

# Reverse Pulse Plate Automation

Realisatiedocument

Emalisa Antonioli  
Student Bachelor in de Toegepaste Informatica – AI

# Inhoudsopgave

<b>1. INLEIDING</b>	<b>4</b>
<b>2. ANALYSE</b>	<b>5</b>
2.1. Gebruik Reverse Pulse Plater	5
2.2. RS485 Communicatie	6
2.3. Software	6
<b>3. FYSIEKE OPSTELLING</b>	<b>8</b>
<b>4. APPLICATIE GUI</b>	<b>9</b>
4.1. Main Window	9
4.2. Setup	10
4.3. Overview	12
4.4. Login/Account	14
4.5. Configuratie	15
4.6. Gebruikersbeheer	16
<b>5. APPLICATIE BACKEND</b>	<b>18</b>
5.1. Centrale Signaal Zender	18
5.2. Profielen	19
5.3. Apparaten	21
5.4. Communicatie	22
5.5. Gebruikers	23
<b>6. BESLUIT</b>	<b>25</b>
<b>7. LITERATUURLIJST</b>	<b>26</b>
<b>BIJLAGE A</b>	<b>27</b>
<b>BIJLAGE B</b>	<b>28</b>

# 1. Inleiding

Reverse Pulse Plate Automation. Zo noemt dit stageproject. De eerste drie woorden zullen bij de meeste lezers niets oproepen. Laten we daarom beginnen met een korte uitleg daarover. De stage werd gevolgd bij Cercuits, een bedrijf dat specialiseert in het produceren van keramieken printplaatjes (PCBs). Vroeger werden PCBs gemaakt door kleine koperdraadjes aan te brengen op een plaatje, vandaag de dag wordt op veel kleinere schaal gewerkt. Het is nu gebruikelijker om een hele dunne laag koper aan te brengen op de plaatjes en al het overbodige weg te etsen of laseren. Het *reverse pulse platen* is een speciale techniek om een dun laagje koper aan te brengen op kleine gaatjes (via's) in een plaatje (NetVla Group, 2025). De plaatjes worden hierbij in een bad gehangen met koperionen. Aan het plaatje wordt een kathode bevestigd en in het bad bevindt zich een anode. Door een positieve of een negatieve stroom door het bad te laten gaan kunnen er koperionen op het plaatje bevestigd worden, of er juist afgehaald worden.

Bij Cercuits worden deze baden aangestuurd door apparaten die rectifiers heten, maar die in dit verslag RPPs (*Reverse Pulse Platers*) genoemd zullen worden. Op deze apparatuur kunnen de verschillende parameters (zoals duur en intensiteit van de stroompulsen) ingevoerd worden via knopjes en een LCD-scherm. Het doel van de stage is om dit op de computer te kunnen doen, via een overzichtelijke applicatie. Dit zou de operators bijvoorbeeld in staat stellen om verschillende profielen in te stellen voor de parameters en om alle RPPs vanuit een centrale plaats te beheren, wat niet alleen comfortabeler werkt, maar ook gebruikersfouten kan voorkomen.

Er is dus een applicatie gemaakt vanwaar de RPPs aangestuurd kunnen worden. Verschillende combinaties parameters kunnen opgeslagen worden in profielen, om deze makkelijk te kunnen hergebruiken. Er is in de applicatie ook een pagina met een overzicht van alle aangesloten RPPs en hun status. Als er handelingen nodig zijn van de gebruiker, als er bijvoorbeeld een programma is afgelopen, krijgen ze hier een duidelijke melding van. Fouten met de verbinding of de output van de RPPs worden ook gemeld, alhoewel op dit gebied nog veel uitbreiding mogelijk is. Om bepaalde functionaliteiten af te schermen van bepaalde gebruikers, zijn er accounts met verschillende rollen. Deze accounts kunnen door de gebruiker zelf bewerkt worden, of door een centrale beheerder. Als laatste is er natuurlijk een duidelijke pagina om de variabelen in te stellen die nodig zijn voor de verbinding met de RPPs. Dit klinkt nu als heel veel, maar we gaan daar in de loop van het verslag rustig overheen gaan.

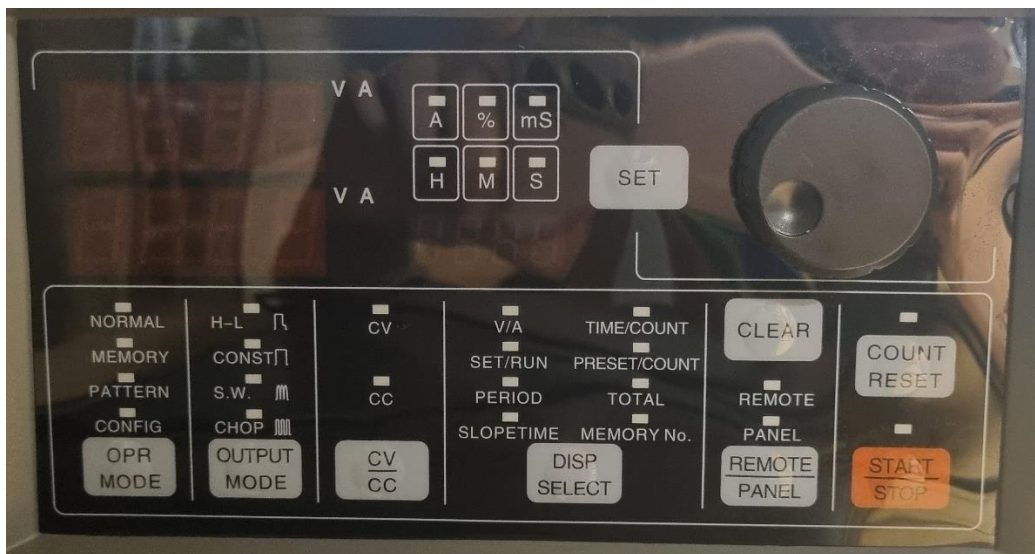
Er zal begonnen worden met een overzicht van de analyse die in de eerste twee weken van het project is gemaakt. Hier komt een verantwoording voor de grootste keuzes aan bod, zoals de gekozen software frameworks. Daarna volgt een overzicht van de fysieke opstelling die benodigd is voor de werking van de applicatie. Hierna kijken we naar de frontend, i.e. de visuele kant van de applicatie. Na de frontend nemen we de backend door, dit is een vrij technisch hoofdstuk en zal dus niet voor alle lezers relevant zijn. We zullen eindigen met een samenvattend hoofdstuk, welke een reflectie op het project zal bevatten. Het is onvermijdelijk dat niet ieder detail van het project besproken zal worden, maar ik ben ervan overtuigd dat dit verslag een duidelijk beeld zal schetsen van de kwaliteit van het behaalde eindresultaat van mijn bachelorproject en het bijbehorende proces.

## 2. Analyse

Alles in dit project is weloverwogen. In dit hoofdstuk zien we hoe en waarom, aan het begin van het project, de grootste van deze keuzes gemaakt zijn. We kijken hierbij specifiek naar de analyse voor de communicatie tussen de RPPs en de computer, naar de keuzes voor het GUI framework en we eindigen met een kort overzicht van de gekozen softwarearchitectuur.

### 2.1. Gebruik Reverse Pulse Plater

Om een applicatie te kunnen schrijven om de RPPs te kunnen gebruiken, moeten we eerst snappen hoe de RPPs nu gebruikt worden, zonder applicatie. Dit kunnen we het beste doen door te kijken naar het aansturingspaneel in **Error! Reference source not found.**



Figuur 1. Aansturingspaneel RPP

We zien links op het paneel de *operational modes*. Voor ons zijn *normal* en *config* belangrijk. In *config* kunnen we de ID en de baudrate vinden, deze waarden zullen terugkomen wanneer de Modbus communicatie verder besproken wordt. De waarden die we hier op de RPP kunnen vinden, gaan overeen moeten komen met de waarden in onze applicatie. Het meeste van de functionaliteit van de RPPs gebeurt in de *normal mode*.

In de *normal mode* moeten we een *output mode* kiezen, hier zijn er vier van. Iedere *output mode* komt met zijn eigen parameters, de *const mode* heeft bijvoorbeeld enkel een totale looptijd, een aanlooptijd, en een output waarde, terwijl de *s.w. (sigmoidal wave) mode* naast een totale looptijd een minimum- en maximumwaarde en een frequentie heeft. De rest van het paneel wordt gebruikt om deze parameters in te stellen.

Het gebruik van de RPPs zal vereenvoudigd worden door deze parameters op te slaan in profielen, waarvan er vier soorten nodig zullen zijn: een voor iedere *output mode*. De overige functionaliteiten van de RPP, zoals het bijhouden van de geschiedenis, zullen buiten de afbakening van het huidige project vallen.

Om ervoor te zorgen dat er evenveel koper op de voor- als achterkant van de PCBs komt, worden deze vaak halverwege het programma omgedraaid. Gebeurt dit niet exact halverwege, dan kan dit de kwaliteit van de PCB negatief beïnvloeden. De RPP zelf heeft echter geen optie om het programma halverwege te pauzeren. Met de magie van software gaan we deze functionaliteit wel kunnen introduceren.

Als een PCB lang in een bad blijft hangen nadat een programma is afgelopen (of gepauzeerd) kan dit ook negatieve effecten hebben op de kwaliteit van de PCB. We moeten er dus voor zorgen dat er een duidelijke melding is om deze situaties aan te geven. De operators zullen niet altijd in de buurt van de computer zijn, een geluidsalarm zal daarom niet een effectieve oplossing zijn. We zullen waarschijnlijk het beste resultaat boeken met een opvallend visueel signaal dat van een verre afstand al te zien is.

Verder moeten we er rekening mee houden dat er een variëteit van mensen met de applicatie zal werken. Er kunnen procesoperators komen die een krak zijn in chemie maar snel in de war raken van computers. Voor hen moeten we zien dat het gebruik van de applicatie duidelijk en simpel is. Cercuits is ook een bedrijf waar vaak stagestudenten werken, we gaan functionaliteiten die dataopslag kunnen aanpassen van hen willen afschermen.

## 2.2. RS485 Communicatie

Bij Cercuits wordt gebruik gemaakt van RPPs met ingebouwde RS485 modules. RS485 is een veelvoorkomende interface voor communicatie tussen computers en elektrische machines (Smoot, 2025). De producent raadt aan om gebruik te maken van een serieel Modbus-protocol voor de communicatie, dus dat is exact wat ik heb gedaan. Modbus is een veelgebruikt protocol om de besturing van elektrische apparaten te kunnen programmeren (Baselier, 2025).

De kabels tussen de RPPs en de computer maken dus contact met de RPPs via de RS485 module, met de computer zijn ze verbonden via een RS485 adapter. De producent van de RPPs had een handleiding beschikbaar voor RS485 communicatie, waarin werd aanbevolen om alle RPPs via eenzelfde adapter met de computer te verbinden. Na advies van de stagebegeleider is echter besloten om een adapter te gebruiken per RPP, en iedere RPP dus hun eigen COM port te geven. Dit zal de initiële ontwikkeling en toekomstige uitbreiding makkelijker maken. De fysieke opstelling zal verder behandeld worden in Hoofdstuk 3.

## 2.3. Software

Een goed werkende verbinding tussen de RPPs en de PC is niet veel waard zonder goede software om ze aan te sturen. De applicatie die tot stand gebracht moet worden heeft geen nood aan een visueel uitgebreide *Graphical User Interface* (GUI). Het moet een makkelijke, intuïtieve manier zijn om de RPPs te bedienen. Er hoeven geen uitgebreide visualisaties op te komen en het moet makkelijk te integreren zijn met Modbus-communicatie. De app zal werken als een *executable* zonder internetverbinding. Bovendien werd het verzoek gedaan om een framework te gebruiken dat makkelijk te leren was voor mensen met voornamelijk ervaring in Python, zodat het in de toekomst makkelijk door anderen onderhouden en uitgebreid kon worden.

Bij een eerder, gelijkaardig project werd Qt (The Qt Company Ltd., 2025) gebruikt, specifiek de open-source, Python versie PySide6 (The Qt Company Ltd., 2025). Voor dit project is PySide6 ook de uiteindelijke keuze geworden. Deze keuze is niet alleen gemaakt omdat het bedrijf al ervaring had met het framework, hoewel dat er wel een onderdeel van was, maar voornamelijk omdat het simpelweg de beste keuze was.

