Project Robotica

Wessel Tip <contact@wessel.gg> (https://wessel.gg/)
Technische Informatica
Hoogeschool Inholland Alkmaar
Jaar 2, Semester 2 (Jan. 2024 - Jun. 2024)

2 juli 2024

Inhoudsopgave

1 Inleiding

In de moderne tijd speelt automatisering een steeds grotere rol in het verbeteren van efficiëntie en het verlichten van huishoudelijke taken.

Een goed voorbeeld hiervan is het stofzuigen. Dit wordt in de moderne tijd steeds meer door autonome robots gedaan. Deze robots zijn ontworpen om zelfstandig schoon te maken en tijd te besparen. Om dit te kunnen doen moet de robot niet alleen de ruimte kunnen navigeren, maar ook objecten kunnen vermijden en de meest efficiënte schoonmaakroute kunnen plannen.

Om aan de eigenaar van desbetreffende robot controle te geven wordt er vaak gebruik gemaakt van knoppen op de robot zelf. Dit is echter niet altijd even handig, omdat de robot vaak onder meubels staat en de knoppen niet altijd even goed bereikbaar zijn.

Om dit te verhelpen dient er een controlepaneel (ookwel bekend als een *Web Interface*) gemaakt te worden. Via deze interface zal de eigenaar de mogelijkheid krijgen om de robot op een afstand te bedienen en monitoren.

2 Probleemstelling

2.1 Probleemanalyse

API's (*Application Programming Interfaces*) zijn onmisbare componenten in moderne software. Door middel van een API in combinatie met een set van afspraken (een *protocol*) is het mogelijk meerdere softwarecomponenten met elkaar te integreren. Deze API's maken de communicatie tussen verschillende softwarecomponenten mogelijk.[1]

Er zijn over de jaren vele verschillende manieren bedacht voor het ontwerpen en implementeren van API's, elk met zijn eigen voordelen en nadelen. [2] In dit onderzoek wordt er een vergelijking gesteld tussen vier verschillende API architecturen (REST, SOAP, GraphQL en gRPC) om te bepalen welke het meest geschikt is voor het project.

Bij het kiezen van een API architectuur wordt er rekening gehouden met een bepaald aantal criteria. De API moet ontwikkelt worden zonder dat er externe libraries gebruikt worden. Dit betekent dat de API vanaf de grond af aan moet worden opgebouwd.

De API moet ook in staat zijn om de communicatie tussen de verschillende componenten van de robot te ondersteunen. Dit betekent dat de API de communicatie tussen de gebruiker en de robot moet ondersteunen, en de communicatie tussen de verschillende robots onderling.

De interface die wordt gepresenteerd aan de eindgebruiker dient alle verzamelde data te laten zien. Hij zal de mapping en de geplande route moeten presenteren in een overzichtelijke manier. Ook moet de robot diverse commando's vanuit de gebruiker kunnen ontvangen om zo de gewenste taken uit te kunnen voeren.

De gekozen API-architectuur moet eenvoudig te onderhouden zijn, met een

heldere en overzichtelijke structuur. De architectuur moet ook felxiebel genoeg zijn om toevoegingen en aanpassingen te ondersteunen zonder dat de codebase herschreven moet worden.

De robot vereist een structuele manier om deze gegevens uit te wisselen, en de optie om meerdere robots tegelijkertijd te besturen.

Een bijkomende eis vanuit de opdrachtgever is dat de software van het beroepsproduct object-georieënteerd is.

