponent (Boggiano, u.å.-a).

5.6.11 swiftmailer/swiftmailer

Den mailhanteraren som jag valde i samband med att jag behövde kunna skicka ut påminnelser till användare som glömt sina lösenord var swiftmailer, en fristående del av Symfony-sviten. Men sen slutet av 2021 underhålls den inte längre, jag kan emellertid inte byta till den föreslagna Symfony Mailer eftersom den kräver PHP8.1 och min utveckling sker i PHP 7.4.

Även mail-skickningen är abstraherad så att Swiftmailern implementerar en Maildispatcher och vilken implementation av Maildispatchern som används sätts i injector-filen. Det betyder att jag kan testa att skicka mail utan att det går ut något mail på riktigt. Det gör också att jag när jag väl uppdaterar till PHP8.x behöver skapa en ny Maildispatcher implementation som implementerar Symfony mailern istället för Swiftmailern. En snabb switch i dependency-filen gör att jag bytt mailhanterare.

6 DESIGN AV ANVÄNDARGRÄNSSNITTET OCH API-KOMMUNIKATION

6.1 HTML

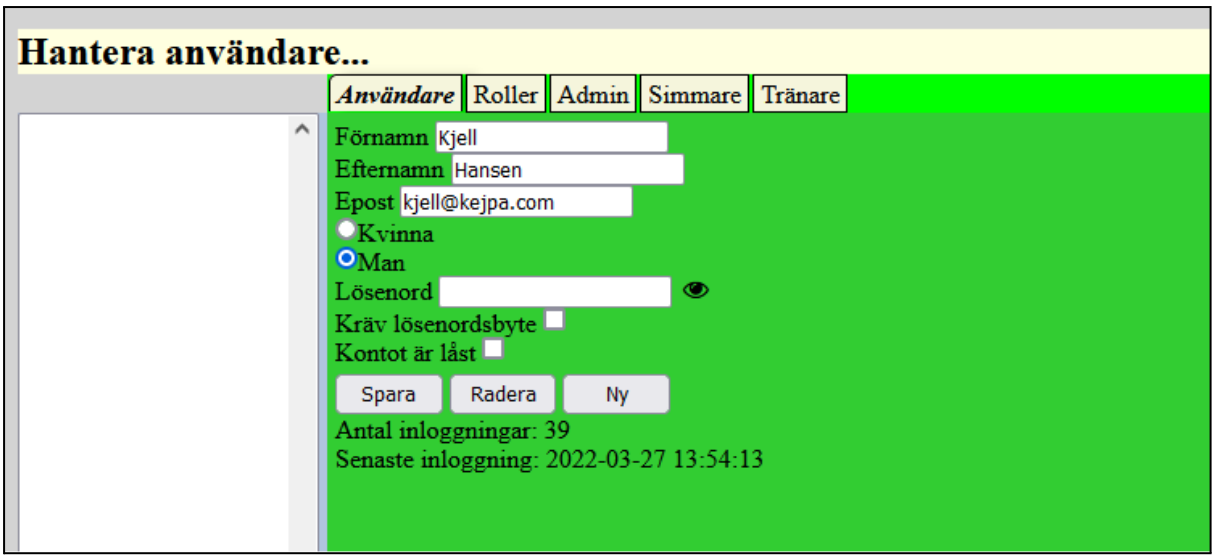
Användargränssnittet är HTML genererad via Twig-filer. Det finns en twig-fil för grundstommen, alltså sidans layout där head-informationen finns inlagd, vilka css-filer som laddas in. Layout-filen innehåller de delar av sidan som är gemensamma för alla sidor, alltså en header-sektion, en vänsterpanel, en footer och en main-sektion dit de enskilda sidornas innehåll renderas. Innehållet lägger jag i en egen tagg som jag kallar *contents*.

Mellan body-sluttaggen och html-sluttaggen har jag lagt in dels de sidogemensamma Javascript-länkarna och en separat funktion för att lägga in de sidospecifika Javascript-filerna. Orsaken till att jag lagt Javascript-länkarna sist i dokumentet är så att de laddas först efter att alla sidelement finns. Risken är annars att en tagg som ännu inte skapats anropas vilket gör att skripten slutar fungera.Figur 19 visar hur webbsajtens grundlayout ser ut, färgerna visar de olika elementen i grid-layouten. Menyerna i headern och i vänsterpanelen är funktioner inbäddade i själva layout-filen och är beroende av inloggningsstatus och användarrättigheter.

Figur 19. Webbsajten med de olika sidoelementen i olika färger.

6.1.1 Användargränssnitt med flikar

Det finns ganska många sidor som har flikar och för dem har jag skapat egna taggar, jag använder taggen *tabs* för att ha en container för de enskilda tabbarna och taggen *tab* för att definiera själva tabben, på *tab*-taggen har jag ett attribut som säger vilken *section*-tagg som ska visas när tabben är aktiv. I nedanstående exempel finns de två första tabbarna i källkoden, medan de tre sista är dynamiskt skapade baserat på de roller som användaren har. Figur 20 visar hur den grå ytan från figur 19 fylls med ett gränssnitt med flikar. Figur 21 visar motsvarande HTML-kod.



Figur 20. Användargränssnitt med flikar

```
<tabs id="tabs">
    <tab id="Anvandare" showTab="tabAnvandare" class="tabActive">Användare</tab>
    <tab id="Roller" showTab="tabRoller">Roller</tab>
</tabs>
<section id="tabAnvandare" class="tabContainer">
    <...25 lines />
    <p>Antal inloggningar: <span id="loginCount"></span></p>
    <p>Senaste inloggning: <span id="lastLogin"></span></p>
</section>
<section id="tabRoller" class="tabContainer">
    <...7 lines />
</section>
```

Figur 21. HTML-kod för flikarna och deras innehåll

6.2 CSS

Jag har använt vanlig css för att designa sidan och inget färdigt css-ramverk typ bootstrap eller tailwind. Till en början var sajten baserad på positionerade block och inline-block element, men i ett senare skede bytte jag ut den layouten till en gridbaserad layout där endast main-elementet och vänsterpanelen scrollar vid behov. En grid går också enkelt att anpassa till aktuell skärmstorlek vilket gör att sajten numera anpassar sig bättre till små skärmar än före bytet till grid-layouten. I det tabbade användargränssnittet är *main*-delen av rutnätet ett *grid*-element.