De reden dat toch voor Python is gekozen, en niet voor een andere taal zoals Java, C# of JavaScript, is omdat Python exceptioneel goed is voor Modbus-protocollen. Python beschikt over verschillende, goed gedocumenteerde *libraries* zoals Pymodbus en Pyserial, die makkelijk te integreren zijn met alle Python code. Dit in combinatie met het gegeven dat zowel mijn begeleider als ik allebei graag werken met Python maakt dit een makkelijke keuze voor de backend. Nu zouden we voor de GUI natuurlijk een andere taal hebben kunnen kiezen, zeker omdat Python hier geen gebruikelijke taal voor is. Als in de toekomst complexere visualisaties toegevoegd zouden moeten worden aan de applicatie zou dit een waardevolle optie zijn; de applicatie is dan ook geschreven met een gescheiden frontend en backend om deze optie open te houden. Echter, aangezien de eisen voor de GUI voor de huidige versie van de applicatie zo simplistisch zijn, heb ik ervoor gekozen om de simpele route te nemen en de GUI ook te programmeren in Python.

In Python zijn er verschillende frameworks bekeken voor de GUI. Qt kwam als winnaar uit de verf omdat deze makkelijk is om mee te werken, cross-platform ondersteuning en geïntegreerde eventhandling heeft, er is

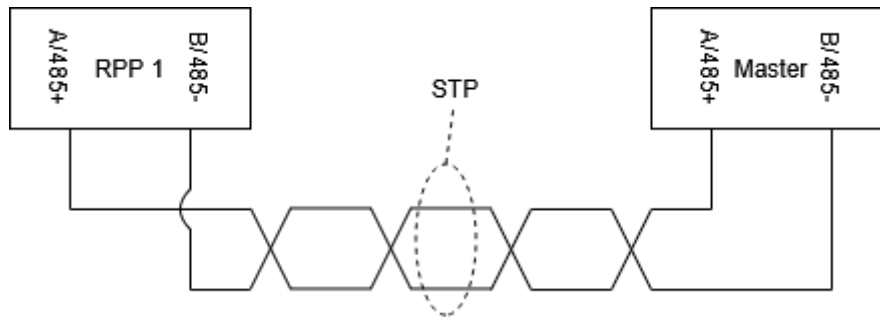
veel documentatie over te vinden en het heeft dit allemaal zonder dat het te zwaar wordt. Qt is er in vele varianten. De originele taal voor dit framework is C++. Het voordeel van C++ is dat het sneller is dan Python, maar dat is voor onze applicatie niet relevant. Ook voor de Python versie van Qt zijn er nog verschillende opties. Wij zijn gegaan voor PySide6 omdat dit de gratis, opensource versie is. Dit betekent echter wel dat voor commercieel gebruik van de applicatie de originele Qt-bibliotheken ongewijzigd moeten blijven of dat er een commerciële licentie aangeschaft moet worden. Voor een meer grafisch overzicht van de bovenstaande keuze, in de vorm van een gewogen beslissingsmatrix, kan gerefereerd worden naar Bijlage A.

Verder is er gekozen om een modulaire *Model View Controller* (MVC) structuur aan te houden in de applicatie. Dit houdt in dat, zoals eerder al kort aangehaald, de verschillende componenten van de applicatie zo min mogelijk onderlinge afhankelijkheden hebben. Dit om toekomstige uitbreidingen zo makkelijk mogelijk te maken. De MVC-structuur helpt hierbij door de code op te splitsen in *models*, welke in de backend zitten en zakelijke logica bevatten, *views*, welke in de frontend zitten en het visuele aspect op hun handen nemen, en *controllers*, welke de communicatie tussen de *models* en *views* faciliteren. De implementatie hiervan zal duidelijker worden in Hoofdstuk 4 en Hoofdstuk 5.

Om af te sluiten was het belangrijk dat er *event-based* gewerkt zou worden. Er is gekeken naar verschillende frameworks om dit te faciliteren, zoals Dispatcher (oliviervedier, 2012) en Blinker (Blinker, 2010). Het bleek uiteindelijk echter dat Qt hier geïntegreerd *Slots* en *Signals* voor gebruikt, waarbij *Slots* worden gebruikt als *Listeners* en *Signals* als *Events* (The Qt Company Ltd., 2025).

### 3. Fysieke Opstelling

Zoals eerder vermeld is er besloten om iedere RPP via een RS485 adapter aan te sluiten op een eigen COM port. Figuur 2 toont een systematische weergave van de bekabeling van een RPP tot de computer (de *master*). We zien hier dat de bekabeling erg simpel is en dat er best gewerkt kan worden met *Shielded Twisted Pair (STP)* kabels. Echter, door gebrek aan voldoende STP-kabels zijn in de uiteindelijke opstelling ook RPPs bekabeld met *Unshielded Twisted Pair* kabels, door de korte afgelegde afstanden zou dit geen problemen moeten opleveren.



Figuur 2. RS485 Aansluitschema

Een leuk extraatje tijdens de stage was het bouwen van een opstelling om de computer en randtoebehoren neer te zetten, waar er eigenlijk geen plaats voor was. Figuur 3 laat het uiteindelijke resultaat hiervan zien. Er is gebruik gemaakt van PEG profielen om het scherm en de muis met het toetsenbord te laten 'zweven' boven twee van de RPPs. De computer staat er onder de tafel om het ietwat te beschermen tegen rondvliegende chemicaliën. Voor een niet-mechanisch aangelegd persoon als mijzelf was dit een interessante opdracht. Door mijn vele vragen was het ook een goede kans om de rest van het team beter te leren kennen.



Figuur 3 Computeropstelling om RPPs te bedienen.



## 4. Applicatie GUI

Dit hoofdstuk biedt een overzicht van de gebruikersinterface (GUI) van de applicatie en is geschreven met zowel technische als niet-technische lezers in gedachten. Het doel is om inzicht te geven in hoe de applicatie werkt en hoe verschillende pagina's van de applicatie zijn ontworpen en bijdragen aan een gebruiksvriendelijke ervaring.

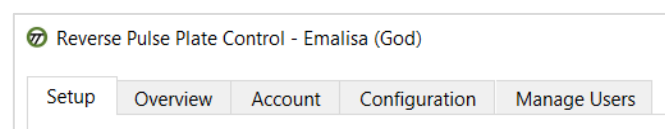
De gebruikersinterface is opgebouwd uit verschillende pagina's, elk met een specifieke functie. Van de overzichtspagina, waar alle gebruikers de status van apparaten kunnen volgen, tot de configuratie- en gebruikersbeheerpagina's, die exclusief toegankelijk zijn voor beheerders, elke pagina is ontworpen met een duidelijke focus op gebruiksgemak en functionaliteit.

### 4.1. Main Window

De volledige applicatie speelt zich af in de *MainWindow*. Deze is ontworpen als een container waarin alle andere pagina's worden geladen, georganiseerd via een *QTabWidget* (The Qt Company Ltd., 2025). Dit modulaire tabbladensysteem biedt een duidelijke en intuïtieve navigatie door de applicatie en maakt het eenvoudig om de GUI uit te breiden of aan te passen.

Een opvallend kenmerk van de *MainWindow* is een dynamisch achtergrondkleursysteem. Verschillende signalen, zoals een succesvol afgerond proces, een waarschuwing, of een fout kunnen de achtergrondkleur van het venster veranderen. Omdat meerdere apparaten tegelijkertijd verbonden kunnen zijn, heeft elk apparaat de mogelijkheid om een gewenste achtergrondkleur te specificeren. Deze kleuren worden opgeslagen in een *dictionary*, welke wordt gerangschikt op (een vooraf vastgestelde) prioriteit. De kleur met de hoogste prioriteit wordt toegepast op de achtergrond van het hoofdvenster. Dit systeem zorgt ervoor dat kritieke informatie altijd duidelijk zichtbaar is, zelfs van een afstand, wat de gebruiksvriendelijkheid van de applicatie verhoogt. We zullen voorbeelden hiervan tegenkomen in de loop van het verslag.

De *MainWindow* past zich ook dynamisch aan wanneer een nieuwe gebruiker inlogt. Afhankelijk van de rol van de gebruiker worden de beschikbare tabs in de *QTabWidget* aangepast. Dit zorgt ervoor dat gebruikers alleen toegang hebben tot de functionaliteiten die relevant zijn voor hun verantwoordelijkheden. De mogelijke rollen zijn "*Regular*", "*Technician*" en "*God*". Een reguliere gebruiker heeft toegang tot de tabs "*Overview*", "*Account*" en "*Setup*", terwijl een technicus ook toegang heeft tot "*Configuration*" en een God ook nog tot "*Manage Users*". De titel van het venster wordt eveneens bijgewerkt om de naam en rol van de ingelogde gebruiker weer te geven, zodat direct voor iedereen duidelijk is wie ingelogd is. In **Error! Reference source not found.** is een voorbeeld hiervan te zien waarbij ikzelf ingelogd ben als God.



**Error! Reference source not found..** Venster titel met God gebruiker Emalisa ingelogd, daaronder zijn de verschillende tabs van de applicatie te zien.

Daarnaast integreert de *MainWindow* verschillende controllers die de communicatie tussen de gebruikersinterface en de backend beheren. Deze controllers zorgen voor taken zoals apparaat communicatie, profielbeheer en gebruikersauthenticatie. De specifieke functionaliteiten van deze controllers zullen doorheen het verslag worden besproken.

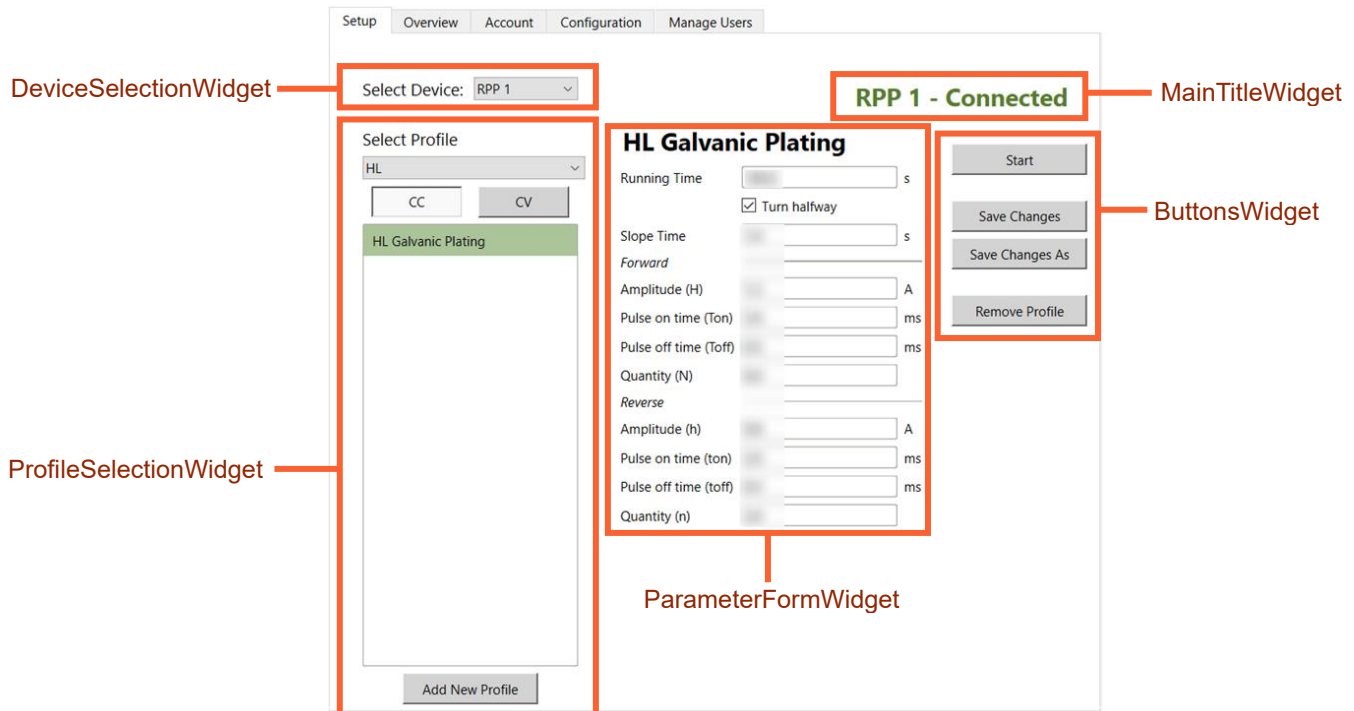
Als de applicatie gesloten wordt, zorgt de *MainWindow* ervoor dat de gebruiker een pop-up scherm gepresenteerd krijgt om te bevestigen dat de applicatie inderdaad mag sluiten. Mocht de gebruiker dit bevestigen terwijl er nog actief RPPs gemonitord worden door de applicatie, komt er een tweede pop-up



scherm, waar de gebruiker dit nogmaals moet bevestigen. Dit is gedaan omdat het monitoren van een RPP niet hervat kan worden als de applicatie eenmaal is afgezet.