2.2 Vraagstelling

De hoofdvraag van dit onderzoek luidt:

> "*Welke API architectuur is het beste geschikt voor de specifieke eisen van het project, waarbij de communicatie tussen verschillende softwarecomponenten, de ondersteuning van gebruikersinteracties, en de flexibiliteit en onderhoudbaarheid van de code optimaal zijn?*"

Om deze hoofdvraag te beantwoorden worden de volgende deelvragen geformuleerd:

Voorafgaand aan het project zijn in deelparagraaf 2.1 een aantal eisen opgesteld, zoals dat de API moet worden opgebouwd zonder het benut van externe libraries. De volgende deelvraag zal dit beantwoorden:

> "*Welke stappen zijn nodig om de API vanaf de grond af aan op te bouwen?*"

Ook zal er gekeken worden welk architectuur het beste bij het project past. Om dit te beantwoorden is de volgende deelvraag geformuleerd:

> "*Hoe kunnen REST, SOAP, GraphQL, en gRPC worden vergeleken op basis van de gestelde criteria?*"

3 theoretisch Kader

3.1 RESTful (Representational State Transfer)

REST is een architectuur dat wordt gebruikt voor het communiceren van gegevens om netwerkapplicaties te ontwerpen.

Het maakt gebruik van standaard HTTP-methoden (zoals GET, POST, PUT, DELETE) om gegevens te communiceren tussen clients en servers.[3]

Door de simpiliciteit en het al gebruik maken van de bestaande HTTP protocollen is REST een populaire keuze voor API's. Dit maakt het ook erg geschikt voor web gebaseerde applicaties.

3.1.1 Voordelen van REST

1. **Eenvoudige structuur** — gebaseerd op standaard HTTP-methoden en JSON
2. **Makkelijke implementatie** — meeste programmeertalen ondersteunen de basis van webrequests en JSON serialisatie al
3. **Schaalbaarheid** — RESTful API's zijn stateless en kunnen eenvoudig horizontaal geschaald worden, echter zal de performance wel slechter zijn dan andere methodes zoals gRPC of GraphQL.[2]

3.1.2 Nadelen van REST

Echter heeft REST ook zijn nadelen, sommige hiervan zijn:

1. **Overbodig veel data** — Minder geschikt voor complexere query's en datastructuren (over-fetching en under-fetching)
2. **Documentatie** — Geen directe ondersteuning voor documentatie, schemas en typecontrole
3. **Real-time** — Doordat de connectie niet in stand wordt gehouden nadat de aanvraag is afgehandeld is het minder geschikt voor real-time communicatie.

3.2 SOAP (Simple Object Access Protocol)

SOAP is een protocol voor uitwisseling van informatie netzoals REST vermeld in deelparagraaf 3.1. Echter is SOAP meer complex en minder populair dan REST.

SOAP maakt gebruik van XML voor het sturen van berichten en ondersteunt in tegenstelling tot REST verschillende transportprotocollen zoals HTTP en SMTP. SOAP biedt ook robuuste beveiligings- en transactiebeheerfuncties wat REST niet heeft.[2, 4]

3.2.1 Voordelen van SOAP

1. **Beveiliging** — Sterke beveiligingsfuncties met behulp van WS-Security en ondersteuning voor ACID transacties.

3.2.2 Nadelen van SOAP

Net zoals REST heeft SOAP ook zijn nadelen:

1. **Ingewikkeld formaat** — Complexiteit en overhead door XML-berichten
2. **Prestatieproblemen** — Langzamere prestaties vergeleken met REST

3.3 GraphQL

GraphQL is een querytaal voor API's die is ontwikkeld door Facebook in 2012[5]. In tegenstelling tot REST en SOAP, waarbij de server bepaalt welke gegevens worden geretourneerd, stelt GraphQL clients in staat om een specifieke set data op te vragen. Ook biedt GraphQL een sterke typecontrole en introspectie aan door middel van schemas.[6]

3.3.1 Voordelen van GraphQL

Voordelen van GraphQL:

1. **Efficiëntie** — Door de query aard van GraphQL is het makkelijk om snel en flexibele bepaalde stukken data aan te vragen.
2. **Gegevensbesparing** — Omdat de gebruiker alleen maar de gegevens ontvangt die zij nodig hebben, vermindert dit over-fetching en under-fetching waardoor er meer bandbreedte en rekenkracht wordt bespaard.[6]
3. **Documentatie** — Sterke typecontrole en introspectie door het gebruik van schemas.
4. **Compatibiliteit** — Ook al is het niet gewenst, GraphQL is compatibiel met RESTful clients, hierdoor kan de implementatie vrij simpel zijn als de kracht van GraphQL niet nodig is.

