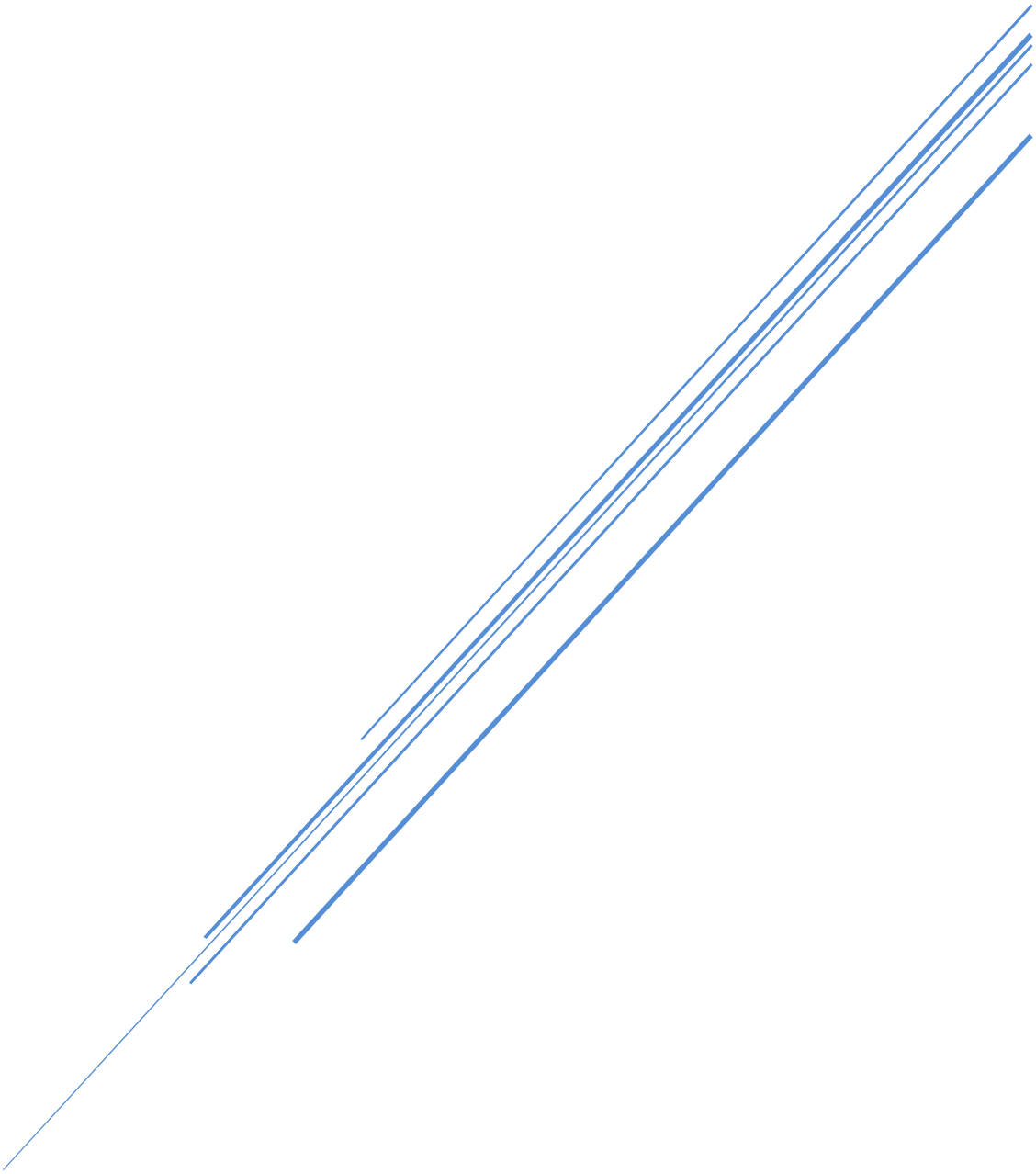
DND VIEWER

S1060679 – Dion van den Berg



Hogeschool Leiden
IIATIMD

Inhoudsopgave

Het probleem / oplossing

Het is mij al vaker voorgekomen dat ik tijdens een sessie Dungeons en Dragons (DnD) (https://dnd.wizards.com/) niet direct een bepaalde statistiek van een monster voor me kon halen. Het komt vaker voor dat je, als je een potje als Dungeon Master voorzit, je ineens willekeurig een vijand of meerdere moet neerzetten voor je spelers. Dit zorgt er dan voor dat de grote Monster manual (https://dnd.wizards.com/products/monster-manual) tevoorschijn moet halen om hierin te gaan bladeren. Daarbij heb ik dan een groot boek erbij op tafel liggen op vaak al een beperkt formaat tafel zoals te zien is op het screenshot hiernaast. Ik wilde hier een oplossing voor.

