ZAUTHLY: A ZERO TRUST OAUTH2 AUTHORIZATION TOOL

By

Matthew Robert Perry

A Project Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

in

Computer Science

University of Alaska Fairbanks

May 2022

APPROVED:

Dr. Orion Lawlor, Committee Chair
Dr. Glenn Chappell, Committee Member
Dr. Jon Genetti, Committee Member
Dr. Jon Genetti, Department Chair
Department of Computer Science

Abstract

Controlling authentication and authorization is a pivotal part of managing modern web resources. Over the past decade, Oauth and OpenID Connect have shown that they are capable and secure protocols used for secure communication between the Identity Providers (IdP) and requesting parties that consume them. Zero Trust (ZT) architectures are based on authenticating individual requests instead of machines or networks. ZT has shown a pathway that enables a more secure flow of trusted communication. This is done by defining the control systems and their counterpart the data systems. Zauthly applies ZT principles to Oauth2 flows to create a middleware service that solely controls the authorization of users. It aims to enable increased security in existing tools and control flows while it utilizes Google as an IdP to enable authentication of end users. A Single Sign On (SSO) proxy is used to consume the provided Oauth2 authorization from Zauthly. Then its users are managed by a simple interface that communicates with a user database. Zauthly is designed to be deployed in a modular way drawing inspiration from the microservice architectural style. Its deployment is controlled by Docker and Docker-Compose to provide enhanced scalability and flexibility. This paper will explore the design choices of Zauthly, relevant drawbacks, and performance of the tool.

Acknowledgements

I would like to thank Dr. Orion Lawlor, Department of Computer Science, University of Alaska Fairbanks, Dr. Glenn Chappell, Department of Computer Science, University of Alaska Fairbanks, and Dr. Jon Genetti, Chair of the Department of Computer Science, University of Alaska Fairbanks, for their guidance and technical knowledge. I would also like to thank my parents, Micheal Perry Sr. and Pearl Perry, for supporting me throughout my academic career. Lastly, I would like to thank my girlfriend, Jenna VanDenHeuvel for her support.

Table of Contents

Introduction

The World-Wide Web was first proposed in the paper *The World-Wide Web* by Berners-Lee et al in 1994. The web was developed to be a pool of human knowledge which would allow collaboration of many people across many different regions of the world, to share ideas (Berners-Lee, 1994). The W3 consortium sought to define the standards for how the web functioned and it included definitions of Universal Resource Identifiers (URI), Hypertext Transfer Protocol (HTTP), and Hypertext Markup Language (HTML), all of which remain vital parts of the web today. HTTP is a protocol for transferring information between clients and servers. HTTP defines important operation codes GET, PUT and POST as the foundation for how communication occurs between server and client. These are the building blocks of how web applications interact with one another.

In 1996, "*Authentication systems for secure networks*" (Oppliger, 1996) a book on authentication in secure networks was published. This book explored the current protocols for authentication outside of the traditional username and password. This included Kerberos, NetSP, SPX, TESS, and SESAME. With the increased use of HTTP for sensitive applications, it was clear that a secure method for transmitting HTTP was required. In 2000, the Internet Engineering Task Force (ITEF) released a Request For Comment (RFC) on HTTP over TLS. RFC2818, (Rescorla, 2000) defined the practice of communicating HTTP using Transport Layer Security (TLS). This became known as HTTPS. This defines the TLS handshake where a client and server would begin to communicate and share the details of how the secure communication will be handled. Once the handshake is complete, the session would begin between the two parties on the agreed upon port and cipher.

Oauth1 is a protocol that was released by the IETF in April of 2010. This is a protocol that provides a "method for clients to access server resources on behalf of a resource owner" (IETF, 2010). Oauth1 provides the process for end-users to authorize third-party applications and services to access their server resources without sharing credentials. This standard was published as RFC5849, which defines important terminology for client, server, protected resource, and resource owner. Client is an HTTP client capable of making Oauth1 requests. Server is an HTTP server capable of accepting Oauth1 requests. Protected resources are an access restricted resource such as a website. The resource owner is an entity capable of accessing and controlling protected resources to authenticate with a server. Oauth1 also introduces the idea of **token based authentication**, which is a token that is uniquely created for a user. The server creates tokens to represent the user and validate they have completed the authentication process. These tokens are commonly JSON objects that contain information on subject, token expiration time, and nonce.

