# Functionele beschrijving

De ledenadministratie van Citroën Club Nederland (CCNL) omvat de volgende functies:

* Administreren leden
  + Nieuw lid opvoeren inclusief aanmaken van een welkomsbrief
  + Gegevens wijzigen, met name adres en bankgegevens.
  + Lid uitschrijven op verzoek van het lid inclusief bevestigingsmail
  + Lid uitschrijven wegens wanbetaling
  + SEPA-machtiging koppelen aan een lid
* Administreren van adverteerders in het clubblad  
  Dit ten behoeve van toezending van het clubblad
* Administreren van relaties  
  Dit ten behoeve van toezending van het clubblad
* Administreren van bestuursleden  
  Dit ten behoeve van toezending van het clubblad
* Adressenbestand voor verzending van het clubblad
* Adressenbestand voor jaarlijks versturen van ledenpassen
* Administreren van clubbladen die als onbestelbaar retour komen
* Jaarwerk
  + Resetten van betaalgegevens voor een clubjaar.
  + Aanmaken bestand voor automatische incasso van de jaarlijkse contributie
  + Afletteren van ontvangen betalingen
  + Genereren herinneringsbrieven
  + Archiveren van de stand van het afgesloten clubjaar
  + Verwijderen van leden die hebben opgezegd danwel niet betaald hebben.

Daarnaast is er het beheer van het mailadres van de ledenadministratie.   
Die is buiten scope voor het ledenadministratieprogramma.

Het ledenadministratie programma de volgende ondersteunende functies.

* Beheren documenten bij een lid
* Importeren van gegevens uit Excel
* Exporteren diverse overzichten naar Excel
* Backup/restore van de database
* Beheer van gearchiveerde gegevens
* Beheer van templates die worden gebruikt bij het genereren van output

# Randvoorwaarden voor de applicatie

De ledenadministratie wordt door één persoon uitgevoerd op een persoonlijke Windows-PC. Er is geen behoefte aan authenticatie/autorisatie voor het gebruik van de applicatie. Ook is het beveiligen (encryptie) van gegevens niet aan de orde.

# Uitgangspunten

Er is gekozen om de applicatie te ontwikkelen met Java. (Domweg omdat ik een Java ontwikkelaar ben.)

De applicatie is een Spring Boot applicatie. Spring Boot is de facto de standaard in Java voor het structureren van functionaliteit in componenten. Voor de User Interface is gekozen voor JavaFX omdat een desktop applicatie het meest aansluit bij de behoefte.

Er is gekozen de schermen te modelleren in FXML om een zo goed mogelijke scheiding te krijgen tussen de View (FXML) en Controller (JavaFX) conform het MVC-paradigma.

Gegevens worden opgeslagen in een lokale relationele database. Er is gekozen voor een H2-database omdat dit naadloos aansluit bij Spring Boot en omdat er voor de single user applicatie geen behoefte is aan meer functionaliteit.

De deployable applicatie wordt, via de spring-boot-maven-plugin, opgeleverd in de vorm van een jar-bestand. Dit jar-bestand kan worden uitgevoerd op een Windows-PC met een Java-14 Runtime omgeving.

# Ontwerpbeslissingen

## Gelaagde structuur

De applicatie is opgezet als een Maven-3 project. Binnen dit project zijn een aantal Maven-modules onderkend die tezamen de applicatie een functioneel gelaagde structuur geven:

1. De presentatie laag
   * Module **javafx-springboot** met generieke functionaliteit
   * Module **frontend** met de User Interface van de Ledenadministratie
2. De service laag t.b.v de presentatielaag
   * Module **services**
3. De functionele laag welke wordt aangestuurd vanuit de services laag.
   * Module **annual-ccnl** voor het jaarwerk
   * Module **excel-ccnl** voor import uit en export naar MS-Excel spreadsheet
   * Module **output-ccnl** voor de aanmaken van documenten in MS-Word en PDF.
4. De data-laag
   * Module **common-ccnl** met data structuren en integratie met de database.

Er is een klein aantal generieke utilities onderkend. Omdat het zo’n klein aantal is, is daarvoor geen aparte Maven-module onderkend. Zij zijn opgenomen in **common-ccnl**.

## Spring Boot JavaFX integratie.

Het framework FxWeaver pretendeert integratie tussen Spring Boot en JavaFX te bieden. Naar mijn mening is zo’n framework niet nodig omdat Spring Boot en JavaFX al op een natuurlijke wijze te combineren zijn.