Mijn idee was dan ook om een zo simpel mogelijke app te maken die puur doet wat hij moet doen. Monsters tonen!
Ik wil alle informatie die ik nodig heb simpel en snel bij de hand hebben zonder advertenties, abonnementen of vooral: te veel informatie. Want dat is iets wat ik vooral bij de officiële app merk van DnD is dat ze gigantisch veel informatie beschikbaar hebben. Magie, Kleding, Wapens, Lore. En dan ergens ertussen 5 schermen en een paar filters verder ben je dan aan de Monsters. Dat moest simpeler kunnen.

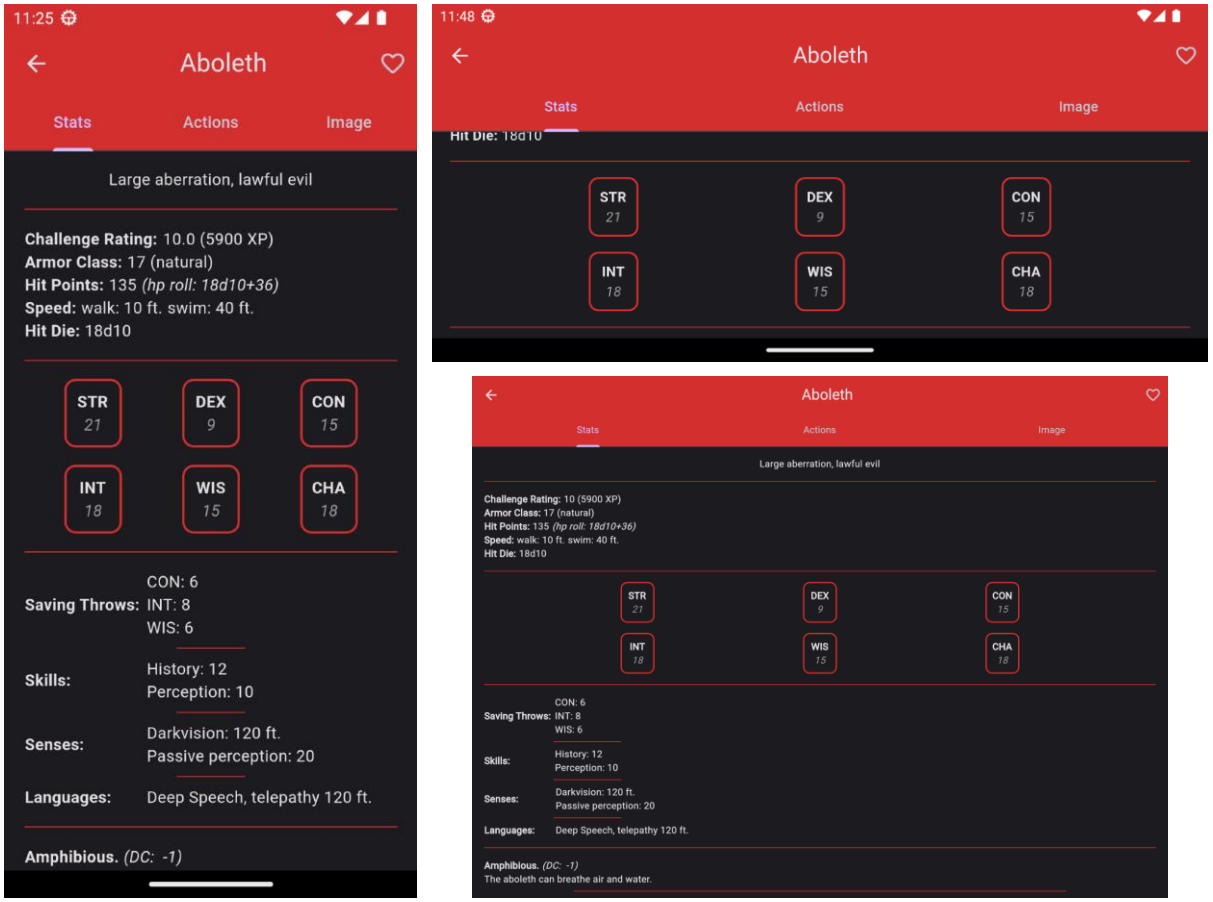
Screenshot 1: volle dnd tafel ter illustratie