In the years following the original paper from the W3 consortium, there have been many changes to how users and services interact with the web. The landscape has continued to evolve with new

protocols and new revisions of old protocols. They all share the same fundamental goal of increasing security of the web.

Prior Work

The web has continued to evolve over the last twenty years, the modern web is full of applications that require users to verify their identity and also have permissions for what resources they can access assigned. The processes are known as authentication and authorization and many products have been proposed as solutions for these processes. These products include Microsoft's Active Directory, Okta's Cloud Identity and Access Management, Jumpcloud's Domainless Enterprise, and Microsoft's Azure Cloud Active Directory. This section covers some basic information about these systems and some potential drawbacks.

Active Directory (AD) was created by Microsoft in 1999 to address the need for authentication and authorization within enterprise domains. It provides a structure for managing organizational units, computers, groups, and users. AD is a LAN-native service which is deployed on an On-Premise Windows server. The server has a tree-like directory structure that provides the ability to control users' permissions based attributes including groups within organizational units. Kerberos (Steiner, 1988) was the original protocol used to communicate between Microsoft systems and is a network authentication service for computing environments. This authentication is based on a ticket system. It provides more than a user's identity and talks about authentication of machines and file systems. AD supports the use of a protocol to query the service using Lightweight Directory Access Protocol (LDAP). LDAP, (IETF, 2006) is defined in the RFC4511 as the method for querying tree-like directory structures. The LDAP protocol relies on utilizing abstract syntax notation to encode the data that is in the directory structure. This protocol is core to the way AD is interacted with and information is found about a user or computer. Because AD service was originally designed as LAN-native, work has been done to enable its use for cloud based applications. One major update was the support of Security Assertion Markup Language (SAML) as a way to issue access tokens for authentication flows. SAML (Campbell, 2015) is defined by RFC7522 and this protocol is used to manage the assertions about users so that an access token may be provided to a requesting party. The privacy and security concerns around the protocol shaped a change in SAML to use TLS and a RSA-sha256 signature for data transmission. The authentication flow for cloud based applications rely on SAML to provide the user information and access token. The major drawbacks of AD are the LAN-native design and the blurring of the separation of responsibility between authentication and authorization. AD has a single point of failure which poses a greater security risk to the enterprise, as the system controls all users and their permissions for integrated systems and applications.

Okta maintains several services that provide SSO, identity management, and authorization. The company provides a cloud based product that looks to provide authentication and authorization solutions. As outlined in Okta's white paper (Okta, 2000), their product is called Cloud Directory. One of the tools is the management of users and groups in a web interface. The cloud directory provides two pathways to configure an authorization flow to a login resource. These pathways include a web interface for simple integration and a developer API. Okta's Cloud Directory provides a place to store users and assign permissions for these users. The authentication and authorization steps are from a user's perspective tightly coupled, as they are both happening inside the Okta product. Okta explains that the Cloud-native design of the product is important for a globally available authentication and authorization service. The security considerations of this project are known as a shared security responsibility model. The model defines the service, infrastructure, and physical security to be taken care of by Okta. The customer application and content as well as service settings are handled by the tenant. This model is a great option for some applications, but an enterprise must accept the risk and cost of passing part of the security responsibility along to Okta.

Jumpcloud is another cloud based product that is looking to move away from the traditional approach of AD. In the whitepaper *Roadmap to the DOMAINLESS enterprise* (Jumpcloud, 2022), they outline the key parts of a modern authentication and authorization protocol that migrates from AD. Jumpcloud provides services to interface into existing authentication and authorization flows that are used by legacy AD based applications. The domainless service can provide flexibility and increased security, by moving from the centralized truth that AD provides, to the cloud based directory where each user has unique permissions. Jumpcloud currently provides migration of an AD system to the cloud by supporting SAML based applications. This product is a pay to use service and is currently only supporting the SAML authentication flow. Jumpcloud is still in active development to further support additional protocols like Oauth2 in the future.