De generieke basis opzet is als volgt:

* De class met de initiële main() method is een class met de annotatie @SpringBootApplication.
* In de main() method wordt een class gestart die een extensie is op javafx.application.Application
* In de init() van de javafx.application.Application extensie wordt de Spring Boot ApplicationContext geconfigureerd. Deze ApplicationContext wordt expliciet met headless=false geconfigureerd omdat de applicatie gebruik maakt van java.awt.print.PrinterJob.
* In de start() van de javafx.application.Application extensie wordt een StageReadyEvent verstuurd met daarin de ‘primary stage’ voor de user interface.

De frontend van ledenadministratie vangt het StageReadyEvent op en plaatst de user interface in de ‘primary stage’ van het event en roept de show() methode aan.

Een JavaFX-applicatie heeft normaliter een modulaire Java-structuur. In bovenstaande opzet is de applicatie primair een Spring Boot applicatie; dus zonder modulaire Java-structuur.

De in FXML gedefinieerde visuele componenten bevatten waar nodig het fx:controller attribuut voor verwijzing naar de gerelateerde Controller component. De FXML-definities worden geladen door de javafx.fxml.FXMLLoader . In zo’n loader wordt via fxmlLoader.setControllerFactory(springContext::getBean); het fx:contoller attribuut uit de FXML gekoppeld aan een Spring Boot @Controller component.

Om het opstarten van de applicatie te bevorderen is Spring Boot geconfigureerd met **lazy-initialization:** true en **jpa.repositories.bootstrap-mode:** lazy.   
Dus pas bij het laden van de FXML wordt de Controller daadwerkelijk geïnstantieerd. De database wordt pas geïnitieerd bij de eerste benadering vanuit de User Interface. Er treedt dan een korte, maar merkbare vertraging op.

## FXML

Er is gekozen voor lazy loading van de FXML-bestanden. Bij het eerste gebruik wordt zo’n bestand geladen via de javafx.fxml.FXMLLoader. Hierbij wordt een JavaFX Parent component aangemaakt en worden de controls en actions in zo’n Parent gekoppeld aan de bijbehorende Spring Boot Controller. Ook wordt de initialize() method in de Controller aangeroepen. Dit alles gebeurt op basis van de @FXML annotatie op de betreffende properties en methods in de Controller.

De aangemaakte Parent van een FXML-bestand wordt vervolgens in een cache bijgehouden. Dit betekent dat bij een vervolggebruik de initialize() method in de Controller *niet* meer wordt aangeroepen. Waar nodig moet zo’n controller expliciet worden gereset voor hergebruik!

## Structuur van de User Interface

De primaire pagina van de applicatie is een BorderPane.   
In de Top-sectie van de BorderPane is een fx:include gedefinieerd voor de menu FXML. In de Center-sectie wordt steeds de pagina geladen die functioneel relevant is.   
De Bottom-sectie wordt gebruikt voor het weergegeven van informatie- en foutmeldingen.   
De Left- en Right-secties worden niet gebruikt.

De menu FXML heeft een referentie naar de MenuController component. Het is deze Controller die iedere dialoog van de applicatie aanstuurt. Om een ontkoppeling tussen de MenuController en de rest van de controller te bereiken is gekozen voor het sturen van org.springframework.context.ApplicationEvents vanuit de MenuController. De overige controllers reageren indien zo’n event relevant is. Relevantie wordt bepaald op basis van de nl.ealse.ccnl.control.menu.MenuChoice enum in het event.

Een menu keuze kent drie varianten:

1. Het bewerken van een formulier; hierbij is een gerelateerde Model object nodig; (conform MVC-paradigma).
2. Het uitvoeren van een commando met scherminvoer.
3. Het uitvoeren van een commando zonder scherminvoer.

De eerste twee varianten zijn gekoppeld aan een FXML. De Spring Boot controller zal worden geladen met het laden van de FXML.   
De derde variant is niet gekoppeld aan een FXML. Zo’n Spring Boot Controller kan geen lazy loading hebben, omdat het dan niet kan reageren op een event vanuit de MenuController. Zo’n controller worden geladen met de annotatie @Lazy(**false**). Voorkomen moet worden dat zo’n Controller de gerelateerde services op voorhand laadt, want dat wordt het principe van zoveel mogelijk lazy loading geweld aangedaan. De services worden in dergelijke Controller dan ook on demand geladen vanuit de Spring ApplicationContext.

De Spring Boot Controller van de centrale BorderPane heeft als voornaamste taak het inladen van de juiste pagina in de Center-sectie van de BorderPane. Deze PageController wordt primair aangestuurd vanuit de MenuController.