## 4.2. Setup

De applicatie opent met de *Setup*-tab, welke is te zien in Figuur 4. Deze pagina biedt een overzichtelijke interface waarmee gebruikers een apparaat kunnen selecteren, een profiel kunnen kiezen en de parameters van dat profiel kunnen aanpassen.



Figuur 4. Setup-tab gezien met hoogste gebruikersrechten. Widget namen zijn aangegeven in de afbeelding.

Alle pagina's zijn opgebouwd als een *main view* (in dit geval *main\_setup.py*), waar verschillende widgets op worden ingeladen. Zoals Figuur 4 toont, heeft deze pagina vijf verschillende widgets; een voor het selecteren van een RPP (*DeviceSelectionWidget*), voor het selecteren van een profiel (*ProfileSelectionWidget*), voor de titel met de naam van de RPP met de connectiviteitsstatus (*MainTitleWidget*), voor het formulier met de profielparameters (*ParameterFormWidget*) en als laatste een widget met alle knoppen die zich aan de rechterkant bevinden (*ButtonsWidget*). Deze widgets communiceren niet direct met elkaar; dit zou al snel een moeilijk te onderhouden spaghetti van afhankelijkheden creëren. In plaats daarvan wordt via de centrale *main view* gecommuniceerd, gebruikmakend van het *Signal-Slot*-mechanisme. Bijvoorbeeld, wanneer een profiel wordt geselecteerd, wordt een *signal* uitgezonden vanaf de *ProfileSelectionWidget*. Deze *signal* wordt opgevangen door de *main view*, die vervolgens de *ParameterFormWidget* aanroept om zijn inhoud bij te werken. Dit ontwerp volgt het *Mediator Design* Patroon (Lavasani, 2024), waarbij de *main view* fungeert als de centrale coördinator tussen de widgets. Dit zorgt voor een duidelijke scheiding van verantwoordelijkheden en maakt de code makkelijker te onderhouden.

We hebben net kort alle verschillende widgets genoemd, laten we wat dieper ingaan op de *DeviceSelectionWidget* welke een drop-down menu biedt waarmee gebruikers een apparaat kunnen selecteren. Elke keer dat een apparaat wordt gekozen, wordt de verbinding met dat apparaat gecontroleerd via de controller. De verbindingstatus wordt vervolgens weergegeven in de titel van de pagina (zoals te zien is in Figuur 5), die wordt beheerd door de *MainTitleWidget*. Als het apparaat verbonden is, wordt de titel bijgewerkt met de tekst "Connected" en een groene tekstkleur (Figuur 5A). Als de verbinding ontbreekt, wordt "Disconnected" weergegeven met een rode tekstkleur (Figuur 5B).

A)

**RPP 1 - Connected**

B)

**RPP 1 - Disconnected**

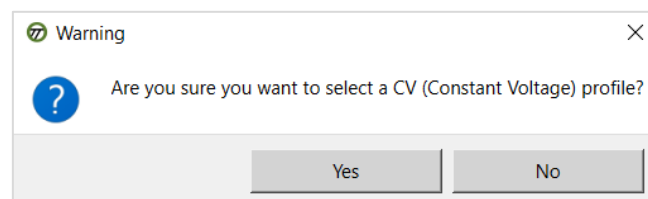
Figuur 5. *MainTitleWidget* voor A) een verbonden RPP B) een niet verbonden RPP.

Een volgend onderdeel van de pagina is de *ProfileSelectionWidget*, waarmee gebruikers een profiel kunnen kiezen. Wanneer de gebruiker een profiel selecteert, verschijnt de *ParameterFormWidget* met de gegevens voor dat profiel, met de relevante velden en validatie van de ingevoerde waarden. De validatie wordt uitgevoerd met behulp van reguliere expressies, zodat gebruikers alleen geldige waarden kunnen invoeren. Dit verhoogt de robuustheid van de applicatie en voorkomt fouten bij het opslaan van profielen.

Er zijn vier soorten profielen beschikbaar, elk met hun eigen parameters. Dit is eerder besproken in Hoofdstuk 2.1, maar het is belangrijk om te benadrukken dat, op het niveau van code, elk profieltype zijn eigen formulier heeft omdat ieder profiel andere parameters heeft om in te stellen. Deze formulieren zijn momenteel gehardcodeerd in de applicatie. Hoewel een dynamische aanpak idealer zou zijn, bleek dit te tijdrovend om te implementeren binnen het huidige project. Mocht in de toekomst de structuur van de profielparameters veranderen zonder dat de formulieren worden aangepast, dan zal er een foutmelding optreden bij het opslaan van het profiel, de ontwikkelaar zal dit dus doorhebben. Dit is een aandachtspunt dat in toekomstige iteraties kan worden aangepakt.

We zien dat een van de opties in het formulier is om halverwege te stoppen (*Turn halfway*). Dit is omdat, zoals besproken in Hoofdstuk 2.1, de PCBs in het bad soms halverwege omgedraaid moeten worden. Deze kwestie is opgelost door in de backend twee variabelen te hebben voor de *running time*; de *running\_time*, welke de waarde heeft die de gebruiker ingeeft, en de *actual\_running\_time*, welke de waarde heeft die naar de RPP wordt gestuurd. Als er niet halverwege gedraaid moet worden zijn deze hetzelfde, anders is de *actual\_running\_time* de helft van de *running\_time*. Het programma wordt dan dus normaal opgezet, maar stopt halverwege vanzelf. De gebruiker krijgt dan een visuele melding en kan na het omdraaien het programma hervatten. Dit voorkomt dat de twee zijdes van de PCBs ongelijk opgeplaat worden.

Een opvallend detail is de aanwezigheid van twee modi: *Constant Current (CC)* en *Constant Voltage (CV)*. In de praktijk wordt CV zelden gebruikt binnen het bedrijf, en het selecteren van deze modus gebeurt meestal per ongeluk. Om dit te voorkomen, verschijnt er een waarschuwingsvenster wanneer een gebruiker de CV-modus selecteert, zoals te zien in Figuur 6. Dit venster vraagt om bevestiging voordat de modus daadwerkelijk wordt gewijzigd, wat onbedoelde fouten helpt voorkomen.



Figuur 6. Waarschuwingsvenster selectie CV-modus

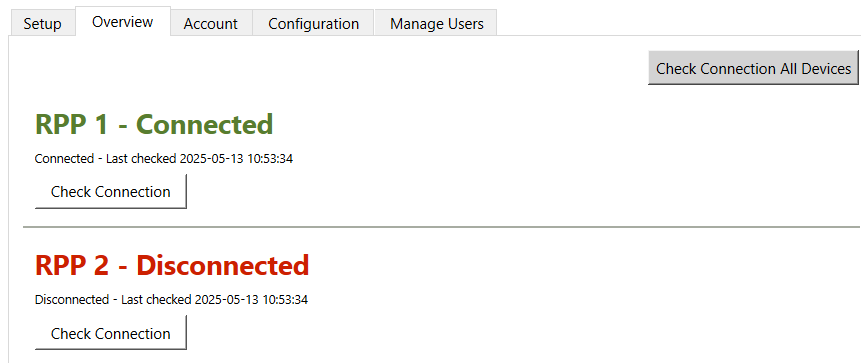
De *ButtonsWidget* bevat knoppen voor acties zoals het starten van een programma, het opslaan van wijzigingen in een profiel, en het verwijderen van een profiel. De startknop wordt automatisch uitgeschakeld wanneer het apparaat al actief is of wanneer het geen verbinding heeft. Dit voorkomt dat een nieuw programma wordt gestart terwijl er al een programma loopt. De gebruiker moet eerst het lopende programma stoppen voordat een nieuw programma kan worden opgezet.

Daarbij zijn we aangekomen bij het einde van de *Setup*-tab. We hebben gezien dat er op veel niveaus rekening is gehouden met gebruiksgemak, zowel voor de eindgebruiker, door middel van validaties en een duidelijke lay-outs, als voor toekomstige ontwikkelaars, door onder andere een duidelijke modulaire opbouw.

### 4.3. Overview

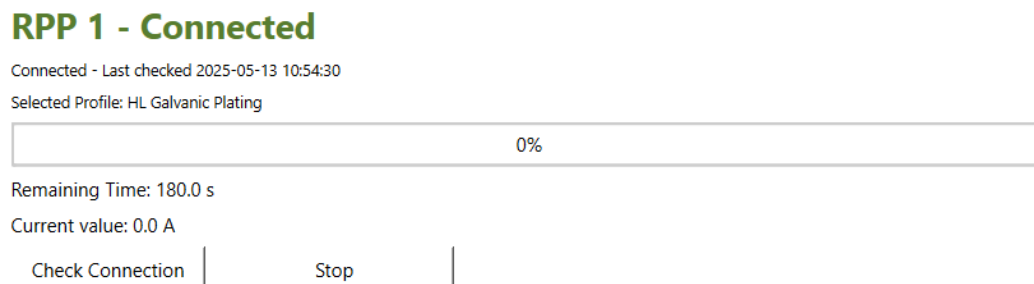
De *Overview*-tab biedt een overzicht van alle geregistreerde apparaten en hun huidige status. Het is een dynamische pagina waarin voor elk apparaat een aparte widget wordt geladen. Deze widgets worden bij het opstarten van de applicatie gegenereerd en toegevoegd aan een scrollbare container. Dit dynamische inladen van widgets maakt de pagina flexibel en schaalbaar, omdat het aantal apparaten eenvoudig kan worden uitgebreid zonder de structuur van de pagina aan te passen. Het is echter belangrijk om te vermelden dat wanneer een nieuw apparaat wordt toegevoegd, de applicatie opnieuw moet worden opgestart voordat de widget voor dat apparaat verschijnt. Idealiter zou dit in een toekomstige versie aangepast worden.

Elke widget toont informatie over de status van het apparaat. Als een apparaat niet actief is, wordt alleen de verbindingstatus weergegeven, dit gebeurt op dezelfde manier als de *MainTitleWidget* op de *Setup*-tab, zoals te zien is in Figuur 7. Wanneer een apparaat echter bezig is met een programma, worden ook de gemonitorde waarden getoond. Figuur 8 toont hier een voorbeeld van. We zien in de widget eerst de verbindingstatus en de datum en tijd van de laatste uitgevoerde verbindingcheck, met daaronder het geselecteerde profiel voor het huidige programma. Daaronder zien we een statusbalk waarmee de gebruiker een duidelijk visueel beeld krijgt van hoe lang het programma al op staat. Daaronder zien we de resterende tijd voor het programma in seconden en de output waarde in Ampère of Voltage, afhankelijk van de modus van het geselecteerde profiel. Deze waarden worden in realtime bijgewerkt via signalen die door de backend worden verzonden. Hoe deze monitoring precies werkt, wordt besproken in Hoofdstuk 5.4.