Azure Cloud AD outlined in the white paper by Microsoft, *Azure Active Directory, Identity and Access Management, and Windows 10* (Madden, 2016) looks to address the downsides of the LAN-native AD system by providing the directory service as a cloud service. Azure AD provides integrations to cloud directory services for traditional LDAP and SAML applications while expanding the abilities to support OpenID and Oauth2 authentication and authorization. Identity and Access Management known as IAM is identified as the core of the service that Azure AD provides. IAM provides a similar role that an AD system does by providing a tree-like structure of users with the capability of complex policies to manage a user's permissions on a resource. The Azure Cloud AD looks to provide a Cloud-native AD that can provide traditional AD services as well as modern authentication and authorization services like Oauth2 and OpenID. While it provides these additional services, the cost and security risk associated with a single authentication and authorization solution are the primary drawbacks of its design.

Motivation for Zauthly

The primary motivation for the creation of Zauthly is to fill the need for a simple open source tool that allows developers to easily integrate authentication and authorization into web applications. It was found that the common tools that already exist rely on a single step to complete this process. Clients normally can only be configured to accept Oauth2 from a single IdP like Google. The IdP has the sole control of the configuration of the provided Oauth2 client and token. This means a developer can not configure properties of the token or client manually.

Drawing inspiration from the National Institute of Standards and Technology (NIST) ZT architecture, a new step can be added in the flow of authentication and authorization. Users must prove who they are to an IdP during the authentication step and access will be checked during the separate authorization process to determine if a user will be granted access to a restricted resource. Zauthly looks to create a tool that can provide authorization on generic web sources with an existing IdP and existing SSO client. It can be classified as a middleware tool that fits in between existing tools in the rich authentication and authorization ecosystem. While being free and open source, Zauthly looks to provide flexibility and security while not negatively impacting the user experience.

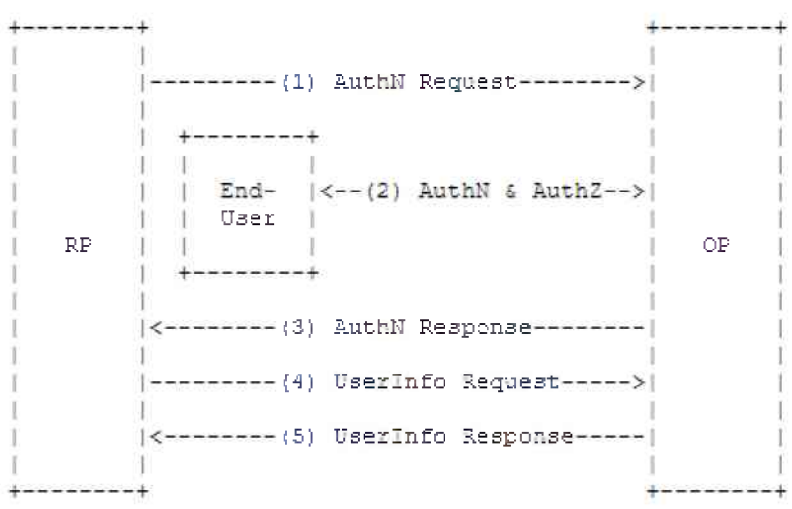
Background

This section will cover all of the relevant information about the different technologies used when designing and implementing Zuathly. These include protocols, architectures, libraries, and tools that made the proof of concept possible.

In October of 2012 **Oauth2**, (IETF, 2012) was released under RFC6749 standard deprecating Oauth1. Oauth2 was an extension of the client server authentication flow that allowed users to authorize third-party applications. The major change in the protocol was extending its flow to include different authorization grants and tokens. An **authorization grant** is defined as “a credential representing the resource owner's authorization (to access its protected resources) used by the client to obtain an access token”(IETF, 2012). The **access token** is the replacement for the traditional username and password based authentication. Oauth2 defines an access token as “Access tokens are credentials used to access protected resources. An access token is a string representing an authorization issued to the client.” (IETF, 2012). The full Oauth2 flow is shown in the figure below.