6.2.1 Sass

Sass är en förkortning för Syntactically Awesome Style Sheets och det är en preprocessor för CSS-filer. I SASS kan man använda variabler för att definiera värden, det finns också ett antal funktioner man kan använda och stöd för arv (Wikipedia contributors, 2022g).

Jag använder SASS för att skapa mina CSS-filer. Det är ett smidigt sätt att få kontroll på vilka element man verkligen vill påverka eftersom det går att nästla css-selektorerna. Jag har ännu inte använt mig av variabler, utan jag har hårdkodat alla attributvärden.

6.3 Javascript

En hel del funktionalitet i front-end bygger på AJAX-anrop, alltså asynkrona anrop till servern för att uppdatera innehållet på sidorna. Anropen sker med vanlig http-kommunikation via GET- och POST-anrop, GET-anropen för att hämta data, POST-anropen är datamodifierande anrop för att lägga till, uppdatera eller radera information ur databasen. Majoriteten av anropen använder sig av FormData-objektet för att skicka datan och i de flesta fallen finns ett formulär på webbsidan där all data som ska skickas ligger. Det gäller framför allt POST-anropen. GET anropen hämtar sin information från andra kontroller som inte nödvändigtvis ligger i något formulär.

6.3.1 Jquery

I början av utvecklingen hittade jag inte riktigt de funktionerna jag behövde i “ren” Javascript utan fick använda mig av en del Jquery-funktioner. På senare tid har jag alltmer hittat rätt i Javascript-syntaxen och sköter det mesta av hanteringen utan Jquery. På sikt kommer jag säkert därför att omarbeta alla Jquery-anrop och därmed också ta bort alla länkar till det biblioteket.

6.3.2 content-tools

För att sköta inline-editering av hela eller delar av sidor har jag laddat ner och installerat content-tools. Content-tools är ett intressant tillägg som möjliggör riktig WYSYWYG-editering av webbsidor. Jag såg det för drygt 1 år sen och blev genast intresserad, det gick lätt att implementera och att justera efter mina behov åtminstone inledningsvis. Ganska snart stötte jag på problem med att ladda upp bilder av annan storlek

än de jag ville ha på sidan och möjligheter att bestämma hur stora bilderna skulle vara. Det är också lite osnyggt med editeringsknappar mm. För att få det riktigt anpassat efter mina behov behöver jag säkert lägga ner mycket tid på att leta i dokumentationen (som inte är helt lättläst, eller komplett) och prova mig fram.

6.3.3 FetchAPI och asynkrona funktioner

Fetch API är en utveckling av den tidigare HttpRequest-funktionaliteten och erbjuder ett interface för att anropa en server endpoint. FetchAPI:et har funnits tillgängligt i webbläsare sen våren 2015 och är implementerad i alla större webbläsare sen 2016 (*Fetch API*, u.å.). FetchAPI:et bygger på asynkron kommunikation och använder Promises för att vänta på att anropets svar. Därmed kan man använda .then()-kommandon som körs först när Promise-objektet slutförts.

Innan jag började använda .then()-funktionaliteten använde jag mig av asynkrona funktioner, men det blir många funktioner och ganska oöverskådligt och inte helt självklart när olika saker händer, därför har jag allt mer gått över till Promise-baserad hantering.

Eftersom det är möjligt att mina anrop till servern fallerar pga felaktiga data behöver jag kunna visa användaren att anropen misslyckades (och lyckades) på ett sätt som inte är irriterande och kräver någon aktiv handling för användaren. Jag har löst det genom en sk toast som visas om ett anrop gick rätt eller fel och vad som hände eller vad för fel som uppstod. Jag har valt att skicka http error statusvärden (4xx) för felen och 200 om det gått bra, men det var problem att ta tillvara felmeddelandena och generera en “feltoast”. Det löste jag genom att läsa respons-Promisens JSON-data asynkront. Figur 22 visar hur jag kombinerar promise och asynkron exekvering av en metod.

```
fetch('/getCompetition?id=' + id)
        .then(async function (response) {
            if (response.status === 200) {
                let competition = await response.json();
                showInfo(competition);
                getResults(id);
            } else {
                throw Error(await response.json());
            }

        })
        .catch(err => {
            let msg = "";
            if (Array.isArray(err)) {
                for (let e of err) {
                    msg = msg + e.error + "<br>";
                }
            } else {
                msg = err.toString();
            }
            toast('error', msg);
        });
```

Figur 22. Exempel på asynkron läsning av respons-objektets json-data i en Promise

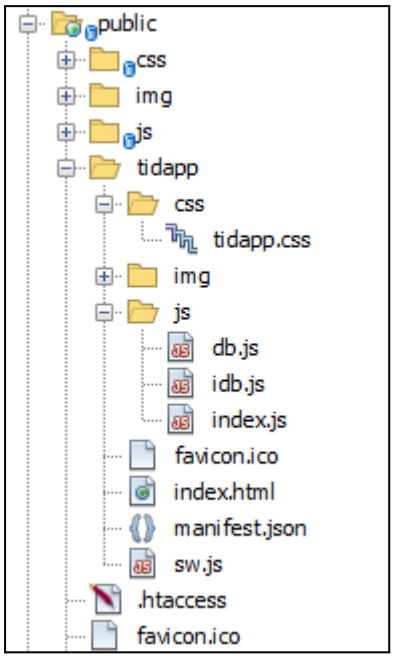
Kombinationen med asynkron inre funktion för att vänta på att läsningen av Promise-objektets JSON-data är väldigt enkel och smidig så den använder jag numera i alla fetch-anrop.

6.3.4 JSON

JSON är ett enkelt sätt att överföra information mellan datorer, det är läsbart för vanliga människor. För datorer är det enkelt att tolka och generera eftersom det bygger på två enkla och universella datastrukturer tillsammans med vanliga datatyper (*JSON*, u.å.). Den första strukturen är ett namn/värde-par och den andra strukturen är en ordnad lista av värden. JSON började som en del av Javascript-standarden i december 1999, men blev vanligare efter mitten av 2010-talet. Jag använder JSON i all min AJAX-kommunikation från webbservern till klienten eftersom det är så enkelt att skapa och enkelt att tolka.

6.3.5 Tidsrapporteringsapp

I sajten finns också en tidsrapportering för manuell rapportering av arbetade timmar för tränarna inbyggd. Figur 24 visar hur det gränssnittet ser ut. Det är egentligen bara en egen mapp med egna Javascript/css och bilder som ligger i webbroten, se figur 23.