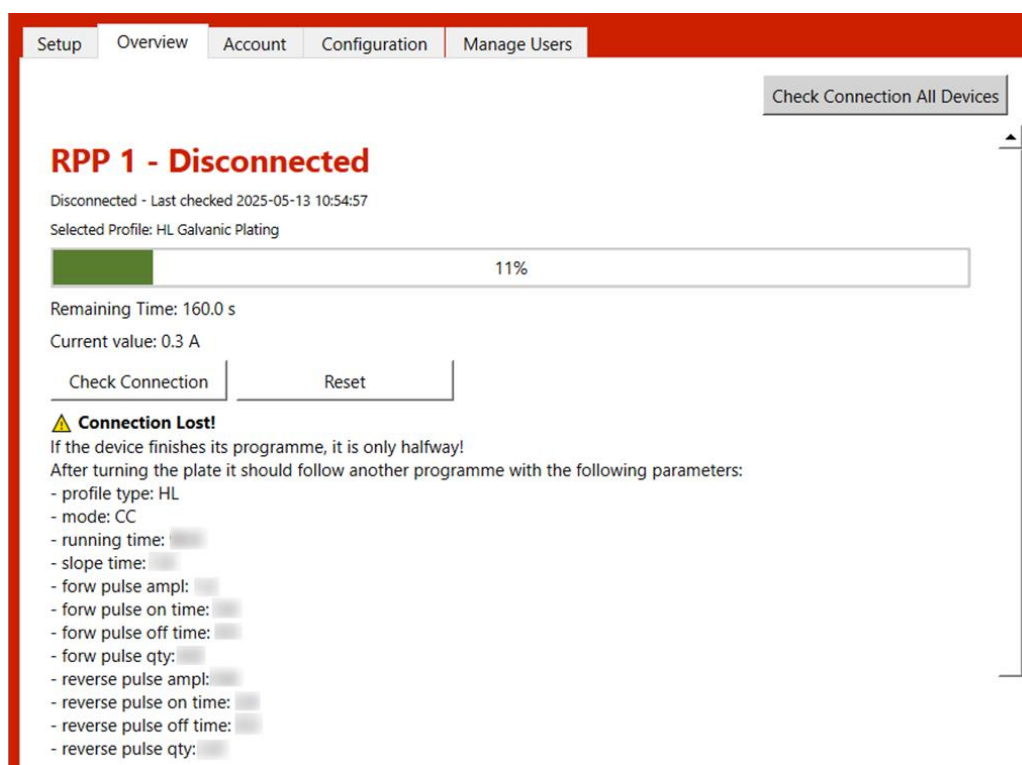


Figuur 7. Overview-tab met een verbonden niet-actieve, RPP 1, en een niet verbonden, RPP 2, RPP geregistreerd.

De applicatie controleert ook of de gemonitorde outputwaarden binnen de verwachte grenzen vallen. Als er afwijkende waarden worden gedetecteerd, verschijnt er een foutmelding in de widget, samen met een rode achtergrondkleur. Dit waarschuwt de gebruiker dat er mogelijk iets mis is met het apparaat of het programma. Een vergelijkbare foutmelding verschijnt wanneer de verbinding met een apparaat verloren gaat. In dat geval wordt een waarschuwing weergegeven met alle informatie die de gebruiker nodig heeft om het programma zo succesvol mogelijk af te ronden. Als het programma bijvoorbeeld halverwege zal stoppen, wordt aangegeven met welke instellingen het programma nadien hervat moet worden, een voorbeeld van deze foutmelding is te zien in Figuur 9.

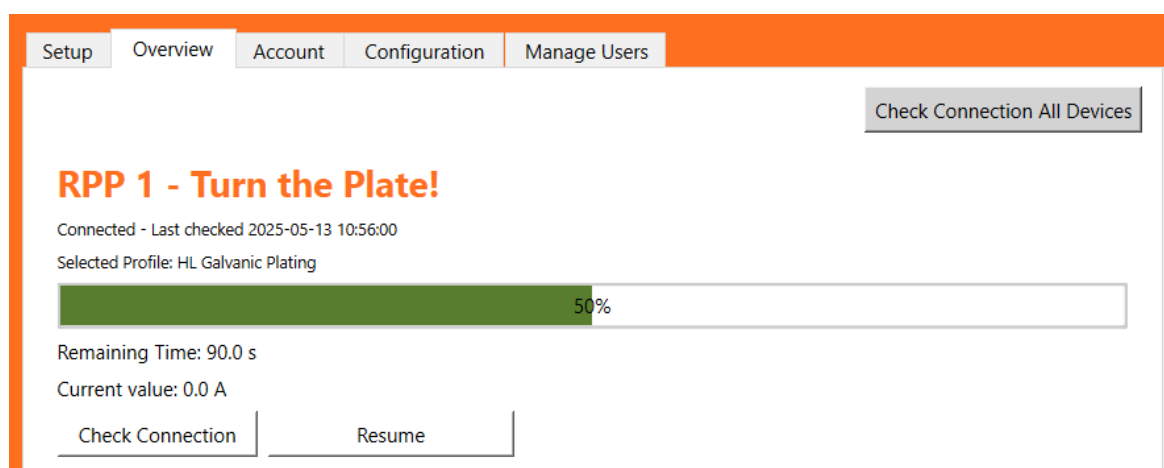


Figuur 8. Overview voor een actieve RPP met monitor waarden.

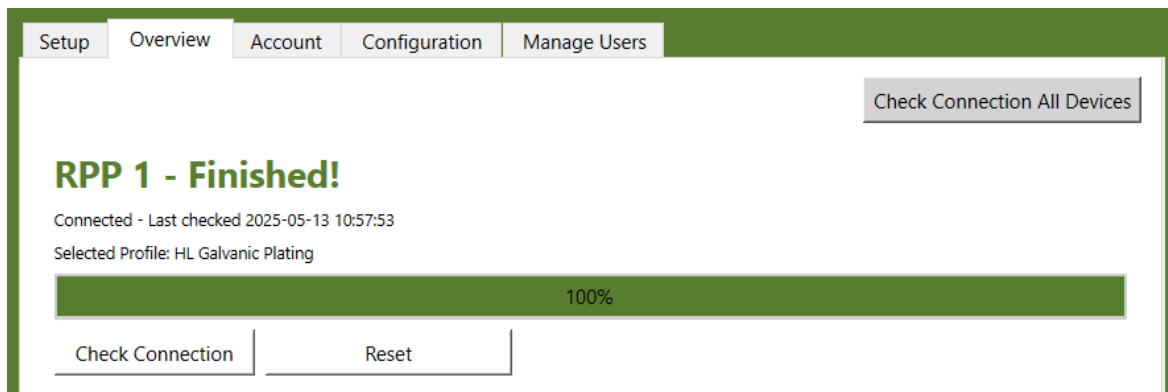


*Figuur 9. Voorbeeld foutmelding: verbinding met RPP is verloren gegaan tijdens een programma waarbij halverwege gestopt zal worden. Foutmelding geeft aan met welke parameters de RPP ingesteld moet worden voor het vervolgprogramma.*

Wanneer een programma halverwege moet worden gestopt om de plaat om te draaien, wordt de gebruiker hiervan op de hoogte gesteld via een oranje achtergrond en een aangepaste titel in de *Overview*-tab, zie Figuur 10. De gebruiker kan vervolgens met de *Resume* knop het programma hervatten. Als het programma volledig is afgerond, wordt dit aangegeven met een groene melding en een succesbericht in de titel, zoals te zien is in Figuur 11. De gebruiker kan de widget en de achtergrondkleur resetten via de *Reset* knop nadat het programma is voltooid, zodat de widget weer klaar is voor een nieuw programma.



*Figuur 10. Overview-tab wanneer RPP 1 gepauzeerd is halverwege een programma. De gebruiker wordt gewaarschuwd via een oranje achtergrond en een melding in de titel en kan het programma hervatten via de Resume knop.*



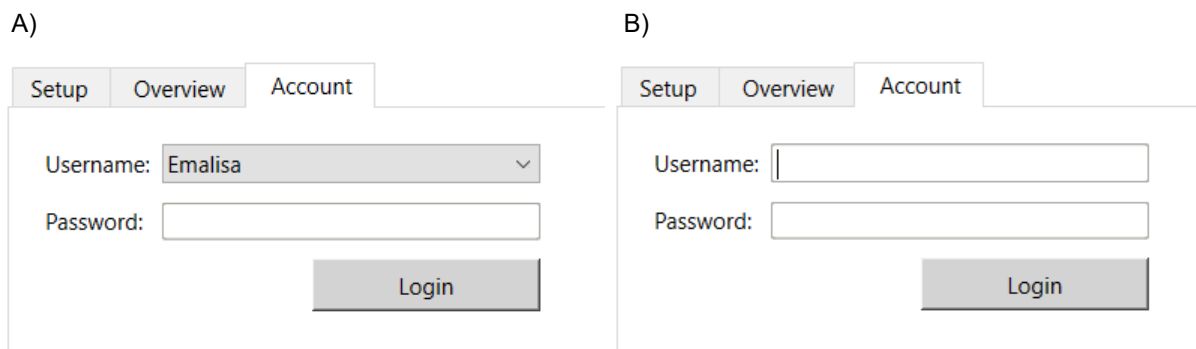
*Figuur 11. Overview-tab wanneer RPP 1 klaar is met een programma. De gebruiker wordt gewaarschuwd via een groene achtergrond en een melding in de titel. De gebruiker kan de meldingen resetten met de Reset knop.*

Een programma kan ook handmatig worden gestopt door de gebruiker. In dat geval kan het programma niet meer worden hervat. Dit is een bewuste ontwerpkeuze om te voorkomen dat een programma in een onvoorspelbare staat wordt hervat. Als de verbinding met een RPP verloren gaat en vervolgens wordt hersteld, kan het monitoren van het programma echter wel worden hervat, op voorwaarde dat het programma niet handmatig is gestopt in de applicatie. Wanneer een RPP bezig is met een programma wanneer de applicatie opstart wordt dit herkend en kan de gebruiker geen nieuw programma opzetten. Het monitoren van dit al lopende programma is naar mijn idee ook mogelijk, maar ik ben hier zelf niet aan toegekomen en staat dus nog op het lijstje met potentiële toekomstige uitbreidingen.

De *Overview*-pagina biedt een intuïtieve en visueel aantrekkelijke interface waarmee gebruikers de status van hun apparaten kunnen volgen en beheren. Het gebruik van dynamische widgets, signalen en duidelijke visuele feedback zorgt ervoor dat gebruikers altijd op de hoogte zijn van de status van hun apparaten en snel kunnen reageren op eventuele problemen.

## 4.4. Login/Account

De *Account*-tab van de applicatie biedt een eenvoudige en gebruiksvriendelijke interface waarmee gebruikers kunnen inloggen en is te zien in Figuur 13A. Een opvallend kenmerk van deze pagina is het gebruik van een *drop down* menu voor het selecteren van de gebruikersnaam. Hoewel dit een ongebruikelijke keuze is in grotere systemen, is het in de context van dit bedrijf, met een beperkte gebruikersbasis, een praktische oplossing; het gebruik van een *drop down* menu vermindert de kans op typfouten en versnelt het inlogproces, wat het gebruiksgemak aanzienlijk verhoogt.

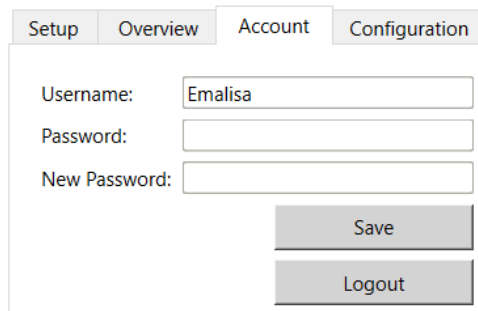


*Figuur 12. A) Regulier inlogscherm B) Inlogscherm voor recovery user*

Een interessant aspect van de inlogpagina is de aanwezigheid van een *recovery user*. Deze gebruiker kan worden gebruikt bij de initiële installatie of als een vangnet wanneer het wachtwoord van de God-gebruiker (de beheerder) verloren is gegaan. Omdat de applicatie geen toegang heeft tot het internet, is het niet mogelijk om een herstel-wachtwoord aan te vragen via een externe service. In plaats daarvan kan het

*recovery user* inlogscher (zie Figuur 13B) worden geactiveerd via een geheime sneltoetscombinatie, waar ingelogd kan worden met de gegevens die te vinden zijn in de technische gebruikershandleiding. De *recovery user* heeft volledige toegang tot de applicatie, maar de instellingen van deze gebruiker kunnen niet worden gewijzigd, wat voorkomt dat toegang tot de recovery user per ongeluk verloren gaat.