Once a Client makes a request for a restricted resource, the Resource Owner provides an authorization grant. This grant is communicated to the authorization server which then issues an access token. The access token is consumed by the Resource Server to provide scopes and claims that are associated with the user. **Scopes** are broad categories of claims that provide the meta information about a user. **Claims** include information like email address, name, address, website, gender, and picture. There are several claims that are reserved but additional claims can be defined by developers when deploying authorization servers.

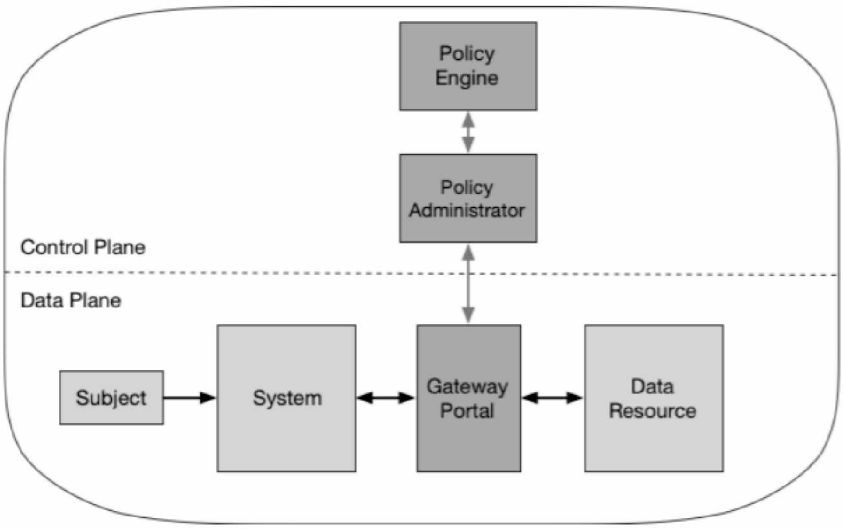
OpenID connect is an extension of the Oauth2 flow that provides identity tokens to the clients utilizing JSON web tokens (JWT). **JWT** is a standard way of encrypting a JSON token, this includes the protocol for how the communication is set up, and which algorithm is used to encrypt and decrypt it. OpenID provides an identity layer onto the Oauth2 protocol as well as minor changes to the flow shown in the figure below.



The OpenID extension introduces a new scope category called profile. The profile scope was designed to provide all the general information about a user. This is a key part of authorization and can be provided by an IdP like Google. Google provides a way to create an Oauth2 server using the Google Cloud Platform. The IdP acts as the authentication server for the Oauth2 flow. The Oauth2

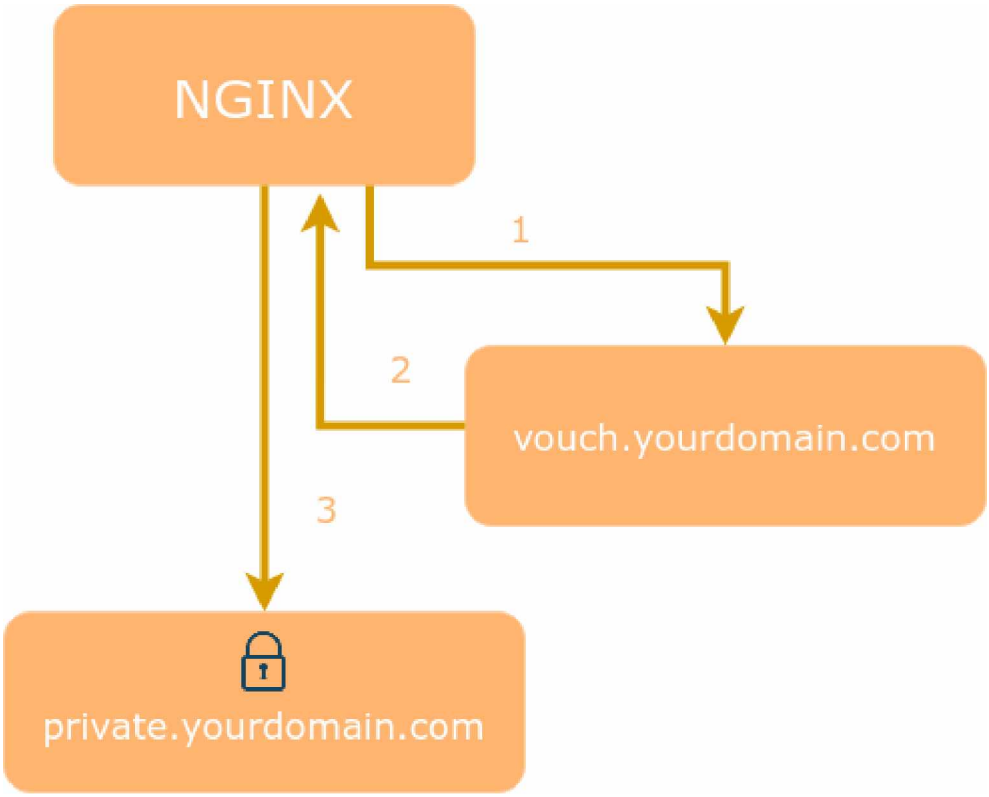
and OpenID standards are used heavily in Zauthly to handle the identity from the authentication step and the permissions in the authorization process.