3.3.2 Nadelen van GraphQL

Nadelen van GraphQL:

1. **Intergratie** — Door de aard van GraphQL is de serverimplementatie veel complexer, en daarom ook aangeraden om een library te gebruiken.
2. **Complexiteit** — Mogelijkheid van te complexe queries die de serverbelasting verhogen

3.4 gRPC (Google Remote Procedure Call)

gRPC is een modern RPC-framework dat in 2016 door Google is ontwikkeld. Het maakt gebruik van HTTP/2 voor transport, Protocol Buffers voor berichtserialisatie en biedt functies zoals load balancing en monitoring. Door zijn load balancing functies is gRPC zeer geschikt voor high-performance en real-time communicatie.[7, 2]

Echter is het wel de moeilijkste methoden om te implementeren en te onderhouden.

3.4.1 Voordelen van gRPC

1. **Prestatie** — Hoge prestaties door het gebruik van protocol buffers, berichten zijn tot 30% kleiner dan JSON.
2. **Real-time** — In tegenstelling tot REST, GraphQL en SOAP is gRPC wel geschikt voor real-time communicatie

3.4.2 Nadelen van gRPC

1. **Complexiteit** — Complexer dan alle andere methodes vermeld om op te zetten en te debuggen
2. **Limitaties** — Door het vele gebruik van HTTP/2.0 is het niet compatibel met oudere web browsers. Ook is er veel minder bekend over gRPC dan de andere methodes door zijn relatief nieuwe staat.

4 Methode

4.1 Welke stappen zijn nodig om de API vanaf de grond af aan op te bouwen?

1. **Behoeftenanalyse en Specificatie** — Identificeer de eisen voor de API, maak een lijst met de vereiste functies.
2. **Architectuurontwerp** — Definieer de toegestane request types (GET, POST, PUT, DELETE) op de benoemde routes en hun verwachte responses.
3. **UML ontwerpen** – Maak een klassendiagram voor de API om de structuur van de code te visualiseren.
4. **Implementatie** — Bouw de API volgens de opgegeven eisen en UML van punt 1, 2 en 3 op.
5. **Testen** — Ontwikkel unit tests voor alle componenten die door de API worden gebruikt. Verwerk vervolgens ieder probleem dat opkomt.
6. **Onderhoud** — Monitor de API na de implementatie, pas deze aan op basis van feedback na gebruik. Voeg nieuwe functies toe en verbeter de bestaande functies waar nodig, zonder de bestaande functionaliteit te breken.

4.2 Hoe kunnen REST, SOAP, GraphQL, en gRPC worden vergeleken op basis van de gestelde criteria?

1. **Vergelijkingscriteria** — Bepaal de evaluatiecriteria waarop de architecturen vergeleken zullen worden, zoals prestaties, schaalbaarheid en flexibiliteit.

2. **Literatuuronderzoek** — Voer een literatuuronderzoek uit naar de kenmerken, voor- en nadelen van REST, SOAP, GraphQL, en gRPC.

3. **Theoretische Analyse** — Analyseer elke API-architectuur op basis van de gedefinieerde projectscriteria. Maak een overzicht van de sterke en zwakke punten van elke architectuur in de context van het robotica project.

4. **Vergelijking** — Vergelijk de resultaten van de analyse. Weeg de voor- en nadelen van elke API-architectuur af tegen de specifieke eisen van het project.

5. **Conclusie** — Trek een conclusie over welke API architectuur het meest geschikt is voor het project op basis van de vergelijking.