Leerdoelen

Responsieve / adaptieve lay-outs

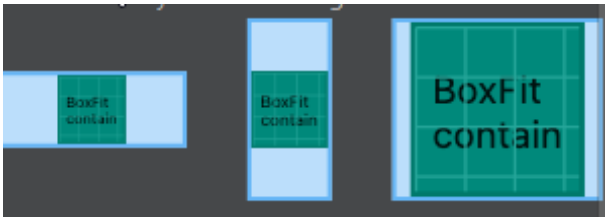
De gemaakte app is gemaakt om te werken op zowel een telefoontoestel als IOS of Android en de grotere broertjes de tablets. Om dit te bereiken heb ik zo min mogelijk gebruikgemaakt van vaste variabelen en juist de kracht van de interne functies van Flutter gebruikt.
Flutter heeft voor zijn Lay-out widgets zoals de Column of Row Widgets gemakkelijk parameters die meegegeven kunnen worden. Zoals de "*mainAxisAlignment: MainAxisAlignment.spaceEvenly"*. Met deze parameter kan aangegeven worden dat de Widget de "children" van deze Widgets gelijk verdeeld moeten worden over de beschikbare ruimte. Dit kan je indammen door de Column Widget ook weer in zijn eigen Container te zetten die wel een limiet op zijn waarde heeft. Op deze manier kan gemakkelijk rekening gehouden worden met veranderende formaten van de lay-out maar ook met het draaien van je toestel.

In het voorbeeld (screenshot 1) hieronder is te zien hoe de Status blokken (STR DEX CON) een andere afstand van elkaar nemen als je het scherm draait omdat hier meer ruimte beschikbaar is gekomen.

