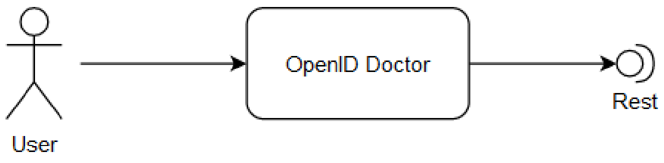
Software Architecture

# OpenID Connect Doctor

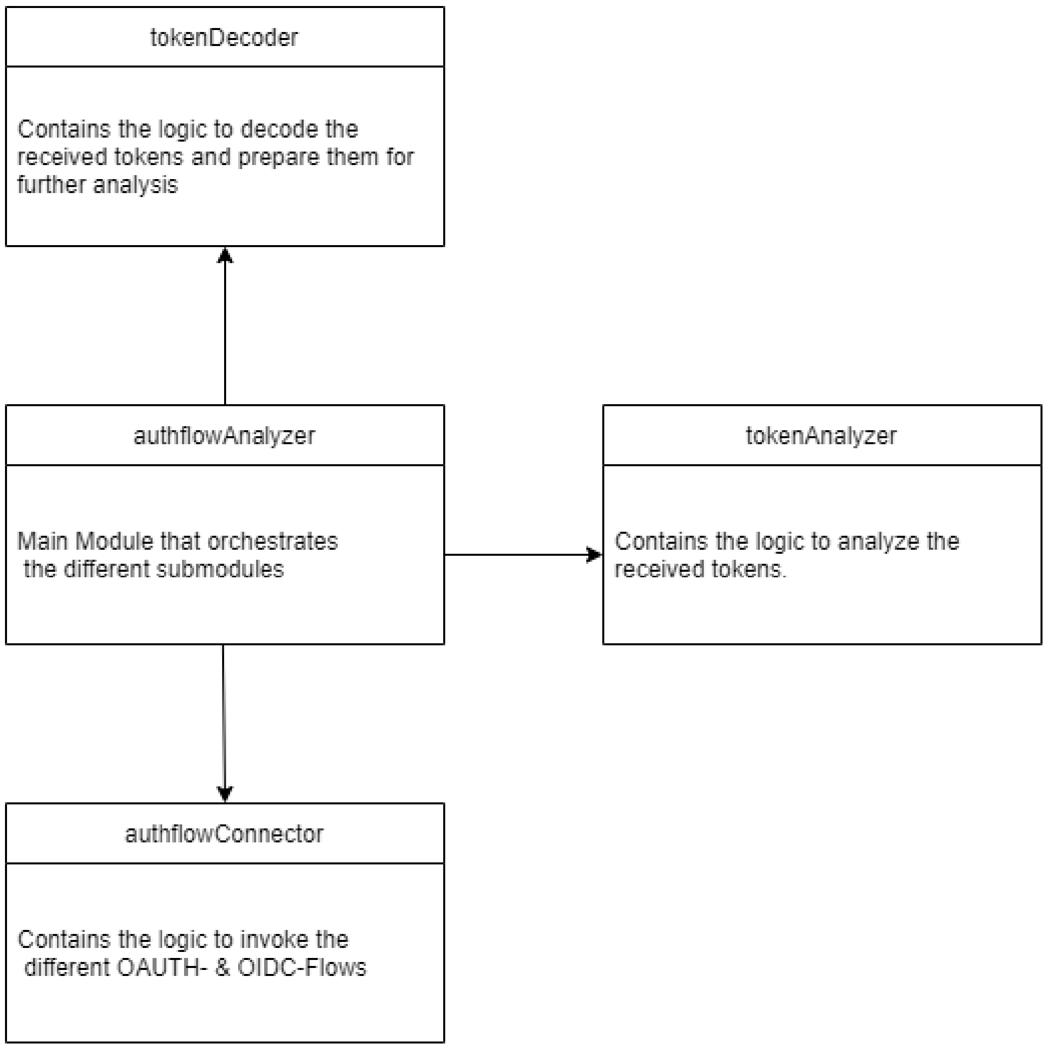
Project 8

## 1 Runtime Components



The OpenID Connect Doctor is a standalone CLI tool, that should be able to run on as many Operating Systems as possible. Furthermore, the application is required to work without the user having to install any additional programs. For this reason, it was decided, that the OpenID Connect Doctor should consist of only a single NodeJS application and use as few dependencies as possible. As the OpenID Connect Doctor must request tokens from an authorization server using OpenID Connect, it requires an interface to connect to a given authorization server using REST. To fully fulfill its task, the OpenID Connect Doctor is split into multiple smaller internal components, which carry out different tasks like requesting a token, decoding a token, etc.

## 2 Code Components


tokenDecoder
Contains the logic to decode the received tokens and prepare them for further analysis
authflowAnalyzer
Main Module that orchestrates the different submodules
tokenAnalyzer
Contains the logic to analyze the received tokens.
authflowConnector
Contains the logic to invoke the different OAUTH- & OIDC-Flows

The code of the application will consist of multiple smaller classes, that each are responsible for a certain part of the application:

**authflowConnector** : This class will contain the logic to send HTTP-requests to the different Identity-Providers and process the returned HTTP-response (e.g. for the different OIDC-Login-Flows).

**tokenDecoder** : This class will contain the logic to decode and transform the received access and id-tokens into a format that can be analyzed by the rest of the application.

**tokenAnalyzer** : This class will contain the logic to analyze the prepared information and report these information to the user (e.g. via the terminal or a logfile)

**authflowAnalyzer** : This class will reference the other three class and will be responsible for orchestrating the different functions. Additionally it will also include input-validation, global error-handling and configuration.

The reason for this split is to keep a certain level of separation between the different parts of the software, so it is easier to modify and test them.

## 3 Technology Stack

| **Type** | Programming Language | | |
|---|---|---|---|
| **Name** | **Description** | **Version** | **License** |
| NodeJS | Programming Language used to implement the application | 16.15.0 | MIT License |

| **Type** | Frameworks | | |
|---|---|---|---|
| **Name** | **Description** | **Version** | **License** |
| NestJS | Fullstack Development Framework for efficient, scalable Node.js server-side applications | 8.4.4 | MIT License |
| Fastify | Node.js Framework to receive fast real-time applications running in NestJS intead of Express | 4.0.0 | MIT License |

| **Type** | Package Manager | | |
|---|---|---|---|
| **Name** | **Description** | **Version** | **License** |
| npm | Package Manager is used in combination with the Programming Language to potentonally add additional libraries to create the application | 8.5.5 | Artistic License 2.0 |

| **Type** | Version Control | | |
|---|---|---|---|
| **Name** | **Description** | **Version** | **License** |
| Git | Version Control employed to syoncronize to code with the developers | - | GPL-2.0-only |

The customer has suggested nodejs as the preferred language, because it has been used very much in their own technology stack. nodejs 16 is an LTS version with long-term support. Additionally it is available for Windows and openSUSE Tumbleweed (Linux). That is the reason for this suggestion, that the software can be developed based on multiple operating systems.
The package manager nmp 8.5.5 is integrated into nodejs 16. Therefore, we are using this version.
We have chosen NestJS because it has got additional features and libraries integrated for authentication. It is usable via cli and it is also possible to create frontends with it. Our application should be also applicable in real-time in the best case. NestJS is configured based on the framework Express as a default, but it is compatible with Fastify. Fastify can integrate the speed for real-time applications into NestJS, that the user does not have to wait for the response such a long time.
NestJS and Fastify are both well maintained nodejs frameworks, which are also used for Enterprise open-source applications. Additionally, there is also a lot of documentation available, how to use it (incl. books).