De inlogpagina maakt gebruik van een *QStackedWidget* (The Qt Company Ltd., 2025) om dynamisch te schakelen tussen de inlog interface en de account details interface. Wanneer een gebruiker succesvol inlogt, wordt de inlogwidget vervangen door een formulier waarmee de gebruiker zijn of haar wachtwoord of gebruikersnaam kan wijzigen of kan uitloggen, zie Figuur 13. Dit formulier is echter gedeactiveerd wanneer de recovery user is ingelogd, omdat de instellingen van deze gebruiker niet mogen worden aangepast.

The screenshot shows a user interface with four tabs: 'Setup', 'Overview', 'Account', and 'Configuration'. The 'Account' tab is selected. Below the tabs, there are three input fields: 'Username:' with the text 'Emalisa', 'Password:', and 'New Password:'. To the right of these fields are two buttons: 'Save' and 'Logout'.

Figuur 13. Account-tab, waar gebruiker kan uitloggen of wachtwoord kan aanpassen.

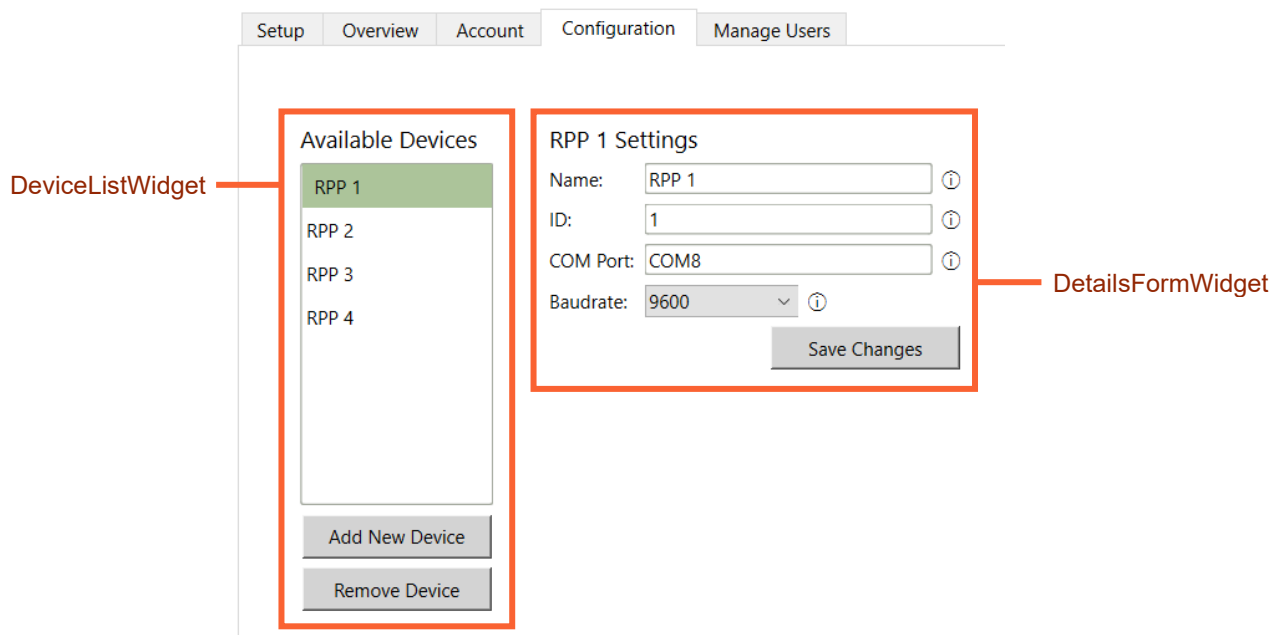
Een ander belangrijk detail is dat het wachtwoordveld is beveiligd tegen kopiëren. Het contextmenu is uitgeschakeld, zodat gebruikers het wachtwoord niet kunnen kopiëren naar het klembord. Dit verhoogt de beveiliging van de applicatie en voorkomt dat wachtwoorden per ongeluk worden gedeeld of opgeslagen op onveilige locaties.

De inlogpagina is, zoals de rest van de applicatie, ontworpen met een focus op gebruiksvriendelijkheid. Het gebruik van een *drop down* menu voor gebruikersnamen, de *fail-safe recovery user*, en de beveiligingsmaatregelen in het wachtwoordveld zijn allemaal voorbeelden van hoe de applicatie is afgestemd op de specifieke behoeften en beperkingen van het bedrijf. Tegelijkertijd biedt de pagina voldoende flexibiliteit om gebruikers in staat te stellen hun accountinstellingen eenvoudig te beheren na het inloggen.

## 4.5. Configuratie

De configuratiepagina is een eenvoudige maar essentiële pagina binnen de applicatie, ontworpen om beheerders en technici in staat te stellen de instellingen van apparaten te beheren. Omdat deze pagina gevoelige configuraties bevat, is deze niet toegankelijk voor reguliere gebruikers.

De pagina bestaat uit widgets: een lijst met alle geregistreerde apparaten (*DeviceListWidget*) en een formulier waarmee de details van een geselecteerd apparaat kunnen worden aangepast (*DetailsFormWidget*), zie Figuur 14. De *DeviceListWidget* toont een overzicht van alle apparaten in de database. Gebruikers kunnen een apparaat selecteren, een nieuw apparaat toevoegen of een bestaand apparaat verwijderen.



Figuur 14. Configuratie pagina. Widgetnamen zijn aangegeven in de afbeelding.

De *DetailsFormWidget* biedt een intuïtieve interface waarmee gebruikers de naam, ID, COM-poort en baudrate van een apparaat kunnen aanpassen. Het formulier bevat ingebouwde validatie om ervoor te zorgen dat alleen geldige gegevens worden ingevoerd. Zo wordt bijvoorbeeld de COM-poort gecontroleerd op het juiste formaat met behulp van reguliere expressies, en wordt de baudrate beperkt tot een vooraf gedefinieerde lijst van geldige waarden. Als de ingevoerde gegevens niet aan de vereisten voldoen, worden foutmeldingen weergegeven onder de betreffende velden. Deze foutmeldingen worden dynamisch ingesteld via signalen die vanuit de backend worden verzonden na validatie.

De ID, COM Port en baudrate kunnen niet zomaar gekozen worden; deze moeten overeenkomen met de instellingen op de RPP en op de computer zelf. Hoe dit werkt, staat uitgelegd in de technische gebruikershandleiding, maar de stappen staan ook uitgelegd in *tooltips* in het formulier zelf. De gebruiker moet zijn muis over de *i* symbooltjes houden om deze stappen te kunnen zien.

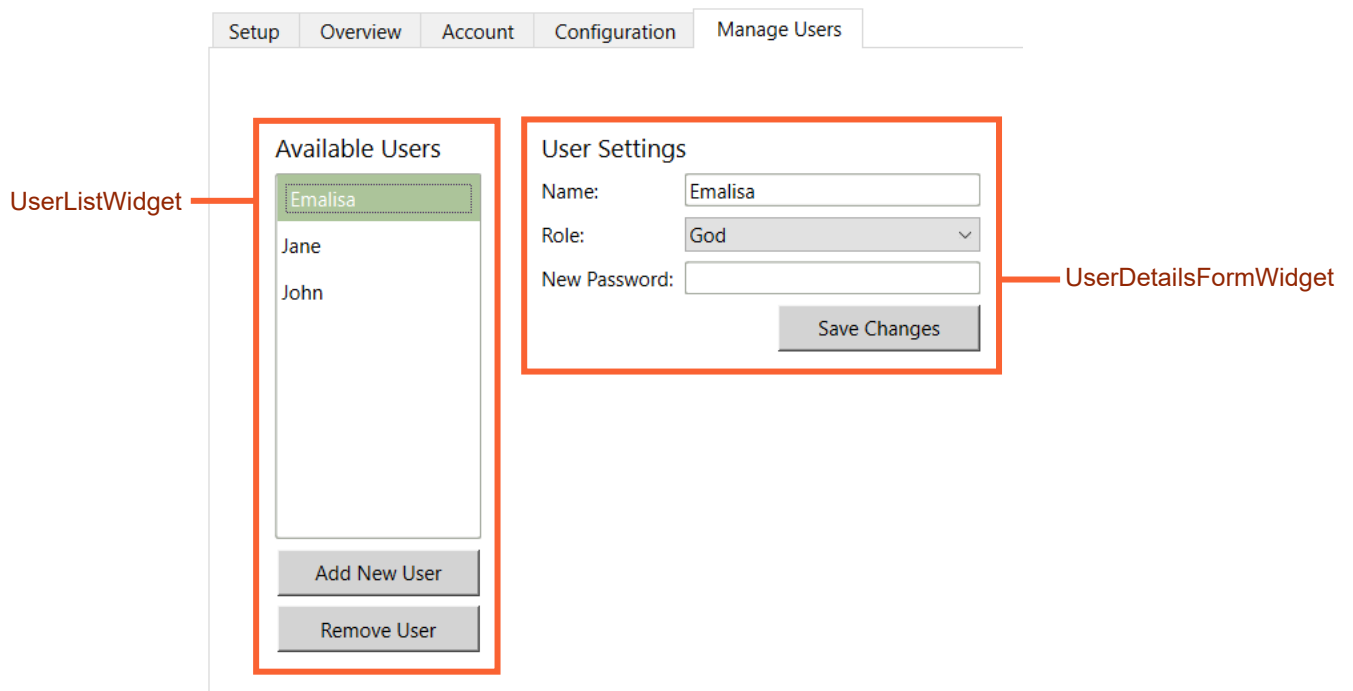
De configuratiepagina, zoals iedere andere pagina, volgt het *Single Responsibility Principle (SRP)* (Moeller, 2024), waarbij elke widget een duidelijke en afgebakende verantwoordelijkheid heeft. De *DeviceListWidget* is verantwoordelijk voor een overzicht van alle RPPs in de database, terwijl de *DetailsFormWidget* zich richt op het aanpassen van de details van een specifiek apparaat. Deze scheiding van verantwoordelijkheden maakt de code overzichtelijk en eenvoudig te onderhouden.

Hoewel de configuratiepagina relatief eenvoudig is, speelt deze een cruciale rol in het beheer van de apparaten. De ingebouwde validatie en duidelijke foutmeldingen zorgen ervoor dat gebruikers met de juiste rechten de instellingen van apparaten veilig en efficiënt kunnen beheren. Dit draagt bij aan de betrouwbaarheid en gebruiksvriendelijkheid van de applicatie.

## 4.6. Gebruikersbeheer

De gebruikersbeheerpagina biedt een overzichtelijke interface waarmee beheerders gebruikers kunnen beheren, i.e. gebruikers aanmaken en verwijderen en gegevens van bestaande gebruikers aanpassen. De pagina bestaat ook uit twee widgets: een lijst met alle geregistreerde gebruikers (*UserListWidget*) en een formulier waarmee de details van een geselecteerde gebruiker kunnen worden aangepast (*UserDetailsFormWidget*), zie Figuur 15.





Figuur 15. Manage Users pagina. Widgetnamen zijn aangegeven in de afbeelding.

De *UserListWidget* toont een lijst met alle gebruikers die in het systeem zijn geregistreerd. Gebruikers kunnen eenvoudig een gebruiker selecteren, een nieuwe gebruiker toevoegen of een bestaande gebruiker verwijderen. Wanneer een gebruiker wordt geselecteerd, wordt een *signal* uitgezonden dat via de *main view* (*main\_manage\_users.py*) naar de *UserDetailsFormWidget* wordt gestuurd. Dit formulier wordt vervolgens bijgewerkt met de gegevens van de geselecteerde gebruiker, zodat deze kan worden aangepast. De lijst wordt alfabetisch gesorteerd om het zoeken naar een specifieke gebruiker te vergemakkelijken.