Zero Trust (ZT) Architecture was proposed in a NIST special publication 800-207 (Rose, 2020), it outlines the general idea of ZT while providing several examples of ZT architectures. ZT is a cybersecurity paradigm that moves defense from a static perimeter focus, i.e the network, to the specific users and resources of a system. ZT assumes that there is no implicit trust granted to assets or users regardless of their physical or network location. This includes asset ownership, for example the CEO's computer does not gain the ability to bypass security just because it is owned by the CEO. ZT further describes the authorization and authentication steps as discrete functions performed by clearly separate operations. This is to reduce the risk associated with a system that performs both authentication and authorization like Active Directory.



There are three main logical components of the ZT architecture shown in the figure above, the Policy Engine (PE), Policy Administrator (PA), and Gateway Portal sometimes called Policy Enforcement Point (PEP). The PE is responsible for the ultimate decision to grant access to a resource for a given subject or user. The PA generates any session-specific authentication and authorization token or credential used by a client to access an enterprise resource. PEP/Gateway system is responsible for enabling, monitoring, and eventually terminating connections between a subject and an enterprise resource. The specific architecture example that was used in Zauthly is the Resource Portal Model (RPM). This model is a "Bring Your Own Device" (BYOD) and relies on no additional software or hardware to be installed on users computers. It brings more flexibility to what users are allowed to use to connect to the restricted resource. This does have a drawback, as with greater flexibility, there is less information that can be fed to the PE that would enable enhanced control logic. Because of the lack of full system visibility or arbitrary control of the user's system, RPM does not have the ability to be as secure as other ZT models.

Vouch SSO Proxy is an Oauth2 client that can be configured and deployed to act as a cookie based SSO web proxy. The proxy can be configured to be a standalone tool or to be deployed alongside a web server. It is designed to be used with Nginx using the auth_request module.



An example flow using Vouch is the following: the incoming web connection hits your webserver, example.com and is forwarded to Vouch (vouch.yourdomain.com) so it may follow the pre-configured Oauth2 client server communication. It provides the ability to be configured to support a generic Oauth2 client and to interface directly with common Oauth2 providers such as Google, GitHub, Azure AD, and AWS Cognito. It can handle custom scopes and claims as it reads the scopes provided and can set session cookies with the values of these custom claims. Nginx then confirms that the cookie is set and provides access. Zauthly will be using the generic OpenID provider and will be configured to read the custom claims and set the session cookies.

Microservices, (J. Thönes, 2015) is defined in the IEEE publication as an architectural style that focuses on a small application that can be deployed independently, scaled independently, and tested independently. Another important feature of microservices is they should follow the single responsibility principle. This is an important distinction which was strongly considered when designing the architecture of Zauthly.

Flask is a small framework described by Grinberg (Grinberg, 2018) as a micro framework. It is made up of three main dependencies: routing, debugging and Web Server Gateway Interface (WSGI). These are provided by Jinja2, a templating engine, and Werkzeug, a WSGI. Flask is

written in Python and is an object oriented approach to a WSGI. Zauthly utilizes the Flask framework to build the main authorization application because of Flask's extensive list of open source libraries. Which provided the ability to speed up development of the tool.

