# Displaimer Arkitektur

Teo Klestrup Röijezon

25 maj 2017

**Sammanfattning**

Detta dokument beskriver den övergripande arkitekturen för Displaimer (tidigare känd som WiLCD).

## Innehåll



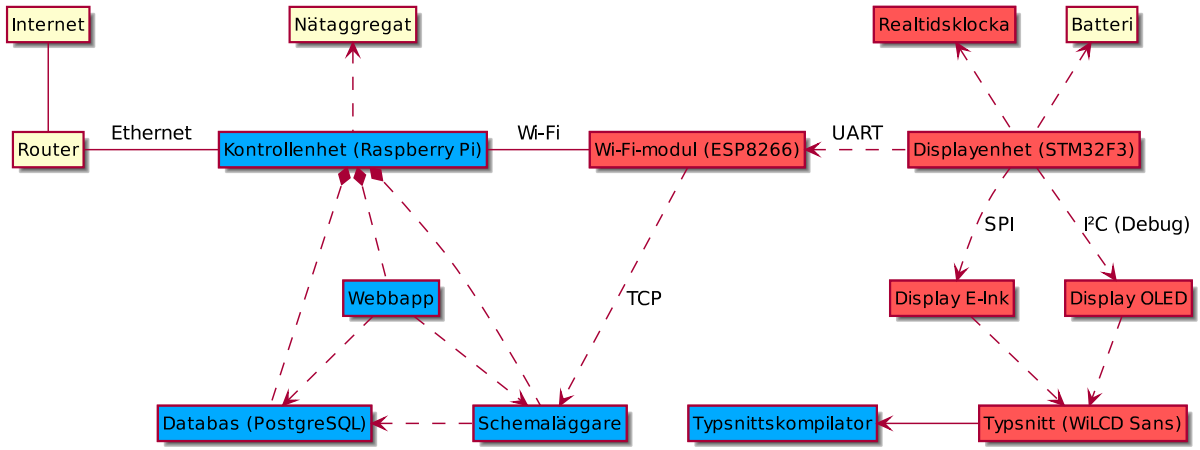
## 1 Dokumentets Syfte och Innehåll

Syftet med detta dokument är att förklara vilka komponenter som Displaimer innehåller, hur de kommunicerar, samt vad som lett till dessa beslut.

Detta dokument behandlar även allmänna kodkonventioner, men *inte* detaljsyrning för hur varje komponent är implementerad.

## 2 Struktur

Projektet är primärt uppdelat i schemaläggaren, webbplatsen, samt displayenheten, sammankopplade enligt följande diagram:




Blåmarkerade områden ansvaras främst av Teo Klestrup Röijezon och Sebastian Heimlén, och rödmarkerade områden ansvaras främst av Yobart Amino och Henrik Björklund. Andra delar är användarens ansvar att tillhandahålla.

Webbappen låter primärt användare skriva meddelanden som displayen ska visa, samt även användare schemalägga meddelanden att visa senare, dessa skickas till schemaläggaren. Dessutom har den ett användarsystem, för att begränsa vem som kan ändra meddelandet.

Schemaläggaren kommunicerar med displayenheten (via ESP8266-modulen) och bestämmer vilket meddelande som ska visas. Dessutom används den som källa för att kalibrera displayenhetens klocka.

Databasen används som primär sanningskälla för båda dessa.

Displayenheten har till ansvar att ta emot och visa meddelanden, samt hålla den visuella klockan uppdaterad. Meddelandet och det nuvarande klockslaget visas på E-Ink-displayen. Felsökningsmeddelanden skickas till OLED-displayen (endast ikopplad under felsökning).

Typsnittskompilatorn kompilerar vårat typsnitt (”WiLCD Sans”) till motsvarande C-kod.

# 3 Kodkonventioner

Webbappen och schemaläggaren är skrivna i Scala, typsnittskompilatorn är skriven i Rust, och displayenhetens firmware är skriven i C. Varje av dessa har egna konventioner.

## 3.1 Scala

Vi följer Scalas officiella stilguide, med följande undantag:

- Filer med flera klasser (se 3.1.1 för det främsta fallet) är namngivna efter den primära klassen, och börjar fortfarande på stor bokstav
- Det klassas fortfarande som rent (”pure”) att orsaka en läsfråga till databasen, eller att skapa en Future, så länge dess innehåll också är rent
- En explicit typdefinition krävs för publika metoder och fält
  - Gäller ej för Slick-kolumndefinitioner

Dessutom delar vi upp viktiga funktionalitetsområden i ”tjänster”, så att användargränssnittet i webbapplikationen t.ex. inte blir beroende av några detaljer i databasen.

### 3.1.1 Databasmodeller

För varje databastabell har vi i allmänt tre klasser, som ligger i samma fil. Om vi exempelvis ska komma åt tabellen *nouns* så skapar vi tre klasser i filen Noun.scala:

- Klassen Noun, en ”case class” som innehåller tabelldefinitionen
  - ID-kolumner utelämnas, ärv i stället från HasId och implementera IdType
- Klassen Nouns, en Slick-tabelldefinition
  - Utelämna ID-kolumner, ärv i stället från IdTable[T], definiera metoden all som motsvarande * i vanliga tabeller, och definiera * som (id, all)
- Objektet Nouns, en innehåller en TableQuery vid namn tq, samt vanliga frågor

Till exempel, för den följande tabellen:

```
CREATE TABLE nouns(
       id SERIAL PRIMARY KEY,
       foo TEXT NOT NULL,
       bar INTEGER
)
```

Skulle Noun.scala kunna se ut som följer:

```
package models

import models.PgProfile.api._

case class Noun(foo: String, bar: Option[Int]) extends HasId {
  override type IdType = Long
}

class Nouns(tag: Tag) extends IdTable[Noun](tag, "nouns") {
  def foo = column[String]("foo")
  def bar = column[Option[Int]]("bar")

  def all = (foo, bar) <> (Noun.tupled, Noun.unapply)

  overrid def * : ProvenShape[WithId[Noun]] =
    (id, all) <> ((WithId.apply[Noun] _).tupled, WithId.unapply[Noun])
}

object Nouns {
  private[models] def tq = TableQuery[Nouns]

  def find(id: Id[Noun]): Query[Noun, WithId[Noun], Seq] =
    tq.filter(_.id === id)
}
```

### 3.2 Rust

Vi följer Rusts officiella stilguide, och rustfmt används för att automatiskt formatera koden.

### 3.3 C

Vi följer i allmänt följande regler:

- 2 mellanrum för indentering
- Alla globala definitioner (funktioner, typer, globala variabler, konstanter) börjar med modulens namn
- PascalCase för funktionsnamn, med _ mellan och efter moduler, exempelvis Epaper_Init
  - Funktioner som är interna i en modul börjar med _, exempelvis _Epaper_Transmit_Byte
- camelCase för typer och variabelnamn
- SCREAMING_SNAKE_CASE för makron och konstanter
- Öppnande måsvingar ({) är inte på en ny rad, men företräds alltid av ett mellanrum
  - Stängande måsvingar (}) är på en ny rad om det inte följs av en else-sats
- Vid inbyggda satser (som for eller if) är det ett mellanrum mellan namnet och den öppnande parantesen ("("), exempelvis "if (1) {"
  - Detta gäller ej vid vanliga funktionsanrop eller indexering
- Måsvingar används *alltid* vid inbyggda satser som tar ett block (som for eller if), även om bara en sats följer
  - Detta gäller ej vid loopar utan satser, det är tillåtet att skriva "while (!ready);"