De *UserDetailsFormWidget* biedt een formulier waarmee beheerders de gebruikersnaam, rol en het wachtwoord van een gebruiker kunnen aanpassen. Het formulier gebruikt de *UserController* om ervoor te zorgen dat alleen geldige gegevens worden ingevoerd. Zo wordt bijvoorbeeld gecontroleerd of de gebruikersnaam uniek is en niet leeg. Als de ingevoerde gegevens niet geldig zijn, wordt een foutmelding weergegeven onder het formulier. Deze foutmeldingen worden dynamisch ingesteld via signalen die vanuit de backend worden verzonden.

Een belangrijk detail is de manier waarop nieuwe gebruikers worden aangemaakt. Wanneer een nieuwe gebruiker wordt toegevoegd, genereert de applicatie automatisch een tijdelijk wachtwoord. Dit wachtwoord wordt direct naar het klembord van de beheerder gekopieerd en weergegeven in een *toastmelding*. De *toast* bevat een duidelijke instructie om het wachtwoord te delen met de gebruiker en het klembord daarna leeg te maken.

Alle pagina's maken gebruik van *toastmeldingen* om gebruikers op de hoogte te stellen van belangrijke gebeurtenissen. Bijvoorbeeld, wanneer een gebruiker succesvol is toegevoegd, wordt een succesmelding weergegeven. Als er een fout optreedt, zoals een ongeldig wachtwoord of een ontbrekende gebruikersnaam, wordt een foutmelding weergegeven. Deze meldingen verbeteren de gebruiksvriendelijkheid van de applicatie door directe feedback te geven op de acties van de beheerder. Er is gebruik gemaakt van de *toast library* voor *pyqt* van Niklas Henning (Henning, 2025).

De gebruikersbeheerpagina stelt de gebruiker in staat om gemakkelijk nieuwe en bestaande gebruikers te beheren. De ingebouwde validatie, duidelijke foutmeldingen en gebruiksvriendelijke *toastmeldingen* zorgen voor efficiënt en veilig gebruik. Dit draagt bij aan de betrouwbaarheid en gebruiksvriendelijkheid van de applicatie.

## 5. Applicatie Backend

In dit hoofdstuk wordt de backend van de applicatie behandeld. Dit is waar de zakelijke logica is geprogrammeerd. Het is dan ook een vrij technisch hoofdstuk geworden en wellicht niet toegankelijk voor minder technische lezers.

We gaan eerst kijken naar de centrale signaal zender, dit is een element dat essentieel is voor de communicatie tussen de backend en de frontend op het gebied van foutmeldingen. Daarna gaan we kijken naar de code omtrent de profielen. Profielen (profiles), zijn een van de drie modellen die we hebben, samen met Apparaten (devices) en Gebruikers (users). Ieder van deze modellen heeft in dit hoofdstuk een eigen paragraaf gekregen, waarin niet alleen het model wordt besproken, maar ook bijbehorende *controllers*, *managers* en *services*. Bij de apparaten zit ook het hele element van Modbus communicatie, door de complexiteit heeft dit ook zijn eigen paragraaf gekregen. In Bijlage B kan een klassendiagram gevonden worden, welke een overzicht geeft van de structuur van de backend. Voor visueel ingestelde lezers kan dit diagram helpen om dit hoofdstuk makkelijker te volgen.

### 5.1. Centrale Signaal Zender

We zullen beginnen met een belangrijk element om de communicatie tussen de backend en de gebruikersinterface mogelijk maken: de *SignalEmitter*. Dit onderdeel speelt een cruciale rol in het ontwerp van de applicatie en is ontstaan uit een uitdaging die voortkwam uit het omgaan met foutmeldingen.

Aanvankelijk was het idee om een duidelijke scheiding te maken tussen verschillende soorten foutmeldingen op basis van de oorsprong. Fouten die voortkwamen uit de gebruikersinterface zouden worden weergegeven in *error labels* (als een GUI element), fouten uit de controllers zouden worden afgehandeld via *toastmeldingen* (nog altijd direct zichtbaar voor de gebruiker), en fouten uit de modellen zouden worden gelogd (de gebruiker moet moeite doen om ze te vinden). Dit bleek in de praktijk een probleem te veroorzaken: sommige fouten uit de backend, zoals problemen bij het uitlezen van registers, waren te belangrijk om alleen in de logs te worden opgenomen. Deze fouten moesten direct aan de gebruiker worden gecommuniceerd, maar tegelijkertijd moest natuurlijk voorkomen worden dat de backend afhankelijk werd van de gebruikersinterface.

Om dit probleem op te lossen, werd de *SignalEmitter* geïntroduceerd. Dit is een centrale klasse in de backend die signalen uitzendt naar de frontend. Deze signalen kunnen in de gebruikersinterface worden opgevangen en omgezet in visuele feedback, zoals error labels of gekleurde achtergronden. Het belangrijkste voordeel van deze aanpak is dat de backend volledig onafhankelijk blijft van de gebruikersinterface, maar nog wel alle niveaus van de applicatie kan bereiken. De *SignalEmitter* fungeert als een abstractie laag tussen de backend en de frontend. Als er in de toekomst een nieuw frontend-framework wordt geïntroduceerd, hoeft alleen de *SignalEmitter* te worden aangepast, terwijl de rest van de backend onaangetast blijft. Dit volgt het *Dependency Inversion Principle (DIP)* (Osbourne, 2024), waarbij hoog-niveau modules (zoals de backend) niet afhankelijk zijn van laag-niveau modules (zoals de gebruikersinterface).

Een andere benadering die werd overwogen, was om de controllers verantwoordelijk te maken voor het afhandelen van foutmeldingen uit de backend, deze zijn toch al afhankelijk van de frontend. De backend zou in dat geval de controller aanroepen, die vervolgens signalen naar de frontend zou sturen. Hoewel dit technisch mogelijk was, introduceerde het een aantal problemen. Ten eerste kregen de controllers hierdoor te veel verantwoordelijkheden, wat in strijd is met het *Single Responsibility Principle (SRP)* (Moeller, 2024). Ten tweede maakte deze aanpak het pad dat een foutmelding aflegde moeilijk te volgen, wat de onderhoudbaarheid en uitbreidbaarheid van de applicatie bemoeilijkte. Door de *SignalEmitter* rechtstreeks in de backend te plaatsen, werd dit probleem opgelost en bleef de verantwoordelijkheid van de controllers beperkt tot hun primaire taak: het coördineren van de interactie tussen de managers en de GUI.

De *SignalEmitter* volgt het *Singleton Design Pattern* (Geeks for Geeks, 2024), wat betekent dat er slechts één instantie van deze klasse bestaat gedurende de levensduur van de applicatie. Dit is belangrijk omdat signalen vanuit verschillende delen van de backend worden verzonden en er een centrale plek nodig is om

deze signalen te beheren. Door het *singleton*-patroon te gebruiken, wordt ervoor gezorgd dat alle signalen consistent worden afgehandeld en dat er geen conflicten ontstaan door meerdere instanties van de *SignalEmitter*.

De *SignalEmitter* biedt een flexibele en schaalbare oplossing voor het communiceren van belangrijke gebeurtenissen en fouten vanuit de backend naar de frontend. Door de backend onafhankelijk te houden van de GUI en gebruik te maken van duidelijke ontwerppatronen, zoals het *Singleton*-patroon en het *Dependency Inversion Principle*, is een robuuste en toekomstbestendige oplossing gecreëerd.

## 5.2. Profielen

De functionaliteit rond de profielen vormt een belangrijk onderdeel van de applicatie en is ontworpen met flexibiliteit en uitbreidbaarheid in gedachten. Omdat er vier verschillende profielen zijn met ieder hun eigen parameters zijn er ook vier profiel klassen: *Const*, *Chop*, *SW*, en *HL*. Deze vier erven allemaal van een abstracte *Profile*-klasse. Het gebruik van een abstracte klasse zorgt ervoor dat elk profieltype zijn eigen specifieke implementaties kan hebben, terwijl de gemeenschappelijke functionaliteit wordt gedeeld. Een belangrijk aspect van deze klasse is het dynamisch omgaan met attributen. In Figuur 16 is een fragment uit de *Profile*-klasse te zien. Wat we hier zien is dat er bepaalde parameters zijn, zoals *mode* en *name*, die alle profielen hebben, maar dat er ook ruimte is om een willekeurige hoeveelheid extra parameters mee te geven, waarin de kinderklassen vrij zijn om te kiezen, welke terechtkomen in *\_params*. Als vervolgens *\_create\_dynamic\_properties* wordt aangeroepen (wat in de constructors van alle kinderklassen gebeurt) wordt alles in *\_params* omgezet in een *property*, volledig met *getters* en *setters*. Hierdoor kunnen profiel specifieke parameters eenvoudig worden toegevoegd zonder dat de structuur van de *Profile*-klasse hoeft te worden aangepast. Dit ontwerp volgt het *Open-Closed Principle (OCP)* (Thorben, 2018), omdat nieuwe profieltypes kunnen worden toegevoegd zonder bestaande code te wijzigen.

```
class Profile(ABC):
    def __init__(self, mode: Mode, name: str, running_time: int, turn_halfway:
bool, **kwargs):
        self._params = kwargs

    def _create_dynamic_properties(self):
        '''Dynamically create properties for each attribute passed through
kwargs'''
        for attr in self._params:
            setattr(self.__class__, attr, self._create_property(attr))

    @staticmethod
    def _create_property(attr):
        '''Dynamically create getters and setters for a given attribute.'''
        def getter(self):
            return self._params[attr]

        def setter(self, value):
            self._params[attr] = value

        return property(getter, setter)
```

Figuur 16. Dynamische attributen Profile klasse

Een andere belangrijke eigenschap van de *Profile*-klasse is de abstracte methode *apply\_to\_device\_modbus*. Deze methode impliceert dat iedere profiel klasse een methode moet hebben, waarin wordt gedefinieerd naar welke registers geschreven moet worden om dat specifieke profiel in te stellen. Hoewel het toevoegen van zo'n specifieke methode niet ideaal klinkt vanuit het oogpunt van modulariteit (de profiel klassen zouden zich niet moeten bezighouden met Modbus communicatie), biedt het een duidelijke structuur. Elk profieltype implementeert zijn eigen versie van deze methode, wat voorkomt dat er in de *DeviceCommunicationController*, welke als primaire taak heeft om de Modbus verbinding te faciliteren, een lange reeks *if-else-statements* nodig is, welke aangepast zou moeten worden iedere keer dat er een nieuw profiel wordt toegevoegd. Bovendien zorgt dit ontwerp ervoor dat nieuwe profieltypes automatisch worden gedwongen om hun eigen implementatie van deze methode te definiëren, wat bijdraagt aan de consistentie en onderhoudbaarheid van de code. Als er in de toekomst een nieuwe communicatiemethode wordt geïntroduceerd, kan een nieuwe abstracte methode worden toegevoegd aan de *Profile*-klasse. Hierdoor worden alle bestaande profieltypes automatisch verplicht om deze nieuwe communicatiemethode te implementeren.

Verder is er een *\_\_eq\_\_*-methode toegevoegd om het vergelijken van profielobjecten te vereenvoudigen, met name in het kader van *unit tests*, welke veel zijn gebruikt bij het ontwikkelen van de backend. Deze methode is in staat zowel de gemeenschappelijke attributen als de profiel specifieke parameters te vergelijken. Daarnaast biedt een *to\_dict*-methode een gestandaardiseerde manier om profielobjecten om te zetten naar een JSON-representatie, wat essentieel is voor opslag.

We hebben het al een paar keer gehad over de controllers. De *ProfileController* fungeert als de schakel tussen de gebruikersinterface en de backend op het gebied van profielen. De controller roept methodes aan in de *ProfileManager*, die verantwoordelijk is voor het beheren van de profielobjecten. Er zijn methodes voor alles wat de GUI nodig kan hebben, van het opslaan van profielen, tot het ophalen van de metadata van alle profielen. De profielen worden opgeslagen in JSON-bestanden, wat een eenvoudige en effectieve oplossing is voor de huidige schaal van de applicatie. Een mogelijke uitbreiding in de toekomst zou het implementeren van een back-up-mechanisme zijn om de profielgegevens te beschermen tegen dataverlies.