FastAPI is a modern web framework (Ramírez, 2018) for building APIs with Python 3.6 or greater. Its key functionality is to provide an efficient and performant Asynchronous Server Gateway Interface (ASGI). The framework uses strong type hinting and function decorators to automatically generate API documentation by leveraging Swagger-UI. FastAPI was benchmarked in February of 2021 by TechEmpower and was placed in 11th place for response time handling 20 queries per request. Zauthly utilizes this framework to design and build the strongly typed database API.

Pydantic is a Python library (Colvin, 2022) that is used for data modeling and validation using Python type annotations. Pydantic is able to enforce type hints at runtime which is used to validate the data. Pydantic is an important part of validating the data that is received in Zauthly's Database API. The strict validation enables a guarantee of safety when communicating data to and from the database.

MongoDB is a document focused database that is NoSQL. The structure of the database is JSON like documents that are stored in collections. Data can be mapped to objects in application code and provide flexible query options. Mongodb is a distributed database that provides high availability and horizontal scalability. Zauthly uses the NoSQL MongoDB driver as the core of the Database API.

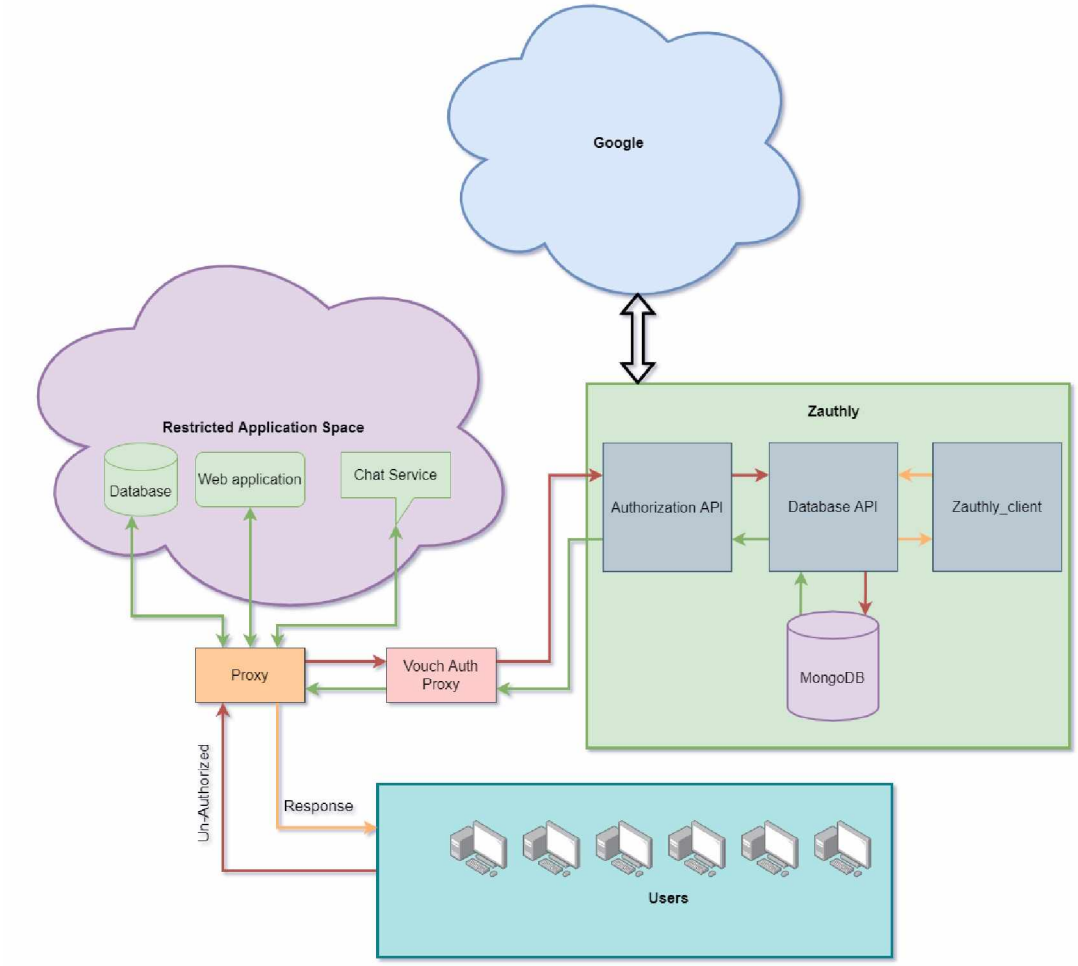
Google Cloud Platform (GCP) is a cloud provider that enables many different applications to be deployed and was created by Google. The primary use of GCP with Zauthly is providing an API endpoint for Identity. The Google Oauth2 client creation process enables the creation of a channel of communication that is used to validate who a user is. The profile of the logged in user is passed to Zauthly and used to look up the user.