Figur 23. Tidsredovisningsappen

Index.html-filen är uppbyggd av olika tabbar ungefär som en del av det tabbade interfacet i webbapplikationen. Informationen som man lägger in skickas dels till webbservern och lagras där, men det finns även off-line lagring som använder sig av webbläsarens inbyggda indexedDB. Så långt möjlig har jag försökt att göra tidsrapporteringen till en PWA och det fungerar tillfredsställande för de flesta telefoner som jag har fått rapporter om från tränarna. Rapporter från användare meddelar att det inte är helt enkelt att behålla sin inloggning över tid och det är också någonting som är ganska svårt att testa eftersom det kräver att man inte använder appen under en lång tid och sen vid första anropet till den så får man ibland felaktiga (gamla) inloggningstokens.

Figur 24. Tidsredovisningsappen

7 TESTNING

Jag kom igång ganska sent i utvecklingsskedet med enhetstestning (eng. *unit testing*), men när jag väl började få ordning på det såg jag till att alla funktioner som returnerar någonting testas. Ett normalt arbetssätt innebär att jag börjar på en funktion på en webbsida, och jag arbetar mig sakta framåt så att jag får den att fungera i alla delar. Sedan tar testskapandet vid och jag använder mig av *glass box*-testning. Jag vet ju hur implementeringen av funktionaliteten ser ut så målet med testerna är att först få dem att fungera och sen skriva flera scenarier som först inte fungerar för att implementeringen av funktionaliteten inte har tagit hänsyn till dem. Så justerar jag implementeringen så att de nya testerna går igenom och sen fortsätter jag att skriva tester som inte fungerar och som jag sen får att fungera genom att ändra i implementationen.

Jag är noga med att se till att koden i alla förgreningar omfattas av testerna, så min täckningsgrad (*code line coverage*) ligger säkert över 90%. Jag kan inte ange en exakt siffra eftersom jag inte fått *code coverage* rapporteringen att fungera med PHPUnit som jag använder för tester av serverscripten.

Tyvärr har jag inte hittat något bra verktyg för att testa mina databas-klasser alltså de klasserna som interagerar direkt med databasen. En tanke är att ha en testdatabas som jag ser till innehåller en känd uppsättning data och att jag sen manipulerar den datan och ser om resultatet är det förväntade. Problemet är att databasen ännu inte är stabil och att det kommer till nya tabeller, fält mm liksom att informationen inte heller är helt stabil. Dessutom känns det som ett ganska stort steg att skapa och se till att data är konsistent mellan tester.

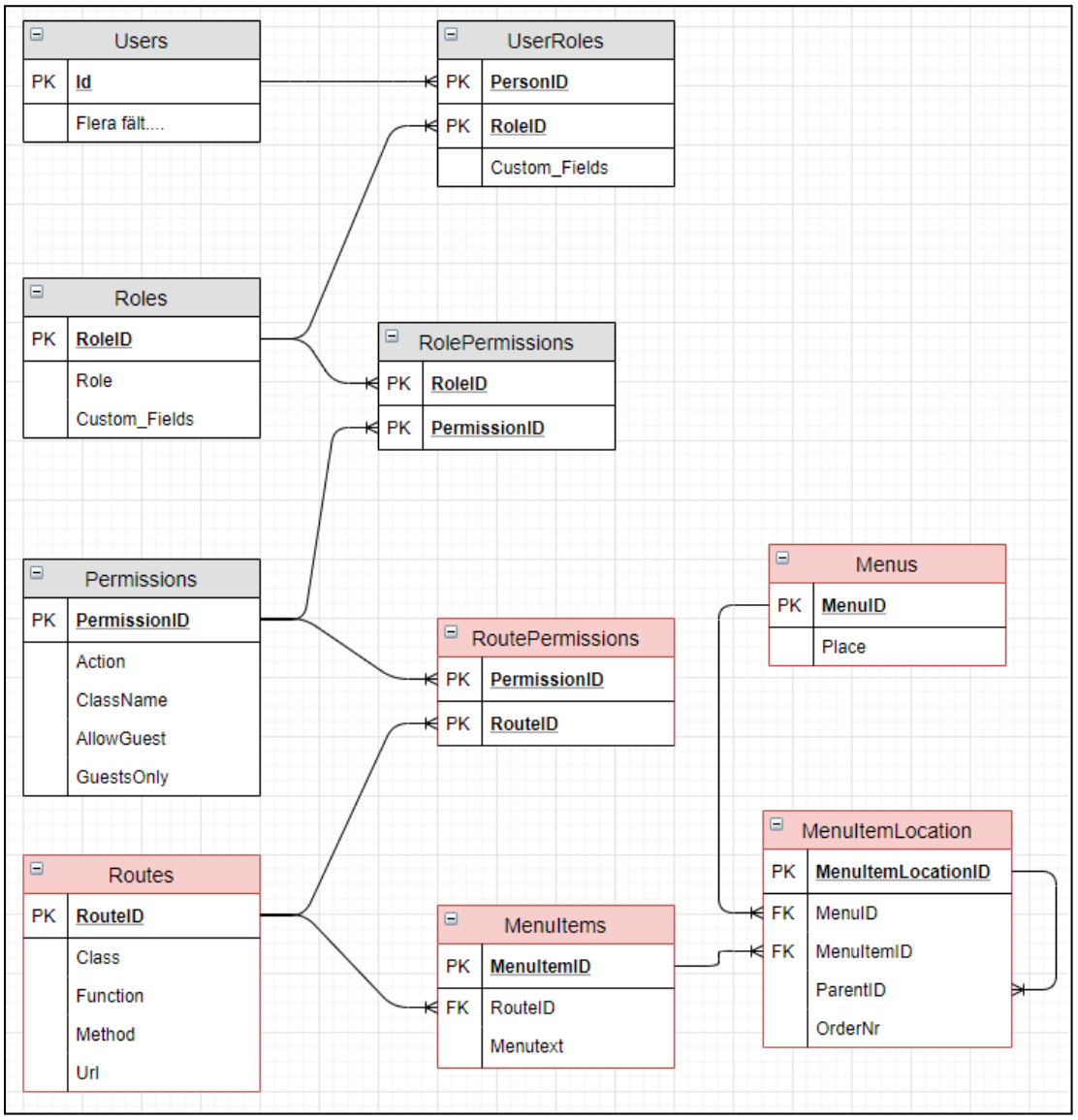
Jag har heller inget verktyg för testning av Javascript eller användargränssnittet utan där behöver jag lita på att jag som utvecklare provar att alla knappar osv.