Een belangrijk kenmerk van de profielopslag is het gebruik van *lazy loading* (Geeks for Geeks, 2025). Profielobjecten worden pas aangemaakt wanneer ze daadwerkelijk nodig zijn. Dit bespaart geheugen en verhoogt de efficiëntie van de applicatie, vooral wanneer er veel profielen zijn opgeslagen. De *ProfileManager* maakt gebruik van een cache om geladen profielen op te slaan, zodat ze niet opnieuw hoeven te worden geladen bij herhaald gebruik.

De profielen worden geïdentificeerd via een UUID (*Universally Unique Identifier*). Dit ontwerp is gekozen om een eventuele uitbreiding naar een relationele database in de toekomst te vergemakkelijken. Door gebruik te maken van UUID's kunnen profielen eenvoudig worden gekoppeld aan andere entiteiten in een database, zoals apparaten of gebruikers.

Een ander belangrijk onderdeel van de profielenfunctionaliteit is de *ProfileFactory*, welke te zien is in Figuur 17. Deze *factory* krijgt het JSON-bestand binnen, kijkt naar de *\_type* variabele om te zien van welk type het profiel is. Vervolgens maakt het een profiel object aan volgens de *from\_dict* methode van de juiste profiel klasse. Als er in de toekomst nieuwe profieltypes worden toegevoegd, hoeft alleen de *factory* te worden bijgewerkt om deze nieuwe types te ondersteunen bij het inladen vanuit JSON-bestanden.

```

from rppcs.models.profiles.const_profile import Const
from rppcs.models.profiles.hl_profile import HL
from rppcs.models.profiles.sw_profile import SW
from rppcs.models.profiles.chop_profile import Chop

profile_classes = {
    "Const": Const,
    "HL": HL,
    "SW": SW,
    "Chop": Chop
}

def create_profile_from_dict(data):
    '''Factory function to instantiate the correct Profile subclass'''
    # determine the profile type that needs to be created
    profile_type = data.get("_type")
    profile_cls = profile_classes.get(profile_type)

    if profile_cls:
        return profile_cls.from_dict(data)
    raise ValueError(f"Unknown profile type: {profile_type}")

```

Figuur 17. Code profile factory

Samenvattend biedt de profielenfunctionaliteit een solide basis voor het beheren en toepassen van verschillende profieltypes. Het gebruik van dynamische attributen, *lazy loading*, UUID's, en verschillende ontwerp patronen zorgt voor een flexibele en toekomstbestendige oplossing. Tegelijkertijd biedt de huidige implementatie voldoende ruimte voor verdere optimalisatie en uitbreiding.

### 5.3. Apparaten

De functionaliteit rond de apparaten in de applicatie is ontworpen met een focus op flexibiliteit, validatie en onderhoudbaarheid. De *Device*-klasse vormt de kern van deze functionaliteit en biedt een representatie van een fysiek apparaat, inclusief eigenschappen zoals ID, naam, baudrate en COM-poort. Daarnaast bevat de klasse methoden voor het beheren van de status van het apparaat en het communiceren met de bijbehorende communicatiecontroller.

Een belangrijk aspect van de *Device*-klasse is een *callback* naar de communicatieservice, welke de Modbus-verbinding onderhoudt. Wanneer een attribuut van een apparaat wordt aangepast, zoals de baudrate of de COM-poort, wordt de *\_notify\_observers*-methode aangeroepen. Deze methode informeert alle geregistreerde *observers*, i.e. de communicatieservice, over de wijziging. Dit is essentieel omdat de communicatieservice zichzelf moet aanpassen aan de nieuwe instellingen van het apparaat. Dit ontwerp volgt het *Observer Design Pattern* (Geeks for Geeks, 2025), waarbij de communicatieservice als *observer* fungeert en automatisch wordt bijgewerkt wanneer er wijzigingen plaatsvinden in het apparaat.

De *setters* in de *Device*-klasse bevatten ingebouwde validatiechecks. Zo controleert de setter voor de baudrate of de waarde behoort tot een lijst van ondersteunde snelheden, en wordt de COM-poort gecontroleerd op de juiste syntax (bijvoorbeeld "COM4"). Deze validatie op het niveau van de klasse voorkomt dat ongeldige gegevens worden opgeslagen in het apparaat object en verhoogt de betrouwbaarheid van de applicatie.

Een *set\_attr*-methode maakt het mogelijk om meerdere attributen van een apparaat in één keer bij te werken. Dit is vooral handig bij het laden van apparaat gegevens uit een JSON-bestand of bij het toepassen van wijzigingen vanuit de gebruikersinterface. De methode controleert of de opgegeven attributen geldig zijn en negeert eventuele ongeldige sleutels, wat bijdraagt aan de robuustheid van de applicatie.

De *DeviceController* fungeert als de schakel tussen de gebruikersinterface en de backend voor apparaat beheer. Net als de *ProfileController* is het voornamelijk een doorgeefluik dat methoden in de *DeviceManager* aanroept. De controller bevat echter ook validatiechecks, bijvoorbeeld om te controleren of een apparaat naam uniek is of dat een ID binnen het toegestane bereik valt. Dit betekent dat er validatie plaatsvindt op meerdere niveaus: in de gebruikersinterface, in de controller en in de Device-klasse zelf. Deze gelaagde aanpak zorgt ervoor dat fouten vroegtijdig worden opgevangen en dat de applicatie robuust blijft, zelfs als een van de lagen een fout over het hoofd ziet. Dit draagt ook bij aan de modulariteit van de applicatie; de verschillende lagen zijn niet afhankelijk van elkaar voor het opvangen van fouten.

De *DeviceController* bevat een methode om een unieke naam te genereren voor nieuwe apparaten zonder naam. Dit gebeurt door te controleren of de standaardnaam "New Device" al bestaat en, indien nodig, een nummer toe te voegen (bijvoorbeeld "New Device 1", "New Device 2", enzovoort). Dit voorkomt conflicten en maakt het eenvoudig om nieuwe apparaten toe te voegen.

In tegenstelling tot de profielen, die pas worden geladen wanneer ze nodig zijn (*lazy loading*), worden alle apparaat objecten bij het opstarten van de applicatie aangemaakt. Dit is een bewuste keuze, omdat de status van alle apparaten direct beschikbaar moet zijn in de gebruikersinterface. Aangezien het aantal apparaten in de praktijk beperkt is, heeft dit geen grote impact op de prestaties van de applicatie.

## 5.4. Communicatie

De verbinding tussen de applicatie en de apparaten wordt beheerd door de *ModbusService* en de *DeviceCommunicationController*, die samen een robuuste en flexibele infrastructuur vormen voor communicatie en monitoring. De *ModbusService* is specifiek ontworpen voor Modbus-communicatie en biedt methoden om registers te lezen en te schrijven. Hoewel deze service specifiek is voor Modbus, is de code zo geschreven dat een nieuwe communicatiemethode eenvoudig kan worden geïntegreerd. Door een *parent*-klasse te introduceren waar zowel de *ModbusService* als een nieuwe communicatieservice van kunnen erven, kan de applicatie worden uitgebreid zonder bestaande functionaliteit te verstoren. Dit concept van uitbreidbaarheid is een terugkerend thema in de applicatie. Hoewel niet alle klassen momenteel een *parent* hebben, is er bewust gekozen voor abstracte namen en een modulaire structuur om toekomstige *refactorings* te vergemakkelijken.

De verbinding tussen de applicatie en de RPP wordt vaak gecontroleerd om onverwachte fouten te voorkomen. Als er iets mis is met de verbinding kan een verbindingcheck vrij veel tijd kosten. Om deze tijd te beperken hebben we twee soorten checks in de *ModbusService*: *check\_connection* en *check\_connection\_and\_repair*. *Check\_connection* kijkt niet alleen of de COM interface gevonden kan worden, maar test ook of de verbinding succesvol is door een register uit te lezen en te controleren of hier een valide respons van komt. *Check\_connection\_and\_repair* doet hetzelfde, maar als hier uitkomt dat er geen succesvolle verbinding is maakt hij de Modbus *client* opnieuw aan, wat vaak een oplossing biedt, maar wel al snel twee keer zo lang duurt. Momenteel wordt de langere methode automatisch gebruikt wanneer de verbinding wegvalt terwijl een RPP actief gemonitord wordt of wanneer in de *Overview* tab voor een individueel device de *Check connection* knop wordt gebruikt. Er worden ook verbindingchecks uitgevoerd bij het opstarten van de applicatie en bij het betreden van de *Setup*- of de *Overview*-tab om de gebruiker zo'n accuraat mogelijk informatie te presenteren over de verbindingstatus, maar om vertraging van de applicatie te voorkomen wordt hier de snellere check gebruikt.

Bij elke methode in de *ModbusService* worden de antwoorden van de registers uitgebreid gecontroleerd om fouten zo gedetailleerd mogelijk te loggen en verbindingproblemen snel te detecteren. Dit verhoogt niet alleen de betrouwbaarheid van de applicatie, maar maakt het ook eenvoudiger om problemen te debuggen. De *SignalEmitter* wordt hier ook gebruikt, zodat zelfs deze diepe backend signalen kan sturen naar de



frontend zonder directe afhankelijkheid. Dit maakt het mogelijk om gebruikers direct op de hoogte te stellen van fouten of belangrijke gebeurtenissen, zoals verbingsproblemen of onverwachte waardes.

Een ander belangrijk aspect van de *ModbusService* is het gebruik van een *enum* voor de registers, die de structuur van de RPP-handleiding volgt. Dit maakt de code niet alleen leesbaarder, maar vermindert ook de kans op fouten bij het werken met registeradressen. De registers worden gebruikt voor verschillende acties, zoals het starten en stoppen van apparaten, het instellen van parameters en het uitlezen van waarden zoals spanning en stroom. Deze structuur maakt het eenvoudig om nieuwe registers toe te voegen als de functionaliteit van de apparaten wordt uitgebreid of vervangen.

De *DeviceCommunicationController* fungeert als een abstractielaag bovenop de *ModbusService* en vertaalt meer complexe acties, zoals het starten van een apparaat, naar een reeks specifieke methoden in de *ModbusService*. Dit maakt de code overzichtelijker en zorgt ervoor dat de *ModbusService* zich kan richten op de technische details van de communicatie. De controller bevat ook een *callback* naar de *DeviceController*, die wordt gebruikt voor monitoring. Wanneer een apparaat bezig is worden om de seconde verschillende registers uitgelezen, om zo in de gaten te houden dat het ingestelde programma juist wordt uitgevoerd. Dankzij de *callback* wordt de *DeviceController* automatisch op de hoogte gesteld van deze uitgelezen waardes, waarna hij deze kan verwerken voor presentatie op de gebruikersinterface. Dit monitoring gebeurt op een aparte *thread*, zodat het proces kan blijven draaien terwijl de applicatie andere taken uitvoert. Connectiviteitschecks worden ook vaak op een aparte thread uitgevoerd, vooral wanneer ze puur informatief zijn. Dit voorkomt dat de applicatie vastloopt tijdens langdurige checks, die soms wel 20 seconden kunnen duren. Als een connectiviteitscheck echter wordt uitgevoerd als onderdeel van een sequentiële actie, zoals het starten van een apparaat, gebeurt dit niet op een aparte thread.

Het monitoren van apparaten is een van de meest complexe onderdelen van de applicatie. Tijdens het monitoren worden de verbinding, de verstreken tijd en de outputwaarden (spanning of stroom) continu gecontroleerd. De outputwaarden worden gecontroleerd op basis van een vooraf gedefinieerde range, wat een begin is voor toekomstige uitbreidingen naar meer geavanceerde validatie. Als de verbinding wegvalt, blijft het monitoren doorgaan door elke seconde opnieuw verbinding proberen te maken. Dit ontwerp zorgt ervoor dat het monitoren automatisch kan worden hervat zodra de verbinding is hersteld, tenzij de gebruiker het proces handmatig stopt. Tijdens het monitoren worden signalen verstuurd bij fouten of belangrijke gebeurtenissen, zodat de gebruiker altijd op de hoogte is van de status van het apparaat.

Er is ook een begin gemaakt aan het monitoren van apparaten die niet in de huidige sessie van de applicatie zijn aangezet. Hoewel de applicatie al kan herkennen dat een apparaat actief is (en zo kan voorkomen dat er een nieuw programma wordt opgezet zonder het oude af te zetten), kan het nog niet de juiste registers uitlezen om het monitoren volledig te ondersteunen. Dit is een toekomstige uitbreiding die de functionaliteit verder zal verbeteren.