Dit werkt als volgt:

* De MenuController geeft de PageController de opdracht een pagina te laden.
* De PageController vraagt de JavaFX Parent van de pagina op. Deze komt uit de cache indien al eerder geladen en naders wordt de Parent vanuit het relevante FXML-bestand aangemaakt.
* Bij het laden van de FXML is de bijbehoren Spring Boot Controller geïnstantieerd. Deze controller kan dus reageren op events.
* De MenuController vuurt vervolgens het event af dat hoort bij de menu keuze.
* De bij de geladen pagina horende controller verwerkt het relevante event.
* Voor eventuele vervolgschermen stuurt zo’n controller de PageController aan

De PageController heeft methodes om de Info- en Error-messages te tonen in de Bottom-sectie van de BorderPane.

De PageController heeft ook een methode om een pagina te laden zonder deze te tonen. Functioneel hoort dit niet tot de taak van de PageController, maar het is wel handig vanuit het perspectief van de rest van de applicatie.

Bij een menukeuze met een formulier bevat het event ook het Model Object. Bij een nieuw Model Object is het de MenuController die zo’n nieuwe instance in het event plaats.

Bij een reeds bestaand Model is het een zoekdialoog die tot de juiste instantie leidt en een event afvuurt met het juiste Model Object.

Bij het opvoeren van een lid is conceptueel sprake van een formulier in de User Interface.   
(Anders dan HTML kent JavaFX/FXML hier geen specifieke structuur voor.)   
Er is gekozen zo’n formulier te verdelen over meerdere FXML pagina’s. Al deze pagina’s zijn gekoppeld aan dezelfde controller! Dus bij het laden de FXML wordt per pagina de initialize() method in de controller aangeroepen! Er zijn dus neveneffecten aan deze keuze, maar anderzijds is er een eenvoudige interactie mogelijk met het Model Object waarop het formulier betrekking heeft. (Bij mijn weten ondersteunt FxWeaver een dergelijke opzet niet.)

De properties van een Model Object moeten worden gemapt naar de JavaFX-controls van het formulier. Bij een *reset* op een scherm moet zo’n mapping herhaalbaar zijn. Omgekeerd moet bij opslaan de inhoud van de JavaFX-controls worden gemapt naar het Model Object.   
Voor deze mapping is binnen de applicatie een generiek framework ontwikkeld, waarbij de JavaFX controls van het formulier leidend zijn.

Een formulier moet worden gevalideerd alvorens de mapping naar het Model Object gebeurt. Ook hiervoor is binnen de applicatie een generiek framework ontwikkeld.

## Database

De database omvat de volgende soorten persistent objecten:

* Model Objecten voor Leden, Adverteerders, Clubs, Relaties en Functies
* Documenten bij Leden
* Configuratie (algemeen en voor incasso)
* Document templates
* Betaalgegevens
* Gearchiveerde gegevens

De persistente objecten zijn middels JPA-annotaties geconfigureerd. Database benadering gebeurt via JpaRepositores.

Bij de eerste keer starten van de applicatie wordt via een EnvironmentPostProcessor de database geinitieerd en worden de defaults voor de algemene configuratie geladen. Dit initiële laden gebeurt asynchroon om het opstarten zo min mogelijk te vertragen.

## Gebruikte frameworks

* Lombok
* commons-validator  
  Voor het valideren van betaalbestanden
* Apache fop  
  Voor het genereren van PDF’s
* icepdf-core  
  Voor de rendering van PDF-pagina’s t.b.v. de User Interface
* Apache pdfbox  
  Voor het printen van PDF’s
* Apache POI  
  Voor het lezen een aanmaken van Excel spreadsheets en voor het aanmaken van een Word docx-document

## Het programma draaien

In het opstart commando kan de database locatie worden opgegeven met de parameter ‘spring.datasource.url’; bijvoorbeeld:

**javaw.exe -jar LedenadministratieCCNL.jar --****spring.datasource.url=jdbc:h2:C:/ db/ccnl**

Het programma maakt een logs-map aan voor de logbestanden. Verder configuratie gebeurt via de menukeuze **Beheer > Instellingen**.

De tijd tussen het starten van de applicatie en het tonen van het eerste scherm is, ondanks lazy loading, ruim vijf seconden. Gedurende die tijd wordt een eenvoudig Java Splash-scherm getoond.

# Realisatie functionaliteit

## Aanmaken incasso bestand

Voor het aanmaken van het incassobestand is code van een vorige applicatie hergebruikt.

In de Maven build wordt JAXB-code gegenereerd op basis van het XML-schema camt.053.001.02.xsd. Via een tweetal builders worden per lid de voornaamste blokken van het incassobestand aangemaakt. Het incassobestand wordt uiteindelijk geserialiseerd via JAXB.