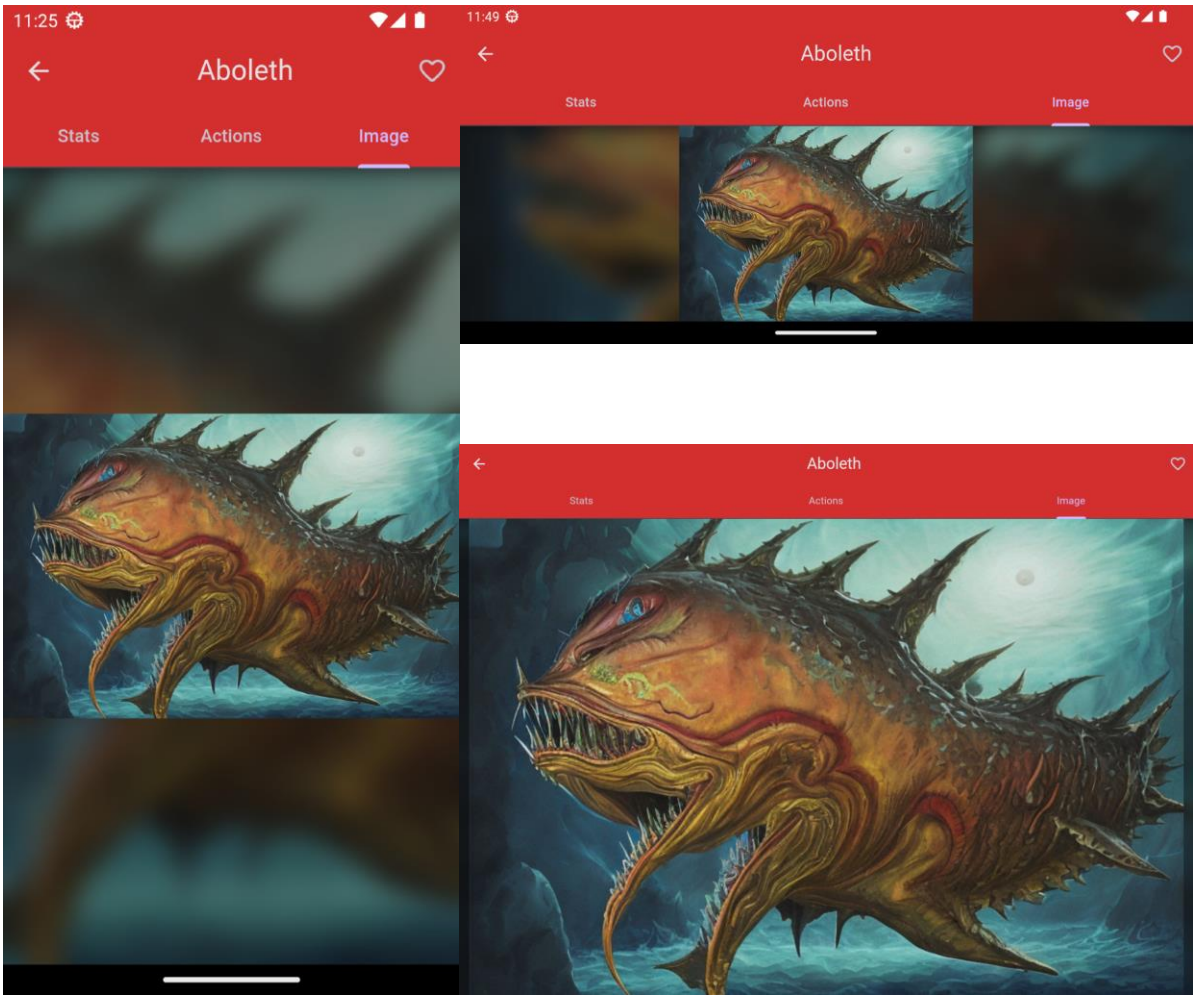


Screenshot 2: Layout of verschillende formaten (telefoon: Links, Telefoon-landscape: rechtsboven, Tablet: rechtsonder)

Als 2e voorbeeld heb ik gebruik gemaakt van het resizen van foto’s. Afbeeldingen worden niet altijd in hetzelfde formaat aangeleverd en dit zorgt ervoor dat hier ook niet op gerekend kan worden. Ik zorg ervoor dat een afbeelding maar zo breed mag zijn als de telefoon of zichzelf. Waarna de afbeelding een *“fit.contain”* meekrijgt om zichzelf op de best mogelijke manier te plaatsen waarin hij zijn verhouding behoudt. Al het overige wordt opgevuld met een geblurde versie van zichzelf. Op deze manier is toch de gehele pagina gevuld en ziet het er gelikt uit.



Screenshot 3: Boxfit.contain past de afbeelding aan, aan de hoogte of de breedte van een container.



Screenshot 4: afbeelding op verschillende layouts

Navigeren tussen schermen

Omdat alle informatie op 1 pagina weergeven niet echt gebruiksvriendelijk is en er dan heel erg veel gescrold moet worden heb ik gekozen om de pagina's op te delen in 2 "fysieke" pagina's die op zichzelf weer opgedeeld zijn in tabs. De hoofdpagina laat alleen een grote lijst zien met alle monsters erop en in een tweede tab staan de gebruiker zijn favorieten. De items in zowel het hoofdscherm als de favorieten zijn klikbaar en zorgen ervoor dat de

gebruiker door navigeert naar een nieuw scherm welke generiek is voor alle items in de lijst. Deze 2e pagina is onderverdeeld in Stats, Actions en Images. Echter zal de image tab niet weergegeven worden mocht er geen afbeelding beschikbaar zijn vanuit de API. Dit komt vaker voor en moet dus ook zeker rekening mee gehouden worden.