## 5.5. Gebruikers

Het laatste element van de applicatie zijn de gebruikers. Zoals eerder vermeld hebben we drie lagen gebruikers '*Regular*', '*Technician*', en '*God*'. Iedere gebruiker heeft een naam, een wachtwoord en een uuid. In de huidige versie van de applicatie wordt de gebruikersfunctionaliteit alleen gebruikt om bepaalde functionaliteit af te schermen van bepaalde gebruikers, maar het zou een grotere rol kunnen spelen in toekomstige uitbreidingen door bijvoorbeeld uitgebreidere logs te maken of een functionaliteit toe te voegen waarbij gebruikers projecten kunnen maken waarbij verschillende profielen en aantekeningen samengebundeld kunnen worden.

De wachtwoorden worden geëncrypt met behulp van *bcrypt*, een veilige en veelgebruikte methode voor het hashen van wachtwoorden. *Bcrypt* maakt gebruik van salting, waarbij een willekeurige waarde aan het wachtwoord wordt toegevoegd voordat het wordt gehasht. Dit voorkomt dat identieke wachtwoorden dezelfde hash opleveren, wat brute-force-aanvallen en rainbow table-aanvallen aanzienlijk bemoeilijkt. Bij het inloggen wordt het ingevoerde wachtwoord opnieuw gehasht en vergeleken met de opgeslagen hash om de authenticiteit te verifiëren.



Bij gebruikers wordt, net zoals bij de profielen, gebruik gemaakt van *lazy loading*. Dit bespaart geheugen en verhoogt de efficiëntie van de applicatie, vooral wanneer er veel gebruikers zijn. De *UserManager* beheert het laden en opslaan van gebruikersgegevens, die worden opgeslagen in JSON-bestanden. Een speciale gebruiker, de eerdergenoemde *recovery user*, is gedefinieerd in de *UserManager*. Deze gebruiker heeft een vaste UUID en kan worden gebruikt om toegang te krijgen tot de applicatie in geval van een verloren wachtwoord.

Bij het aanmaken van nieuwe gebruikers of profielen wordt ervoor gezorgd dat namen uniek zijn. Net zoals bij apparaten genereert de controller een unieke naam door te controleren of de standaardnaam al bestaat en, indien nodig, een nummer toe te voegen (bijvoorbeeld "New User 1", "New User 2"). Dit voorkomt conflicten en maakt het proces van het toevoegen van nieuwe entiteiten eenvoudig en foutloos.

De login-flow is een goed voorbeeld van de samenwerking tussen de verschillende lagen van de applicatie. Wanneer een gebruiker probeert in te loggen, stuurt de UI een signaal naar de *UserController*. De controller gebruikt vervolgens de *session\_manager* om de authenticatie af te handelen. De *session\_manager* controleert de ingevoerde gegevens tegen de opgeslagen gebruikersgegevens en logt de gebruiker in als de gegevens correct zijn. Bij een succesvolle login wordt vanuit de *UserController* de *signal\_emitter* gebruikt om de rest van de applicatie hiervan op de hoogte te stellen. De volledige GUI vangt dit signaal op en past zich aan op de rol van de nieuwe ingelogde gebruiker.

Het beheren van gebruikers gebeurt via de *UserController*, die methoden in de *UserManager* aanroept. De controller fungeert voornamelijk als een doorgeefluik, maar bevat ook validatiechecks om ervoor te zorgen dat de ingevoerde gegevens correct zijn. Deze validatie vindt plaats op meerdere niveaus: in de UI, in de controller en in de User-klasse zelf. Deze gelaagde aanpak zorgt ervoor dat fouten vroegtijdig worden opgevangen en verhoogt de betrouwbaarheid van de applicatie.

## 6. Besluit

Hiermee zijn we aangekomen bij de afsluiting van dit verslag en het bijbehorende project. In dit project stond het ontwikkelen van een betrouwbare, gebruiksvriendelijke en uitbreidbare applicatie voor de besturing van RPP's via Modbus centraal. Terugkijkend kan gesteld worden dat deze doelstellingen succesvol zijn behaald. De applicatie beschikt over een eenvoudige, toegankelijke GUI voor eindgebruikers, en de onderliggende code is modulair opgezet met het oog op toekomstige uitbreidingen en onderhoud.

Tijdens de uitvoering van het project heb ik het belang van een grondige voorafgaande functionele analyse sterk leren waarderen. Deze analyse heeft niet alleen richting gegeven aan de technische keuzes, maar ook gezorgd voor een helder kader waarbinnen ik de applicatie kon ontwikkelen. Daarnaast heb ik mijn vaardigheid in modulair programmeren verder kunnen ontwikkelen, wat niet alleen de kwaliteit van de huidige applicatie ten goede is gekomen, maar ook van waarde zal zijn in toekomstige projecten.

Hoewel het project succesvol is afgerond en de applicatie inmiddels operationeel is binnen de organisatie, biedt het huidige systeem nog volop ruimte voor verdere optimalisatie. Bijvoorbeeld:

- De optie om een programma op een willekeurig moment te pauzeren.
- Momenteel worden de parameters in de profielen (deels) berekend volgens een vast formule. Voor deze berekeningen wordt nu een Excel file gebruikt, maar dit zou ook als functionaliteit aan de applicatie toegevoegd kunnen worden.
- Als een programma al bezig is zodra de applicatie opstart, herkent de applicatie dit. Dit programma kan gestopt worden, maar niet gemonitord. Dit monitoren zou toegevoegd kunnen worden.
- Het monitoren op onverwachte output waardes zou gedetailleerder gedefinieerd kunnen worden.

Dergelijke uitbreidingen zouden niet alleen het dagelijks gebruik ten goede komen, maar ook de betrouwbaarheid en flexibiliteit van de RPP-installatie verhogen.

Tot slot hoop ik dat toekomstige studenten met evenveel enthousiasme aan nieuwe uitbreidingen en optimalisaties zullen werken, en zo verder kunnen bouwen op de basis die met dit project is gelegd.

## 7. Literatuurlijst

- Baselier, J. (2025, Mei 7). *Hoe werkt ModBus*. Opgehaald van Jarno Baselier | Professional No-Nonsense Cybersecurity: <https://jarnobaselier.nl/hoe-werkt-modbus/>
- Blinker. (2010, Februari 14). *Blinker*. Opgehaald van Blinker: <https://blinker.readthedocs.io/en/stable/>
- Geeks for Geeks. (2024, Augustus 21). *Singleton pattern in Python*. Opgehaald van Geeks for Geeks: <https://www.geeksforgeeks.org/singleton-pattern-in-python-a-complete-guide/>
- Geeks for Geeks. (2025, April 5). *Observer Pattern*. Opgehaald van Geeks for Geeks: <https://www.geeksforgeeks.org/observer-pattern-set-1-introduction/>
- Geeks for Geeks. (2025, Februari 18). *What is Lazy Loading?* Opgehaald van Geeks for Geeks: <https://www.geeksforgeeks.org/what-is-lazy-loading/>
- Henning, N. (2025, Maart 25). *PyQtToast*. Opgehaald van GitHub: <https://github.com/niklashenning/pyqttoast?tab=readme-ov-file>
- Lavasani, A. (2024, Januari 4). *Design Patterns in Python: Mediator*. Opgehaald van Medium: <https://medium.com/@amirm.lavasani/design-patterns-in-python-mediator-ca42c2caca52>
- Lutkevich, B. (2021, Juni 1). *race condition*. Opgehaald van TechTarget: <https://www.techtarget.com/searchstorage/definition/race-condition>
- Moeller, G. (2024, Februari 8). *The Single Responsibility Principle (SRP) of SOLID*. Opgehaald van Medium: <https://giovannamoeller.medium.com/the-single-responsibility-principle-srp-of-solid-eb2feed0c64b>
- Mogylatov, R. (2025, April 29). *Dependency injection and inversion of control in Python*. Opgehaald van Dependency Injector: [https://python-dependency-injector.ets-labs.org/introduction/di\\_in\\_python.html](https://python-dependency-injector.ets-labs.org/introduction/di_in_python.html)
- NetVla Group. (2025, Maart 4). *Reverse Pulse Plating - Technique Used In PCB Manufacturing Process*. Opgehaald van NetVia Group an Epac Company: <https://www.netviagroup.com/solutions/reverse-pulse-plating.html>
- oliviervdier. (2012, Juli 6). *dispatcher 1.0*. Opgehaald van PyPi: <https://pypi.org/project/dispatcher/>
- Osbourne, A. (2024, Januari 15). *Mastering Dependency Inversion in Python Coding*. Opgehaald van ArjanCodes: [https://arjancodes-com.translate.goog/blog/dependency-inversion-principle-in-python-programming/?\\_x\\_tr\\_sl=en&\\_x\\_tr\\_tl=nl&\\_x\\_tr\\_hl=nl&\\_x\\_tr\\_pto=sc](https://arjancodes-com.translate.goog/blog/dependency-inversion-principle-in-python-programming/?_x_tr_sl=en&_x_tr_tl=nl&_x_tr_hl=nl&_x_tr_pto=sc)
- Owuordove. (2024, Januari 3). *Hands-On Guide to Model-View-Controller (MVC) Architecture in Python*. Opgehaald van Medium: <https://medium.com/@owuordove/hands-on-guide-to-model-view-controller-mvc-architecture-in-python-ec81b2b9330d>
- Refactoring Guru. (2025). *Template Method*. Opgehaald van Refactoring Guru: <https://refactoring.guru/design-patterns/template-method>
- Smoot, J. (2025, Mei 7). *RS-485 Serial Interface Explained*. Opgehaald van Same Sky: <https://www.sameskydevices.com/blog/rs-485-serial-interface-explained?srsId=AfmBOor2lQfreSv8laLi2aT-1KE5nqgUxwIR-wTUdkzw8FbA1NqP3PLz>
- The Qt Company Ltd. (2025, Maart 4). *Getting Started - QT for Python*. Opgehaald van QT for Python: <https://doc.qt.io/qtforpython-6/gettingstarted.html#getting-started>
- The Qt Company Ltd. (2025, April 28). *QStackedWidget Class*. Opgehaald van Qt Group: <https://doc.qt.io/qt-6/qstackedwidget.html>
- The Qt Company Ltd. (2025, Maart 4). *QT | Software Tools for Each Stage of Software Development Lifecycle*. Opgehaald van QT: <https://www.qt.io/>
- The Qt Company Ltd. (2025, April 28). *QTabWidget Class*. Opgehaald van Qt Group: <https://doc.qt.io/qt-6/qtabwidget.html>
- The Qt Company Ltd. (2025, Maart 4). *Signals & Slots*. Opgehaald van Qt Group | Documentation: <https://doc.qt.io/qt-6/signalsandslots.html>
- Thorben. (2018, Maart 28). *SOLID Design Principles Explained: The Open/Closed Principle with Code Examples*. Opgehaald van Stackify: <https://stackify.com/solid-design-open-closed-principle/>
- Weis, O. (2024, Oktober 4). *Modbus vs RS485*. Opgeroepen op Maart 4, 2025, van Virtual Serial Port: <https://www.virtual-serial-port.org/articles/modbus-vs-rs485/>

## Bijlage A

Gewogen beslissingsmatrix voor keuze GUI framework. Verschillende talen worden vergeleken, ieder gerepresenteerd door 1 framework. PySide6 is de winnaar, voornamelijk door de goede Modbus ondersteuning vanuit Python.

Criteria	Gewicht	Python (PySide6)	C++ (QT)	Java (JavaFX)	C# (WPF)	Electron (JS)
Leercurve & ontwikkelingssnelheid	20%	9	5	7	7	9
Prestaties	15%	7	10	7	9	5
RS485 / Modbus ondersteuning	25%	10	6	5	7	3
Cross-platform compatibiliteit	20%	10	10	10	5	10
Licentie en kosten	10%	10	10	10	10	10
Ondersteuning moderne UI	10%	7	7	9	9	10
Totaalscore	100%	9.05	7.70	7.60	7.40	7.30

## Bijlage B

Het klassendiagram hieronder geeft een schematische weergave van de klassen die samen de backend van de applicatie vormen.