8 FRAMTIDA UTVECKLING

8.1 Databasen

8.1.1 Menyer och rutter

När omstruktureringen av databasen är klar så att alla personer endast finns i persons-tabellen och att users är en tabell med data för användarna (med ett personid-fält med 1-till-1 koppling till persons-tabellen) kommer jag att lägga in alla rutter och menyalternativ i databasen. Just nu finns de i olika filer i filträdet.



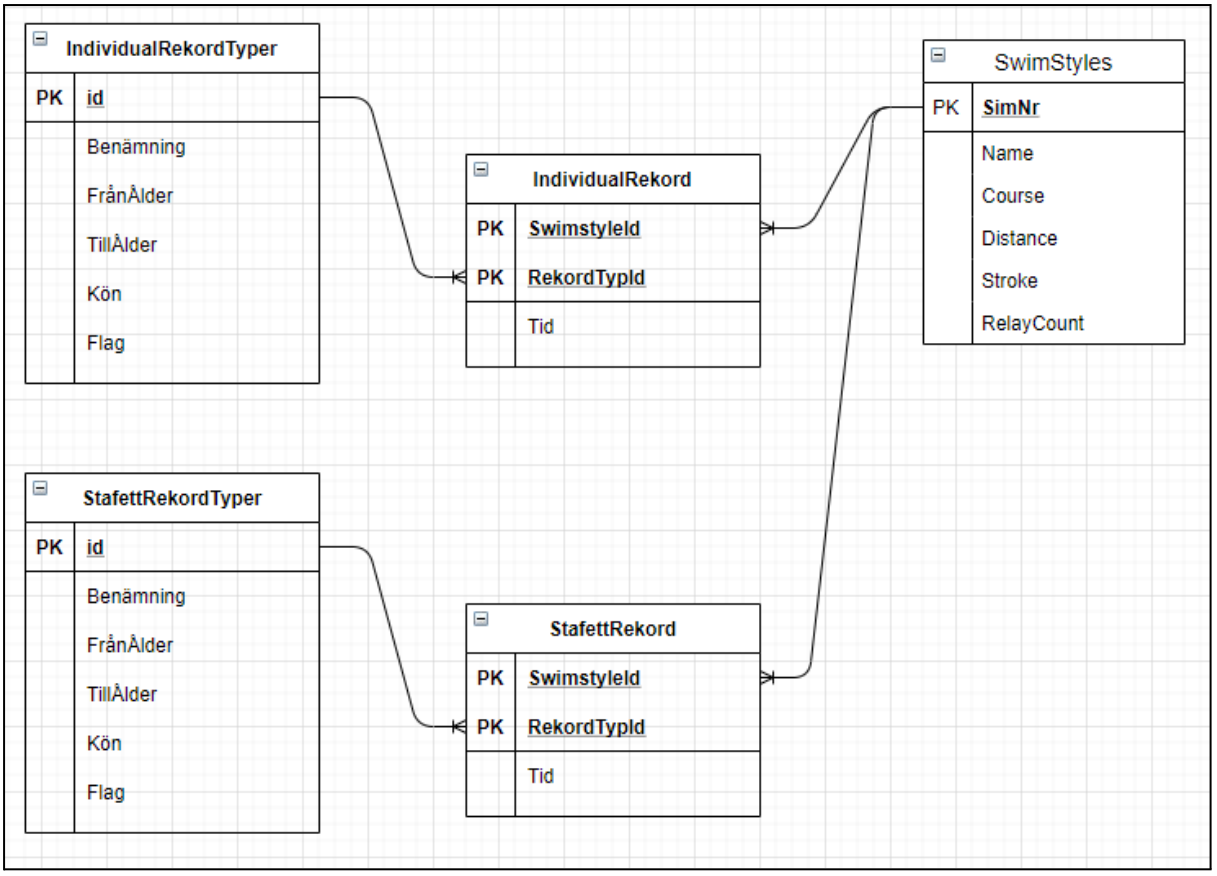
Figur 25. Planerad databasstruktur menyer och rutter, styrt av användarens roller och dess rättigheter, nya tabeller med rosa färg, befintliga med gråblå.

I figur 25 är de planerade tabellerna markerade med rosa färg. Både rutter och menyalternativ ska alltså lagras i databasen och endast de användare med rättigheter till respektive rutt kommer att kunna ha tillgång till rutten. På samma sätt kommer menyalternativ att visas endast för de användare som har tillgång till den rutten som menyalternativet pekar på.

8.1.2 Rekord

All registrering av resultat ska såklart också avspegla sig i att man kan visa upp rekord. I den tidigare databasen finns en särskild tabell som innehåller alla aktuella rekordnoteringar för de olika åldersklasserna som finns. Åldersklassernas indelning, benämning osv finns också i en separat tabell. Från början var planen att mer eller mindre kopiera över tabellen från den gamla databasen till den nya. Eftersom jag delat upp individuella resultat och stafettresultat i olika tabeller är frågan om det inte är bäst att göra samma för rekorden och alltså ha en rekordkategori-tabell för individuella rekord och en rekordkategori-tabell för stafettrekord och på samma sätt ha en tabell för individuella rekord och en för stafettrekord.

I den gamla databasen sparades också en hel del redundant information i tabellen, så som *både* simmarens id-nr och namn, födelseår mm. Orsaken till att det var gjort så var att från början fanns inte alla resultat i resultat-tabellen utan vissa rekord fanns utan att tävlingen och dess resultat var registrerat i databasen. Nu när jag har alla resultat kanske det är bättre att bara lagra id-fälten för själva noteringen. Figur 26 visar det nuvarande utkastet till tabellstruktur för rekorden.