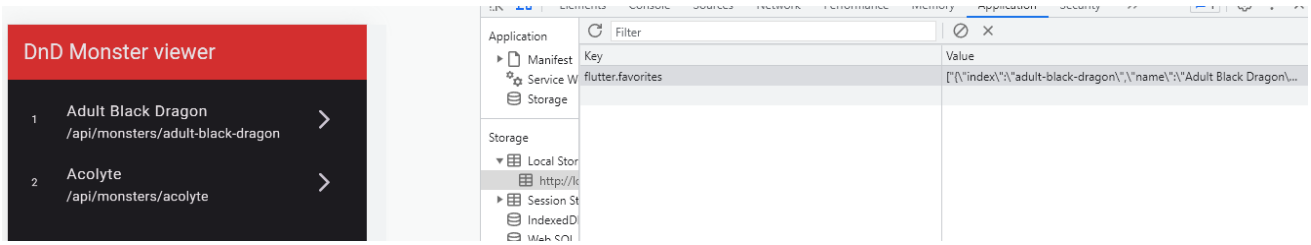
Het opslaan van data

De data die opgeslagen moet worden, zijn de favorieten van de gebruiker en de zelfgemaakte monsters van de gebruiker. Deze data wordt opgeslagen op de mobiele telefoon van de gebruiker of in een map op de computer. Dit verschilt per platform. Ik maak gebruik van de Shared Preferences library (pub.dev/packages/shared_preferences) om gemakkelijk vanuit Flutter Strings op te slaan. Doordat deze library zelf geen objecten kan vasthouden vanuit de app, zorg ik ervoor dat ik alles convert naar een String. Voor het opslaan van een favoriet ga ik de volgende stappen door.

```
"results": [
  {
    "index": "aboleth",
    "name": "Aboleth",
    "url": "/api/monsters/aboleth"
  },
```

1. Gebruiker klikt op het hartje bij een monster
2. Het Monster object wordt naar een functie gestuurd
3. Deze convert het monster object naar Json
4. Deze voeg ik toe aan de huidige lijst aan favorieten
5. Deze voeg ik toe aan de op het platform opgeslagen lijst door de array aan Strings aan te vullen met “await prefs.setStringList("favorites", favoStrings);”
6. En hierna wordt de Favorieten pagina herladen met data bij het terugkeren van een monster pagina.

Het opslaan van een eigen monster gaat vrijwel op dezelfde manier, maar dit wordt geïnitieerd vanuit de monster toevoegen knop



Screenshot 5: data opgeslagen in chrome local storage

Het herkennen van voor - en nadelen van platformen en hybride bibliotheken

Uit alles blijkt eigenlijk dat vanuit 1 Framework werken voor meerdere apparaten extreem efficiënt is. Ik hoef geen tot weinig kennis op te doen voor de native programmeertalen en het bouwen van de applicaties kan allemaal rechtstreeks vanuit Android Studio. Alleen IOS is als buitenbeentje dat daar een Mac voor aan de pas moet komen om de applicatie te releasen. Voor Android is het simpel genoeg om “Flutter build apk” te typen en een minuut later kan je op een mobiele telefoon de apk installeren. Ook voor het uitlezen van de native sensoren is maar minimale kennis nodig. Als ik bijvoorbeeld de gyroscoop uit wil lezen van mijn Android apparaat hoef ik alleen voor de juiste plug-in te zorgen.
Het sensor_plus (pub.dev/packages/sensors_plus) pakket zorgt ervoor dat ik kinderlijk makkelijk een sensor kan uitlezen. Deze pakketten zorgen ervoor dat de native functies aangesproken kunnen worden op een IOS of Android bijvoorbeeld. Of vaak hebben ze een eigen native versie van hun library draaien op de Android of IOS kant van het apparaat zelf waarmee Flutter kan praten.

Een groot nadeel van Flutter kan liggen aan projecten die nog geen eigen vertaalde bibliotheek hebben. In dit project heb ik er niks mee hoeven doen, maar in een werk project moest ik tracking functies inbouwen die alleen een native SDK hadden voor Android/IOS. Dit zorgde ervoor dat ik dus direct functies aan moest spreken in de Android/iOS code. Flutter heeft hier Platform channels (docs.flutter.dev/platform-integration/platform-channels) voor uitgebracht om een soort tunnel te hebben naar deze platformen toe. Hierdoor kan je vrij eenvoudig functies of objecten heen en weer gebruiken tussen de platformen in terwijl je gewoon het overzicht hebt in Flutter zelf. Dit zorgt er ook voor dat bedrijven die voor flutter programmeren toch nog oudere bibliotheken kunnen gebruiken die alleen voor de native platformen zijn geschreven.

Als laatste nadeel zou je kunnen zeggen dat de performance van Native het wint over een Flutter applicatie. Dit is in vele gevallen waar. Als ik kijk naar cijfers van een onderzoek (https://auberginesolutions.com/blog/comparing-the-performance-of-an-app-built-with-native-android-and-flutter/) Is te zien dat flutter over het algemeen 2x langer nodig heeft voor dezelfde taken. Dit loopt echter tot diep in de milliseconden wat voor een normale app totaal niet te merken is. Veelal zijn de gecompileerde apps zeer snel en is er uit de elementen niet te merken dat het niet een native applicatie is. Ik snap dat dit voor wiskundige calculaties of grotere animaties (games) anders kan zijn.