5 Resultaten

5.1 Stappen voor het opbouwen van de API vanaf de grond af aan

Voor de ontwikkeling van de API zijn de volgende stappen doorlopen:

1. **Behoeftenanalyse en Specificatie** — De eerste stap was het identificeren van de eisen voor de API. De volgende eisen zijn opgesteld na aanleiding van de behoeftenanalyse:

   (a) **Basisfunctionaliteit**
   - Ondersteunen van communicatie tussen de gebruiker en de robot.
   - Mogelijkheid om de robot te besturen en inzicht te krijgen via een webinterface.

   (b) **Technische Specificaties**
   - De API moet worden opgebouwd zonder het gebruik van externe libraries.
   - Gebruik van standaard HTTP-methoden (GET, POST, PUT, DELETE).
   - Het gebruik van JSON voor gegevensoverdracht.

   (c) **Flexibiliteit en Onderhoudbaarheid**
   - Eenvoudige structuur die gemakkelijk te onderhouden is.
   - Flexibel genoeg om toekomstige toevoegingen en aanpassingen te ondersteunen zonder dat de hele codebase herschreven moet worden.

   (d) **Gebruikersinterface**
   - Weergave van verzamelde data in een overzichtelijke manier.
   - Presentatie van de mapping en de geplande route van de robot.

- Mogelijkheid om diverse commando's naar de robot te sturen.

2. **Architectuurontwerp** — Vervolgens zijn de toegestane request types (GET, POST, PUT, DELETE) gedefinieerd, samen met de benoemde routes en hun verwachte responses. Dit heeft geresulteerd in een gedetailleerd ontwerp van de API-structuur.

   - **GET** **/api/v1/database/metadata**
     Geeft basisinformatie over de database terug.
   - **GET** **/api/v1/database**
     Query Paramters:
     - ?id=<string> — De UUIDv4 van de gewensde map.
     - ?name=<string> — De naam van de gewensde map.
     - ?version=<int> — Het versienummer van de gewensde maps
     - ?all=<bool> – Laat alle maps zien

     Return Body:

```
[
  {
    "Id": "<string>",
    "Name": "<string>",
    "Version": "<int>",
    "Objects": "<Array<Array<int, int>>"
    "Date": "<DateTime>"
  }, {...}
]
```

     Geeft een lijst van alle maps terug, of een specifieke map op basis van de query parameters.
   - **POST** **/api/v1/database**
     Request Body:

```
{
  "name": "<string>",
  "objects": "<Array<Array<int, int>>>"
}
```

     Return Body:

```
{
  "message": "<success|fail>",
  "id": "<string>",
}
```

     Voegt een nieuwe map toe aan de database.
   - **DELETE** **/api/v1/database**
     Query Paramters:
     - ?id=<string> — De UUIDv4 van de gewensde map.

– ?name=<string> — De naam van de gewensde map.

Verwijdert de desbetreffende map uit de database.

- **GET** **/api/v1/database/path**
  Query Paramters:
  – ?id=<string> — De UUIDv4 van de gewensde route.
  Return Body:

```
{
  "Id": "<string>",
  "Objects": "<Array<Array<int, int>>"
}
```

- **POST** **/api/v1/database/path**
  Request Body:

```
{
  "id": "<string>",
  "objects": "<Array<Array<int, int>>>"
}
```

Return Body:

```
{
  "message": "<success|fail>",
  "id": "<string>",
}
```

Voegt een nieuwe geplande route toe aan de database.

- **DELETE** **/api/v1/database/path**
  Query Paramters:
  – ?id=<string> — De UUIDv4 van de gewensde route.

Verwijdert de desbetreffende route uit de database.

- **POST** **/api/v1/database/path/plan**
  Request Body:

```
{
  "id": "<string>",
  "objects": "<Array<Array<int, int>>>"
}
```

Return Body:

```
{
  "message": "<success|fail>",
  "id": "<string>",
}
```

Voegt een nieuwe geplande route toe aan de database.