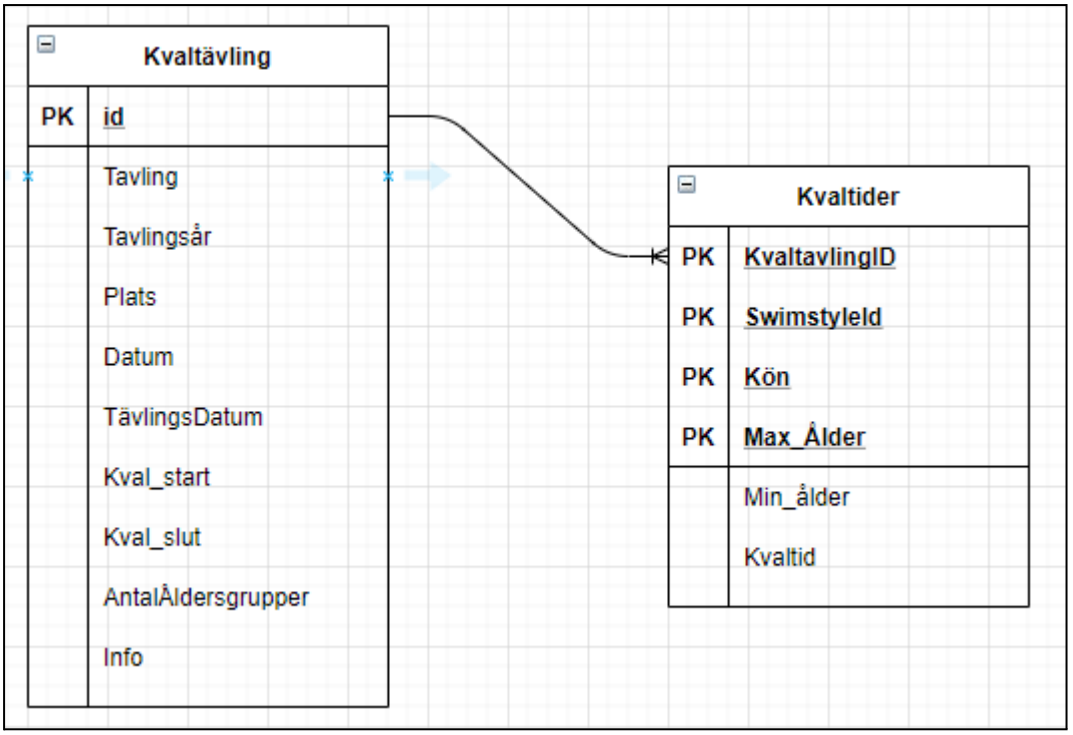


Figur 26. Utkast till databasstruktur - Rekord

8.1.3 Kvaltider

En viktig funktion är också att man kan se vilka tider som gäller för kvalifikation till olika mästerskapstävlingar. Dessa tider finns inlagda i det nationella resultatregistret och det vore bra om jag kunde få en länk till ett API där jag kan läsa dessa tider direkt till min databas istället för att jag som nu behöver mata in dem manuellt.

Den tidigare tabellstrukturen ser i konverterad form ut enligt figur 27.

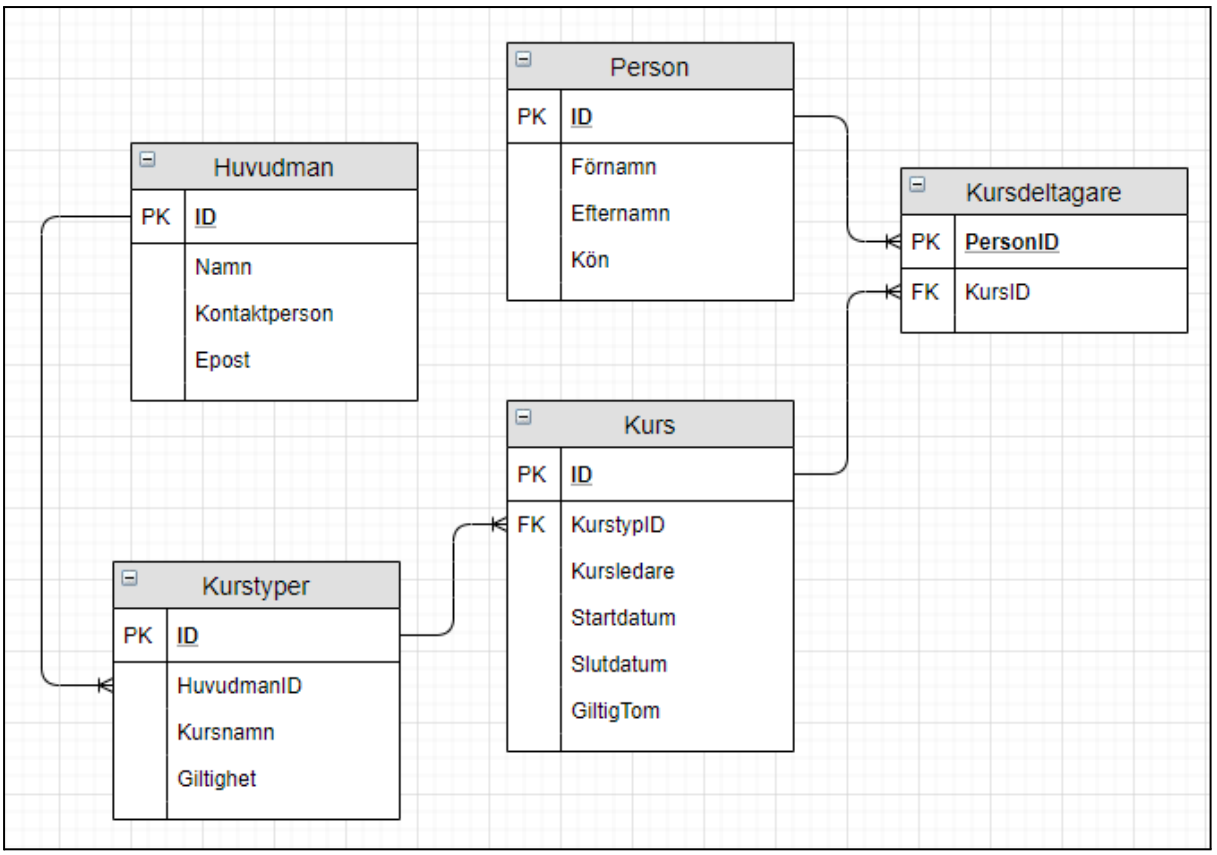


Figur 27. Tidigare (konverterad) databasstruktur - Kvaltider

Jag ser i dagsläget ingen anledning att ändra annat än kolumnnamn och datatyper, men om jag får en API-endpoint med alla kvaltider kan jag behöva fundera om en gång eller två på utseendet.