Naast het incassobestand wordt een Excel bestand aangemaakt met een overzicht van de leden waarvoor een incasso is aangemaakt.

Tevens wordt de betaalstatus van de leden in de database bijgewerkt.

## Verwerken betaalgegevens

Voor het verwerken van betaalgegevens is code van een vorige applicatie hergebruikt.

De functionaliteit valideert allereerst of een geladen bestand een valide betaalbestand is conform ‘camt.053.001.02.xsd’.   
Vervolgens wordt aan de hand van XPath-selecties geprobeerd te achterhalen of een financiële transactie betrekking heeft op contributie.

Hierna wordt voor iedere contributie geprobeerd te achterhalen op welk lid de transactie betrekking heeft. Helaas is sinds de privacy wetgeving matching op adres niet langer mogelijk. Toch kunnen vrijwel alle transacties correct worden gekoppeld aan een lid.  
Uiteindelijk wordt een transactie verwerkt in de financiële status van het overeenkomende lid.

Mocht het niet lukken een contributie uniek toe te wijzen aan een lid dan wordt dit gelogd in een speciaal logbestand.

## Aanmaken tonen en printen van documenten

Documenten worden primair in PDF-formaat aangemaakt. De basis is een template in de vorm van XSL-FO. In de User Interface kunnen content-templates met een set substitutie- en opmaak tags worden gedefinieerd.

Bij het genereren van een PDF vervangt een FO-generator de substitutie-tags door de gewenste gegevens en plaatst de content vervolgens in de XSL-FO template conform de opmaak tags. Deze complete XSL-FO wordt middels Apache FOP omgezet naar een PDF.

De applicatie heeft ook de mogelijkheid de welkomsbrief voor een nieuw lid te generen als een docx-document dat met Word bewerkt kan worden. De gebruikte benadering is dezelfde als voor een PDF, maar dat op basis van een docx-template. Ook hier vervangt een generator de substitutie-tags door de gewenste gegevens en plaatst de content vervolgens in de docx template. Het resultaat wordt opgeslagen en kan buiten de applicatie met MS-Word worden bewerkt. De generator werkt op basis van het usermodel van Apache POI XWPF.

Het tonen van een PDF in de User Interface behelst allereerst de rendering van een PDF-pagina tot een image die vervolgens in een PdfViewer wordt getoond. Er kon op internet geen out-of-the-box oplossing worden gevonden. IcePdf heeft de mogelijkheid een PDF te tonen een Swing PdfViewer.

Het bleek mogelijk een deel van IcePDF te gebruiken voor de rendering van PDF-pagina’s naar Swing images. Zo’n Swing image is eenvoudig te converteren naar een JavaFX image. Op basis hiervan is een eenvoudige viewer gemaakt waarmee door de pagina’s van de PDF gebladerd kan worden.  
Deze viewer wordt om meerdere plaatsen in de applicatie gebruikt.

Printen van PDF’s gebeurt door de byte[] te laden in een org.apache.pdfbox.pdmodel.PDDocument die vervolgens wordt afgedrukt via een java.awt.print.PrinterJob.

Bij het beëindigen van het lidmaatschap op verzoek wordt een bevestigingsmail gestuurd naar de klant. Dit gebeurt ook weer op basis van een content-template met een set substitutie- en opmaak tags.

Herinneringsbrieven bij te late betaling zijn ook PDF’s die worden gegenereerd op basis van een content-template met een set substitutie- en opmaak tags.

Gegenereerde documenten worden als PDF opgeslagen bij het betreffende lid.

## Verwerken en aanmaken Excel spreadsheets

Voor Excel Spreadsheet is code van een vorige applicatie hergebruikt. Deze code bestaat uit wrappers rond het usermodel van ApachePOI.

* CCNLBestand rond een (XSSF)Workbook
* CCNLSheet rond een Sheet  
  (Met subclasses voor Leden, Adverteerders, Clubs, Relaties en Functies)
* CCNLRow rond een Row  
  (Met subclasses voor Leden, Adverteerders, Clubs, Relaties en Functies)

De mapping van kolom nummers uit spreadsheets naar java properties ligt vast in het bestand ‘excel.properties’. Dit bestand map tevens de onderkende sheets naar hun functionele betekenis.  
Tenslotte zijn er een aantal properties die worden gebruikt om specifiek waarden uit een spreadsheet te mappen op enums.