Design

Zauthly's key components have been designed utilizing two important architecture styles known as Microservice and Zero Trust (ZT). The ZT architecture was an important design decision that was motivated by the idea that the system's security could be clearly defined by leveraging this design. The specific ZT architecture that was drawn for inspiration for Zauthly is the Resource Portal Model. This model provides a mixture of flexibility and security to Zauthly. The primary idea that needed to be incorporated was the clear separation of responsibility between the processes that make up authentication and authorization. To achieve this authentication is handled by a 3rd party IdP. The IdP used in the proof of concept for Zauthly was Google. By

leveraging IdP for authentication, Zauthly does not store sensitive information like passwords. The IdP provides a token that provides a proof of authentication from Google. The access token is read and used in the session to complete the additional control logic of Zauthly in the authorization step. The transmission of the token from Google follows Oauth2 flow by configuring Zauthly as a client to the Oauth2 server Google. The token is shared and signed by a predetermined secret, which is configured when setting up the Google Client. The communication and storage of the token is achieved by utilizing secure sessions. The session is encrypted and the session encryption key is dynamically created per deployment.

Because Zauthly has a clear separation of responsibilities, designing the components of Zaulthy naturally fit into the microservices architecture. Each of the system's key components are designed to be able to be replaced as needed. Additionally the Microservice architecture enables the ability for potential horizontal scaling. The key components of Zauthly are the Authorization API, Database, Database API, and Zauthly_client interface. Zauthly's architecture is shown in the figure below.



Zauthly's authorization API is written in Python using the Flask web framework. The choice to use Flask and Python was motivated by the existence of a well documented and complete Oauth2 server framework, called Authlib. Authlib provides generic implementations of important RFCs for Oauth2, OpenID, and JWT. It is an object oriented approach that provides a skeleton of the required parts of the server which give the flexibility where the RFC standard for Oauth2 allows. Authlib provides some integrations for Flask so that secure sessions and requests can be handled within the server object implicitly. Authlib was key to increasing the ability of Zauthly to meet the authorization standard that is required for Oauth2 standard flows, which enabled simple integration of existing tools like Vouch and Google Oauth2 API. The flexibility that Authlib enabled was the implementation details of how clients, tokens, authorization codes, and authorization grant types are defined. Zauthly's Authorization API only supports the authorization code grant type. This grant type is key to enabling a secure server to server communication that does not allow a user to manipulate any parts of the authorization process after a user has logged in. This does come with some drawbacks that Zauthly will not be compatible with Oauth1 applications as this grant type is not included in the standard. Zauthly determined that this is an acceptable drawback as the supported grants for Oauth1 include password based grants which is a pattern that Zauthly is choosing to avoid due to increased risk. The creation of clients is not to be done by the Authorization API as it is strictly a provider for Oauth2 clients. The Authorization API has a strict role and only verifies that users are logged in and communicates the authorization of a user back to the requesting client. This choice is again limiting the scope of responsibility for the Authorization API as it does not need to manage and share secrets. Zauthly's Authorization API contains the two important parts of the ZT architecture, the PE and PA. The PE is defined by the /userinfo endpoint which contains the logic around what a user permissions are, if a user is a valid user, and if a user is authorized. The PA is defined by the /authorize endpoint which handles creating the tokens and enabling the communication between the client and server. Because Flask is a lightweight framework, it does not perform well on heavy application loads. The performance of Flask is hindered because it is a Python application. It is not as performant as a web application written in a lower level language like C. This could be avoided by leveraging the potential ability to scale horizontally if traffic is high.