8.1.4 Kurser och deltagare

Så småningom vill jag också att man ska kunna registrera de kurser och andra utbildningar som ledarna och funktionärerna genomför. En del kurser är i Svenska Simförbundets regi, en del är i Finska Simförbundets regi och andra är i Folkhälsans och Röda Korsets regi. Några har en giltighetstid andra saknar giltighetstid. En del kurser består av flera delkurser vilket gör databasstrukturen ganska komplex. Mitt nuvarande utkast till databasstruktur ser ut enligt figur 28.



Figur 28. Planerad databasstruktur - Kurser och utbildningar

Eftersom det finns kurser som består av delkurser behöver jag troligen implementera designmönstret Composite i koden och därmed blir nog tabellen *Kurs* rekursiv med ett fält till för *ParentId*.

8.2 Backend

8.2.1 PHP8

Den senaste PHP-versionen har många intressanta funktioner som jag gärna vill använda. För att minska ner mina *POJO*-objekt har nya PHP möjligheten att i konstruktorn ta emot och definiera privata variabler i klasserna (*PHP 8.0 Released*, u.å.). Det som idag ser ut enligt figur 29 skulle kunna kortas ner till utseendet i figur 30.

```php
private $id;
private $firstname;
private $lastname;
private $gender;
private $customFields;

public function __construct(PersonId $id, string $firstname, string $lastname,
        string $gender, ?string $customFields) {

    $this->id = $id;
    $this->firstname = $firstname;
    $this->lastname = $lastname;
    $this->gender = $gender;
    if ($customFields === null)
        $customFields = "{}";
    $this->customFields = json_decode($customFields);
}
```

Figur 29. Exempel på nuvarande POJO-klass deklaration

```php
public function __construct(private PersonId $id, private string $firstname,
        private string $lastname, private string $gender,
        private ?string $customFields) {

    if ($customFields === null)
        $customFields = "{}";
    $this->customFields = json_decode($customFields);
}
```

Figur 30. Exempel på framtida POJO-klass deklaration

PHP8 ger mig också möjligheter att typdeklarera mina klass-variabler vilket gör att det blir lättare att direkt se vad de används till och vilken datatyp de är. Man kan också ha flera olika returtyper, man kan returnera till exempel bool och int genom att ange returtypen till *bool|int*. PHP8.1:s utökade möjlighet till enumerationer kommer jag troligen också att utnyttja, men just nu är jag inte helt säker på till vad, eftersom de flesta enumerationer jag använder läses från uppslagstabeller i databasen (*PHP 8.1 Released*, u.å.).

8.2.2 Designmönstret Chain of Rsponsibility för validering av indata

Valideringen av indata görs idag i Form-klasserna med en massa if-satser, figur 31 är ett exempel på en sådan funktion:

```
public function getValidationErrors(): array {
    $errors = [];
    if (!$this->storedTokenValidator->validate(
                'registration',
                new Token($this->token)
            )) {
        $errors[] = 'Ogiltigt token';
    }
    if (filter_var($this->firstname, FILTER_SANITIZE_STRING) === "") {
        $errors[] = 'Förnamn saknas';
    }
    if (filter_var($this->lastname, FILTER_SANITIZE_STRING) === "") {
        $errors[] = 'Efternamn saknas';
    }
    if ($this->emailExistsQuery->emailExists($this->email)) {
        $errors[] = 'Epostadressen är redan registrerad';
    }
    if (!filter_var($this->email, FILTER_VALIDATE_EMAIL)) {
        $errors[] = 'Ogiltig epostadress';
    }
    if (mb_strlen($this->password) < 8) {
        $errors[] = 'Lösenordet måste vara minst 8 bokstäver';
    }
    if ($this->gender !== "H" && $this->gender !== "D") {
        $errors[] = "Kön behöver anges (D) Dam eller (H) Herr";
    }
    return $errors;
}
```

Figur 31. Exempel på nuvarande valideringsfunktioner

Det vore snyggt att skapa klasser för validering av efternamn, förnamn, epost, osv. och att i form-klasserna skapa *Chain of command*-objekt som ser till att göra valideringen. Då skulle man få en enhetligare validering av indata eftersom all validering av t.ex. förnamn kommer att skötas av en *firstnameValidator*-klass som är länkad med övriga klasser som ska valideras vid varje tillfälle. Det minskar också behovet av att skriva test cases för att testa validering av indata.

8.2.3 Komponentbibliotek

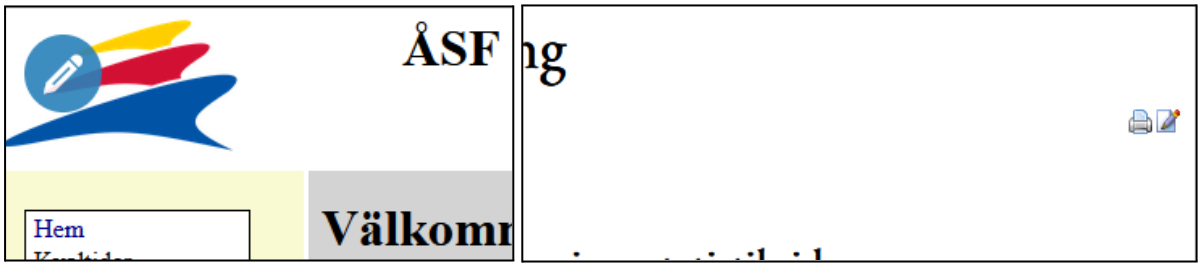
Som jag tidigare nämnt har jag i dagsläget använt mig av 10-talet färdiga komponentbibliotek, en är att jag kommer att behöva en knapp handfull till för att göra saker

som jag vet att jag vill ha med och som rimligen någon annan också har behövt, till exempel export av Excel och pdf-filer. Några bibliotek behöver bytas ut eftersom de är markerade som "övergivna" (*deprecated*) och alltså inte längre underhålls och därför bör bytas ut snarast möjligt. Bland annat vet jag att biblioteket för hantering av mail (swiftmailer/swiftmailer) är *depricated* sedan november 2021 (*Swift Mailer: A Feature-Rich PHP Mailer*, u.å.).