## Backup en Restore van de database

Er is gekozen voor een structuur waarbij zowel backup als restore kunnen plaatsvinden in de draaiende applicatie. Bij de backup wordt een ZIP-bestand met SQL-statements aangemaakt. Bij restore worden deze SQL-statements uitgevoerd.

## Asynchrone verwerking

De User Interface van JavaFX is single threaded; (de FX application thread). Bij een langlopend achtergrond proces zal de User Interface dan ook niet reageren tot zo’n proces klaar is. JavaFx kent een concurrency API als superset op de normale java concurrency API. Deze API is bedoeld om zware processen op de achtergrond uit te voeren waarbij tussentijdse feedback naar de User Interface mogelijk is.

Bij een aantal lang lopende processen wordt een melding getoond bij de start van het proces een melding en als het langlopende proces is afgerond. Dit blijkt alleen goed realiseerbaar met behulp van de JavaFX concurrency API. Per relevant proces is een class als extensie op javafx.concurrent.Task gedefinieerd. Hierin wordt het achtergrond proces uitgevoerd. Zo’n Task-extensie wordt uitgevoerd dor de SpringBoot TaskExecutor.

Voor het opslaan van de herinneringsbrief per lid is een asynchroon proces gebruikt op basis van Spring Boot.

# Unit testen (met JUnit5)

## Spring Boot testen

Testen middels @SpringBootTest is een erg zware manier van testen omdat er steeds een Spring-omgeving gestart moet worden. Het is mogelijk tests te combineren in een test suite waarbij de omgeving maar eenmalig wordt geïnitieerd. Zo’n test suite blijkt in de **maven-surefire-plugin** problemen op te leveren in combinatie met individuele tests; (Begin 2021 is versie 2.22.2 de GA-versie van de maven-surefire-plugin). De milestone build 3.0.0.M5 lijkt geen last te hebben van dit probleem.

Spring Boot test zijn daarom alleen gebruikt om de JpaRepositories op interactie met de database te testen.

## Testen JavaFX

Het is erg lastig componenten met JavaFX controls te unit testen. Het blijkt niet mogelijk mock-objecten te gebruiken voor JavaFX-controls. Er moet dus worden getest met de echte controls.

Die controls vereisen allereerst dat de JavaFX tool gestart is. Verder is er een draaiende FX Application thread nodig voor Scene en Stage objecten. Zo’n FX Application thread draait asynchroon t.o.v. de unit-test thread.; (middels Platform.*runLater*() ). Via Awaitility.*await*() wacht de unit-test thread tot het asynchrone proces klaar is, waarna de Assertions gedaan kunnen worden.

De maven-surefire plugin werkt standaard met één fork JVM-proces dat steeds wordt hergebruikt. Er mag maar één JavaFX tool tegelijk draaien in de fork omdat de fork anders vastloopt. Standaard wordt door de plugin ook maar één thread gebruikt. Opvallend genoeg blijkt het gebruik van één thread (op een 8 core PC) efficiënter dan het werken met een thread per CPU.

Er is gekozen om de JavaFX tool voor alle tests eenmalig te starten in static code van de superclass die wordt gebruikt voor iedere test. Het starten gebeurt via het instantiëren van een JFXPanel.

Starten van een JavaFX omgeving en een FX Application thread kan ook via het test framework TestFX. Dan wordt dus per test class een JavaFx omgeving gestart. Het framework wordt mede om die reden niet gebruikt. De andere reden is dat het framework zich met name focust op de controls, terwijl dat met het gebruik van FXML minder interessant is. De unit tests voor de Ledenadministratie richten zich met name op de events en acties die in een FX-controller worden uitgevoerd.

## Mockito

Testen met gebruik van Mockito is gebruikt waar mogelijk.

## Private Fields/Methods in te testen objcten.

Bij het testen is het soms nodig private Fields te zetten of Methods aan te roepen. Dit gebeurt op basis van de utilities in Apache commons-lang3.

## Functionele tests

Zijn (vooralsnog) geen functionele tests gemaakt!   
**TODO**: cucumber tests toevoegen!

# Kwaliteitscontrole

Via de maven-checkstyle-plugin wordt afgedwongen dat de code voldoet aan de code standaarden conform google\_checks.

Via jacoco-maven-plugin wordt de dekkingsgraad van Unit tests gemeten; (jacoco = Java Code Coverage).

In het project in een speciale module opgenomen waarin de geaggregeerde code coverage gegevens worden verzameld. Via de site van deze module kan het resultaat als webpagina worden getoond.

(Daarnaast is de lokale build gekoppeld aan een SonarQube webapplicatie waarin de applicatie een dekkingsgraad van 80,7% heeft.)