- **POST** **/api/v1/roomba/control**
  Request Body:

```
{
  "command": "<string>"
}
```

Proxies het commando naar de Roomba.

3. **UML ontwerpen** — Een klassendiagram is gemaakt om de structuur van de code te visualiseren. Dit diagram is te vinden in Figuur 2.

   De UML is opgedeeld in drie delen: de HTTP server, het data parsen en de endpoints. De HTTP server is verantwoordelijk voor het ontvangen van de requests en het doorsturen naar de juiste endpoint. Het data parsen zorgt ervoor dat de data correct wordt verwerkt en opgeslagen. De endpoints zijn de verschillende routes die de API aanbiedt en de bijbehorende acties die worden uitgevoerd.

4. **Implementatie** — De API is opgebouwd volgens de specificaties van de vorige stappen. Er is gebruik gemaakt van Csharp zonder externe libraries om de functionaliteiten van de API te implementeren.

5. **Testen** — Er zijn unit tests ontwikkeld voor alle componenten die door de API worden gebruikt. Eventuele problemen die tijdens het testen naar voren kwamen zijn verwerkt en opgelost.

6. **Onderhoud** — Na implementatie is de API gemonitord en aangepast op basis van feedback. Nieuwe functies zijn toegevoegd en bestaande functies zijn verbeterd waar nodig, zonder de bestaande functionaliteit te breken.

5.2 Vergelijking van REST, SOAP, GraphQL, en gRPC op basis van gestelde criteria

De API architecturen zijn vergeleken op basis van de volgende criteria: prestaties, schaalbaarheid, flexibiliteit, onderhoudbaarheid, en geschiktheid voor de projectseisen. Hieronder volgt een samenvatting van de resultaten verzameld in paragraaf 3.

5.2.1 REST

- **Voordelen**: REST heeft een eenvoudige structuur, maakt gebruik van standaard HTTP-methoden en JSON, en is gemakkelijk te implementeren. Het is goed schaalbaar door zijn stateless karakter.
- **Nadelen**: REST kan overbodige data teruggeven, is minder geschikt voor complexe query's, en biedt geen directe ondersteuning voor documentatie en real-time communicatie.

5.2.2 SOAP

- **Voordelen**: SOAP biedt beveiligingsfuncties en ondersteuning voor ACID-transacties. Het maakt gebruik van XML voor berichtuitwisseling en ondersteunt verschillende transportprotocollen.
- **Nadelen**: SOAP heeft een ingewikkeld formaat door de XML-berichten, wat resulteert in een hogere complexiteit vergeleken met REST waar meerdere typfouten op kunnen treden.

5.2.3 GraphQL

- **Voordelen**: GraphQL biedt efficiëntie door de mogelijkheid om specifieke sets data op te vragen, wat over-fetching en under-fetching vermindert. Ook biedt GraphQL type controle door middel van schema's.
- **Nadelen**: De serverimplementatie van GraphQL is complexer, waardoor het moeilijker kan zijn om zonder externe libraries te implementeren.

5.2.4 gRPC

- **Voordelen**: gRPC biedt hoge prestaties door het gebruik van Protocol Buffers, en is zeer geschikt voor real-time communicatie door HTTP/2. Het biedt ook functies zoals load balancing en monitoring.
- **Nadelen**: gRPC is complexer op te zetten en te debuggen, en heeft limitaties door het gebruik van HTTP/2, wat niet compatibel is met oudere web browsers. Ook heeft gRPC net als GraphQL een complexe serverimplementatie.