8.3 Frontend

8.3.1 WYSIWYG-editering

I den tidigare sajten använde jag Tiny-MCE för *WYSIWYG*-redigering, och den var delvis implementerad "på plats" alltså i den sidan som skulle editeras och delvis var det en annan sida som man redigerade och som sen texten visades på. Till en början var min plan att fortsätta med Tiny-MCE, men sen såg jag att det numera inte riktigt var samma upplägg som 2008 så då tittade jag lite på alternativen, framför allt CK-edit som verkade vara ett stabilt och bra alternativ till Tiny. Men jag gjorde aldrig något försök att implementera någondera komponent eftersom in-line redigering inte var riktigt aktuellt i dåvarande utvecklingsfas. Något senare stötte jag på content-tools och dess möjligheter till in-line redigering och gillade upplägget och möjligheterna som det gav, men jag fick aldrig riktigt till hanteringen av bilder. Både storleksförändringarna och uppladdningen var fortfarande stora frågetecken efter en kort tid med försök till implementering. Dessutom tyckte jag inte riktigt om de stora ikonerna som hamnade på editerbara sidor, jag ville helst ha något mer diskret alternativ för att editera sidan än vad som erbjöds som standard. Figur 32 visar skillnaden mellan nya och tidigare redigeringsikonens utseende.



Figur 32. Skillnaden mellan nya och gamla editerings-ikonen

Det finns helt säkert stora möjligheter att ändra ikonens utseende och placering, men det låter sig inte göras helt enkelt efter vad jag kunde se vid en ganska hastig anblick. Så hur och

vilket verktyg jag kommer att använda mig av för in-page redigeringen av sajten är ännu ganska oklart.

8.3.2 React/Angular för tidrapporteringen

Eftersom jag inte är helt nöjd med hur tidrapporterings-appen fungerar vill jag bygga om den i något bättre och mera lätthanterbart Javascript-ramverk och också bättre implementera hanteringen av inloggningar så att man bara behöver logga in första gången man "installerar" appen på telefonen och att inloggningen sen sker automatiskt efter det.

Vilket Javascript-ramverk som jag ska använda är oklart, jag har hört mycket positivt om React, men jag har också förstått att Angular är ett mycket använt ramverk, liksom Vue. Mina krav på ramverket är att det ska vara enkelt att använda och att jag ska kunna köra det på min egen server, en kort demo av React visade att när man väl kompilerat koden genereras en html-fil (plus en massa andra mappar) och att man installerar det på valfri webbserver helt enkelt genom att kopiera in filerna. React har ingen server side-kod, men jag har ett färdigt API redan som jag kan behöva justera med tanke på inloggningshanteringen så det innebär inget extraarbete om jag skulle välja det ramverket. Angular och Vue har jag inte sett lika mycket av, så det kan jag inte riktigt avgöra om det är lika smidigt. En annan sak som väger in till Reacts fördel är att jag i kursen Plattformsoberoende mobilapplikationer gjorde en hel del i React Native som är en utökning av React.

8.4 Testning

8.4.1 Enhetstestning för Javascript och databastabeller

Det vore väldigt skönt att ha någon form av enhetstestning för mina Javascript-funktioner så att jag kan känna mig säker på att de i alla (alla testade åtminstone) fall uppför sig som förväntat. I dagsläget får jag lita på att jag testar dem manuellt och att jag varje gång jag gjort någon förändring testar alla alternativ, vilket är ganska tidsdrygt om jag jämför med hur fort jag kan testa alla mina PHP-filer. Om/när jag implementerar React för frontend-utvecklingen så finns det där möjligheter till testning på ett sätt som verkar likna vanlig *unit testing*.

Något sätt att kontrollera att funktionerna i databasklasserna returnerar rätt poster vore också skönt att ha, men som jag tidigare diskuterat kanske det får vänta till databasen är mer "färdig" och strukturen är stabilare.

Jag har sett att det finns något som heter Selenium där man kan testa gränssnittet, vilket på sätt och vis är tilltalande. Troligen är det en hel del jobb med att sätta upp testerna, men den tiden får man högst troligt tillbaka när man väl testar sajten i senare skeden.

8.4.2 Heroku

En staging site vore bra att ha för att göra de sista testerna i en produktionsliknande miljö och under våren har jag hört en hel del om Heroku och dess möjligheter att fungera som staging-miljö för webbapplikationer. Tanken är då att man kan ladda upp den kommande versionen av sajten dit och sen testa alla funktioner (och be andra användare testa funktionerna) utan att det påverkar produktionsmiljön. Jag har fått förklarat att Heroku är ganska generösa med up-time (500h/månad enligt uppgift) vilket innebär att jag mer än väl kan använda plattformen för continous deployment om jag skulle vilja. Det är inte riktigt aktuellt än så länge, men när jag fått till meny/rutt-hanteringen och rekord-delen av sajten skulle jag kunna göra förändringar tillgängliga i snabbare takt än jag hittills gjort.

9 SLUTSATSER

Det finns ganska många lärdomar att dra av detta projekt. Den som är mest slående är utvecklingstiden. Jag minns att då jag gjorde den mesta utvecklingen av webbsajten i dess första utseende så tog det mellan en och tre kvällar/dagar att få till en sida, det var ganska mycket kopiera kod från tidigare och justera sql-satserna och sen justera vilka fält som skrevs ut baserat på vilken data som kom från databasen. I den här nya versionen tar det betydligt längre tid att skapa sidor, det är flera klasser som behöver skapas och det är fler kontroller som behöver göras och att få ihop en ny sida tar åtminstone en vecka. Men jag känner mig mycket tryggare med att koden är säkrare, stabilare och att risken för att det finns några alternativ som gör att koden kraschar är mindre. P. Louys talar i sin bok om “teknisk skuld” och att snabb utveckling ger en större teknisk skuld som man i något skede behöver betala tillbaka om man inte har en god struktur och ser till att ingående testa de olika komponenterna.

En annan sak som jag upplever är att det finns mycket färdig funktionalitet att tillgå. En hel del av de komponenterna jag använt mig av inte är skrivna för att kombineras ihop med varandra men att de är fristående nog för att kunna användas med andra vilket jag uppskattar mycket. Det tar dock lite tid att hitta dem, men den tiden spar man in på testning och liknande.

En tredje slutsats är att jag har mycket kvar att lära. Även om jag anser mig behärska både PHP och Javascript som språk i sig ganska väl finns det väldigt mycket om effektiv hantering av klasser och objekt i den dagliga hanteringen att önska. För att inte tala om alla onlinetjänster för kontinuerlig integration och leverans.