Student kan ontwikkelen voor meerdere apparaten

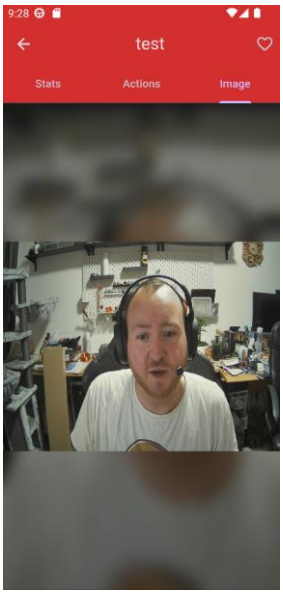
Door Flutter is het extreem gemakkelijk om voor meerdere apparaten te ontwikkelen, daarom houd ik dit sub-hoofdstuk kort met de tekst: *"It Just Works"*.
De applicatie is te gebruiken op tenminste 4 verschillende apparaten die ik heb kunnen testen. Deze zijn getest aan de hand van Android studio en IOS in Xcode op een Mac.

- IOS/Android Telefoon (landscape and portrait),
- Tablets (landscape and portrait)
- Windows als app
- Chrome/Edge als browser

Een sensor aanspreken en gebruiken

De sensor die ik heb gekozen om te gebruiken is de Camera. Deze gebruik ik om de gebruiker mogelijk een foto in te schieten voor zijn eigen monster. In praktijk zal een gebruiker vaak een bestand willen toevoegen uit zijn eigen foto bibliotheek, maar soms kan het voorkomen dat bijvoorbeeld de Kat of Hond van een gebruiker gebruikt wordt als voorbeeld voor een monster.
Om deze functionaliteit te showcases heb ik een kleine functie toegevoegd om een eigen monster toe te voegen. Hierin is de gebruiker vrij om zowel een bestand te uploaden als een eigen foto.

Ik maak gebruik van de plug-in image_picker (https://pub.dev/packages/image_picker) om gemakkelijk en snel de functionaliteit werkend te hebben binnen meerdere devices. Aangezien de Android emulator gemakkelijk de passtrough ondersteund van de computer zijn webcam heb ik deze gebruikt om te testen of het werkt. De Image picker roept de standaard camera functionaliteit van bijvoorbeeld Android aan en gebruikt deze om de image als file op te slaan op de telefoon. Hierna wordt de locatie van het bestand als opslag pad terug gestuurd om hierna opgeslagen te worden in het nieuwe object dat de gebruiker aan het aanmaken is.



Screenshot 6: foto uploaden van camera

Functionaliteit App

Ophalen van data

De hoofdpagina haalt puur alleen de lijst met alle bekende monsters op van de API. Hierin zijn alleen de naam, unieke index naam en de url beschikbaar van de API. Dit houdt in dat om meer info te hebben en dus de andere pagina's van data te voorzien ik een extra call moet doen.


```
{
  "count": 334,
  "results": [
    {
      "index": "aboleth",
      "name": "Aboleth",
      "url": "/api/monsters/aboleth"
    },
    {
      "index": "acolyte",
      "name": "Acolyte",
      "url": "/api/monsters/acolyte"
    },
    {
      "index": "adult-black-dragon",
      "name": "Adult Black Dragon",
      "url": "/api/monsters/adult-black-dragon"
    },
```


Screenshot 7: data /api/monsters

Ik maak gebruik van Provider/Consumers in de App. Dit zorgt ervoor dat ik op de achtergrond data kan vullen en verwerken. Denk hier aan het parsen van de API response body. En deze te verwerken in een Model om zo een object terug te krijgen die ik kan gebruiken over de gehele app. Het gemak van de Consumer is dat deze zelf ook een Listenable object is. Dat betekend dat veranderingen in zijn provider een rebuild kunnen triggeren van de Consumer zelf. Een hele handige usecase hiervoor is dan ook in gebruik bij de Monster detail pagina.

Na het klikken op een item van een monster navigeert deze naar de monster detail pagina toe. Deze pagina is op dat moment nog helemaal leeg van data. Alles wat deze meekrijgt is het Monster object met de naam, index en url. Deze Url is de link naar de api call van het gehele data object van dat monster met daarin alle gegeven die benodigd zijn om de pagina te vullen met informatie.