6 Conclusie

De hoofdvraag van dit onderzoek luidt:

> " *Welke API architectuur is het beste geschikt voor de specifieke eisen van het project, waarbij de communicatie tussen verschillende softwarecomponenten, de ondersteuning van gebruikersinteracties, en de flexibiliteit en onderhoudbaarheid van de code optimaal zijn?*"

Voor het uiteindelijke project zal er gebruik worden gemaakt van **REST** als API architectuur. REST is de meest geschikte keuze voor het project, omdat het voldoet aan de eisen van het project en het ontwikkelen van de API zonder externe libraries mogelijk maakt. REST is eenvoudig te implementeren en biedt een breed scala aan mogelijkheden voor communicatie tussen verschillende componenten.

Om deze hoofdvraag te beantwoorden, zijn de volgende deelvragen geformuleerd:

> "*Welke stappen zijn nodig om de API vanaf de grond af aan op te bouwen?*"

Uit de ontwikkeling van de API blijkt dat er geen nood is voor specifieke dataselectie, wat GraphQL juist zo sterk maakt. Ook is er geen nood voor real-time communicatie of authenticatie.

> "*Hoe kunnen REST, SOAP, GraphQL, en gRPC worden vergeleken op basis van de gestelde criteria?*"

Het project stelt een aantal eisen aan de API-architectuur, waaronder het ontwikkelen zonder externe libraries. Uit paragraaf 5 blijkt dat **REST** de meest geschikte keuze is op basis van de gestelde criteria.

Erkenningen

De auteur wilt graag de volgende mensen bedanken voor hun contributie bij het schrijven van dit verslag:

Buurman, W. J.	Groepsgenoot
Slikker, T.	Groepsgenoot
Ottens, N.	Opdrachtgever
Tilmann, K.	Opdrachtgever

Referenties

[1] L. M. Afonso, R. F. de G. Cerqueira, and C. S. de Souza, "Evaluating application programming interfaces as communication artefacts," 2012, geraadpleegd op Juni 6, 2024. [Online]. Available: http://www3.serg.inf.puc-rio.br/docs/MarquesPPIG2012.pdf

[2] M. Śliwa and B. Pańczyk, "Performance comparison of programming interfaces on the example of rest api, graphql and grpc," *Journal of Computer Sciences Institute*, vol. 21, Dec. 2021, geraadpleegd op Juni 10, 2024. [Online]. Available: https://ph.pollub.pl/index.php/jcsi/article/view/2744

[3] M. Massee, *REST API design rulebook designing consistent restful web service interfaces mark massee. ed.: Simon St. Laurent.* OReilly, 2011, geraadpleegd op Juni 8, 2024. [Online]. Available: https://books.google.nl/books?id=eABpzyTcJNIC&lpg=PR3&ots=vBPD3-ldKC&dq=restful%20api&lr&hl=nl&pg=PP1#v=onepage&q=restful%20api&f=false

[4] W. W. W. C. W3C, “Soap version 1.2 part 1: Messaging framework second edition,” geraadpleegd op Juni 15, 2024. [Online]. Available: https://www.w3.org/TR/soap12-part1/

[5] Facebook, “Graphql: A query language for your api,” geraadpleegd op Juni 14, 2024. [Online]. Available: https://graphql.org

[6] O. Hartig and J. Perez, “Semantics and complexity of graphql,” in *Proceedings of the 2018 World Wide Web Conference*, ser. WWW ’18. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2018, p. 1155–1164, geraadpleegd op Juni 12, 2024. [Online]. Available: https://doi.org/10.1145/3178876.3186014

[7] Google, “grpc - a high-performance, open source universal rpc framework,” geraadpleegd op Juni 16, 2024. [Online]. Available: https://grpc.io/