KÄLLFÖRTECKNING

Boggiano, J. (u.å.-a). *hassankhan/config - Packagist*. Hämtad 24 april 2022, från https://packagist.org/packages/hassankhan/config

Boggiano, J. (u.å.-b). *nikic/fast-route - Packagist*. Hämtad 24 april 2022, från https://packagist.org/packages/nikic/fast-route

Boggiano, J. (u.å.-c). *rdlowrey/auryn - Packagist*. Hämtad 24 april 2022, från https://packagist.org/packages/rdlowrey/auryn

Fast request routing using regular expressions. (u.å.). Hämtad 24 april 2022, från http://nikic.github.io/2014/02/18/Fast-request-routing-using-regular-expressions.html

Fetch API. (u.å.). Hämtad 13 april 2022, från https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

Floyd, C. (1984). A Systematic Look at Prototyping. *Approaches to Prototyping*, 1–18.

Gamma, E., Richard Helm (Computer scientist), Johnson, R. E., & Vlissides, J. (2016). *Design Patterns: Elements of Reusable Object-oriented Software*. Pearson Education.

Hasan, F. (2021, april 22). *What is inversion of control?* Educative: Interactive Courses for Software Developers; Educative. https://www.educative.io/edpresso/what-is-inversion-of-control

Introduction. (2014). I *Help, I'm Rich!* (s. 1–2). John Wiley & Sons, Ltd.

Introduction - Composer. (u.å.). Hämtad 24 april 2022, från https://getcomposer.org/doc/00-intro.md#introduction

JSON. (u.å.). Hämtad 29 april 2022, från https://www.json.org/json-en.html

Louys, P. (2018). *Professional PHP: Building Maintainable and Secure Applications*. CreateSpace Independent Publishing Platform.

Lowrey, D. (u.å.). *auryn: IoC Dependency Injector*. Github. Hämtad 05 juni 2022, från https://github.com/rdlowrey/auryn

Phinx documentation - 0.12. (u.å.). Hämtad 24 april 2022, från

https://book.cakephp.org/phinx/0/en/index.html

PHP 7 ChangeLog. (u.å.). Hämtad 20 april 2022, från https://www.php.net/ChangeLog-7.php

PHP 8.0 Released. (u.å.). Hämtad 24 april 2022, från https://www.php.net/releases/8.0/en.php

PHP 8.1 released. (u.å.). Hämtad 24 april 2022, från https://www.php.net/releases/8.1/en.php

Swift Mailer: A feature-rich PHP Mailer. (u.å.). Hämtad 30 maj 2022, från https://swiftmailer.symfony.com/

Symfony. (u.å.-a). *Home - Twig - The flexible, fast, and secure PHP template engine*. Symfony. Hämtad 24 april 2022, från https://twig.symfony.com/

Symfony. (u.å.-b). *Symfony, high performance PHP framework for web development*. Hämtad 24 april 2022, från https://symfony.com/what-is-symfony

Why should MVC for websites require a single point of entry? (u.å.). Hämtad 05 juni 2022, från https://localcoder.org/why-should-mvc-for-websites-require-a-single-point-of-entry

Wikipedia contributors. (u.å.-a). *Applikationsprogrammeringsgränssnitt*. Wikipedia, The Free Encyclopedia. https://sv.wikipedia.org/w/index.php?title=Applikationsprogrammeringsgr%C3%A4nssnitt&oldid=50495982

Wikipedia contributors. (u.å.-b). *Cascading Style Sheets*. Wikipedia, The Free Encyclopedia. https://sv.wikipedia.org/w/index.php?title=Cascading_Style_Sheets&oldid=50146228

Wikipedia contributors. (u.å.-c). *Javascript*. Wikipedia, The Free Encyclopedia. https://sv.wikipedia.org/w/index.php?title=Javascript&oldid=50285538

Wikipedia contributors. (u.å.-d). *Jquery*. Wikipedia, The Free Encyclopedia. https://sv.wikipedia.org/w/index.php?title=Jquery&oldid=50338648

Wikipedia contributors. (u.å.-e). *JSON*. Wikipedia, The Free Encyclopedia. https://sv.wikipedia.org/w/index.php?title=JSON&oldid=49913207

Wikipedia contributors. (u.å.-f). *Model-View-Controller*. Wikipedia, The Free Encyclopedia. https://sv.wikipedia.org/w/index.php?title=Model-View-Controller&oldid=49198603

Wikipedia contributors. (u.å.-g). *PHP*. Wikipedia, The Free Encyclopedia. https://sv.wikipedia.org/w/index.php?title=PHP&oldid=50378652

Wikipedia contributors. (u.å.-h). *UTF-8*. Wikipedia, The Free Encyclopedia. https://sv.wikipedia.org/w/index.php?title=UTF-8&oldid=42842282

Wikipedia contributors. (2021, oktober 3). *White-box testing*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=White-box_testing&oldid=1047942908

Wikipedia contributors. (2022a, februari 3). *Composer (software)*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Composer_(software)&oldid=1069762348

Wikipedia contributors. (2022b, februari 4). *PHPDoc*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=PHPDoc&oldid=1069917001

Wikipedia contributors. (2022c, februari 16). *Unit testing*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Unit_testing&oldid=1072241278

Wikipedia contributors. (2022d, mars 20). *Inversion of control*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Inversion_of_control&oldid=1078196018

Wikipedia contributors. (2022e, april 4). *Twig (template engine)*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Twig_(template_engine)&oldid=1081029142

Wikipedia contributors. (2022f, april 15). *Universally unique identifier*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Universally_unique_identifier&oldid=1082864872

Wikipedia contributors. (2022g, april 18). *Sass (stylesheet language)*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Sass_(stylesheet_language)&oldid=1083342371

Wikipedia contributors. (2022h, april 28). *Test stub*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Test_stub&oldid=1085070218

Wikipedia contributors. (2022i, april 29). *Plain old Java object*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Plain_old_Java_object&oldid=1085260116

Wikipedia contributors. (2022j, maj 10). *MySQL*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=MySQL&oldid=1087069729

Wikipedia contributors. (2022k, maj 14). *Progressive web application*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Progressive_web_application&oldid=1087730999

Wikipedia contributors. (2022l, maj 31). *XML*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=XML&oldid=1090717636

Wikipedia contributors. (2022m, juni 1). *Indexed Database API*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Indexed_Database_API&oldid=1090951765