Door de consumer te gebruiken hoef ik geen gebruik te maken van Future objecten en kan ik de tijdelijke pagina bouwen (in dit geval een laad indicator), welke weer veranderd als het monster object gevuld is met alle informatie. De functie die dat doet heeft aan het einde een "notifyListeners()" functie die ervoor zorgt dat de Consumer weet dat hij opnieuw moet bouwen. Nu deze wel data heeft kan hij de complete pagina nu laden met alle informatie. Als ik dit niet zou doen zou alles mogelijk "null" bevatten en kost dit veel meer checks om alles netjes er werkend te houden in de Flutter app.

Ook maak ik gebruik van een plugin (https://pub.dev/packages/cached_network_image) om plaatjes te laten zien. Deze plugin houdt standaard 30 dagen zijn afbeeldingen vast op de lokale opslag van de telefoon. Hierdoor besparen we extra data gebruik, maar ook laad tijd voor de monster detail pagina.

Good? code practices

Tijdens het bouwen van de app heb ik geprobeerd om zoveel mogelijk op te delen in files die hun eigen functie hadden. In Flutter zou je in principe alles in 1 file kunnen doen en dan werkt het ook. Maar om alles overzichtelijk te houden heb ik geprobeerd om gedeeltelijk met het MvC model te werken. Alle data die ik ophaal van de API wordt eerst in de Provider (in dit geval een controller) opgehaald van het internet waarna deze door het juiste Model door een factory gehaald worden. Deze vult de objecten en zorgt ervoor dat alle data op de juiste plek staat. Dit geeft mij de mogelijkheid om niet een Json midden in mijn file te hoeven aanspreken met een complete array (voorbeeld: Json[“results”][“url”]) en deze ook nog eens moet checken of velden wel bestaan, maar nu gewoon het object kan pakken (voorbeeld: results.url) en hier zeker van kan weten dat het type en de inhoud hiervan gevuld is. Dit is vooral handig aangezien sommige onderdelen van de opgehaalde Json soms 4 diep kunnen gaan en je dan je schema erbij moet gaan houden om te weten wat ook alweer wanneer komt. Als object druk je gewoon op je . teken en zie je precies wat hieraan hangt en wat het type is.


```
{
  "name": "Psychic Drain (Costs 2 Actions)",
  "desc": "One creature charmed by the aboleth
  "attack_bonus": 0,
  "damage": [
    {
      "damage_type": {
        "index": "psychic",
        "name": "Psychic",
        "url": "/api/damage-types/psychic"
      },
      "damage_dice": "3d6"
    }
  ]
}
```


Screenshot 8:
Voorbeeld diepte Json

De controllers regelen alle extra functionaliteit die in nodig heb eromheen. Hierdoor is het toevoegen van een favoriet geregeld in een aparte klasse die ik overal aan kan spreken door het gebruik ook van een global library. Hierin bewaar ik ook de variabelen die ik nodig heb op meerdere pagina’s of om gemakkelijk een url te veranderen.

Naast dit alles zijn er de Views. Dit zijn de pagina’s zelf Die wat tonen. Ik heb de verschillende tabjes wel in de views zelf gehouden om het voor mijzelf overzichtelijk te houden.

Reflectie

Tijdens het maken van de app heb ik mij echt vermaakt. Ik had in tijden niks met flutter meer gedaan voor mezelf en was vooral vaak bezig met bugs op te lossen of klanten hun belachelijke functies in te bouwen.

Ik heb wel leuke nieuwe dingen geleerd over het opzetten van een app. Zoals het gemakkelijk opdelen van een pagina in meerdere tabs en het blurren van foto's. Ook had ik maar kort wat gedaan met een NFC sensor als proof of concept, maar heb ik dit nooit lang hoeven gebruiken en dat heb ik nu toch voor de app kunnen implementeren.

Al met al is het denk ik een geslaagde app. Graag zou ik er verder aan werken en een collega Dungeon master begon gelijk al user stories in te schieten voor een uitbreiding erop. Op het moment is de gebruikte API niet door mijzelf gebouwd, enkel gehost en wil ik daar nog wat aan toevoegen. Zo missen er gigantisch veel foto's voor monsters in de database. Dit is iets wat ik graag aan zou willen passen om deze ook bijvoorbeeld op de hoofdpagina te kunnen tonen. Net als de challenge rating van een monster zodat hier mogelijk op gesorteerd kan worden. Ook zou ik nog graag willen zoeken door de lijst heen op de hoofdpagina. Dit zijn toevoegingen die snel op de planning staan na de oplevering en ik naar uitkijk om te realiseren.