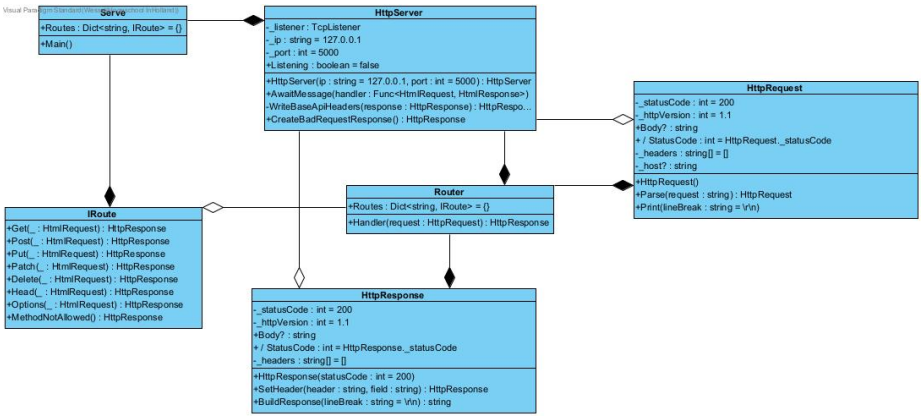
7 Bijlage

User Story

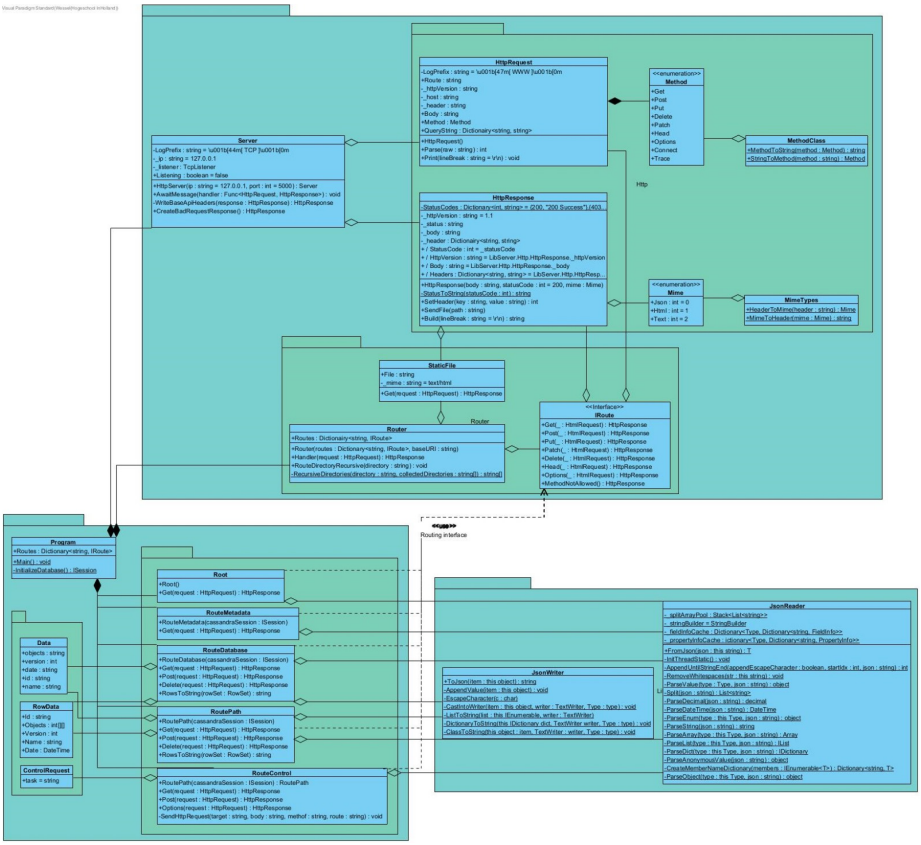
Nadat de robot de ruimte in kaart heeft gebracht en de optimale route bepaald heeft, moet dit allemaal op een manier toegankelijk worden gesteld aan de gebruiker.

Dit zal gedaan worden d.m.v. het hosten van een web interface op een aparte "server PI". Deze PI zal vervolgens ook een manier van communicatie opstellen met de roomba PI om vervolgens hun informatie met elkaar te delen.

UML modellen



Figuur 1: 1e versie UML model.



Figuur 2: Uiteindelijke versie UML model.