The Zauthly Database is used to store the information required to complete Oauth2 authorizations. Zauthly selected MongoDB as the database because the primary storage state of this database is JSON and the simplicity of NoSQL. It provides no complex database configuration setup or definitions of relations. The structure is based on collections and the Database contains four unique collections; Users, Clients, Tokens, and Authorization Codes. The User collection's primary key is the id, and the collection maintains the general information about a user: name, email, and permissions. The Clients collection's primary key is the id and contains the information required to verify a client based on Oauth2 standards. This includes the issue time, client secret, and client unique id. The Tokens collection's primary key is the token

string. The token contains information about what user it belongs to, when it was issued, and when it expires. The Authorization Codes collection's primary key is the code and contains information about when it was created and when it expires. Pydantic models, an example of the user model is shown below and the complete models can be found in the appendix, are used to define the data that is stored in the collections. In order to enforce this scheme before it is stored in the collection, Zauthly uses an abstraction in the form of a Database API to restrict interactions to the database. A downside of MongoDB is that it does not have the ability to handle complex queries as the best performance comes from searching by the collections primary key.

```
class UserObject(ObjectId):
    @classmethod
    def __get_validators__(cls):
        yield cls.validate

    @classmethod
    def validate(cls, v):
        if not ObjectId.is_valid(v):
            raise ValueError("Invalid Object")
        return ObjectId(v)

    @classmethod
    def __modify_schema__(cls, field_schema):
        field_schema.update(type="string")

class UserModel(BaseModel):
    id: UserObject = Field(default_factory=UserObject, alias="_id")
    email: str = Field(...)
    name: str = Field(...)
    hd: str = Field(...)
    permissions: List[str] = Field(...)

    class Config:
        allow_population_by_field_name = True
        arbitrary_types_allowed = True
        json_encoders = {ObjectId: str}
        schema_extra = {
            "example": {
                "email": "mperry37@alaska.edu",
                "name": "joe bob",
                "hd": "mperry.io",
                "permissions": ["web", "admin", "chat"],
```

```
        }
    }

class UpdateUserModel(BaseModel):
    email: Optional[str]
    name: Optional[str]
    hd: Optional[str]
    permissions: Optional[List[str]]
    class Config:
        allow_population_by_field_name = True
        arbitrary_types_allowed = True
        json_encoders = {ObjectId: str}
        schema_extra = {
            "example": {
                "email": "mperry37@alaska.edu",
                "name": "joe bob",
                "hd": "mperry.io",
                "permissions": ["web", "admin", "chat"],
            }
        }
```

Zauthly's database API is written with FastAPI, a Python framework that enables a simple and fast way to create JSON based API's. The API is designed to perform some basic CRUD operations on the four collections. The main reason that FastAPI was selected was the asynchronous functions that enable reading and writing from a database. The asynchronous function calls allow the server to be non blocking in the main thread. Additionally FastAPI provides an automatic documentation generation based on decorators in the code. The design of the user endpoints are: get a user, get all users, update a user, create a user, and delete a user. The client endpoints are: get a client, delete a client, and create a client. The user endpoints are shown in the figure below and the complete definition of end points are included in the appendix.

```
@app.post("/user/create", response_description="Add new user",
response_model=UserModel)
async def create_user(user: UserModel = Body(...)):
    user = jsonable_encoder(user)
    new_user = await mongo_users.insert_one(user)
    created_user = await db['users'].find_one({"_id": new_user.inserted_id})
    return JSONResponse(status_code=status.HTTP_200_OK, content=created_user)
```

```
@app.get('/user/list', response_description="List all users", response_model=List[UserModel])
async def list_users():
    users = await mongo_users.find().to_list(1000)
    return JSONResponse(status_code=status.HTTP_200_OK, content=users)

@app.get("/user/id/{id}", response_description="Get a user", response_model=UserModel)
async def get_user_by_id(id: str):
    user = await mongo_users.find_one({"_id": id})
    if not user:
        return JSONResponse(status_code=status.HTTP_404_NOT_FOUND, content="No user
found")
    return JSONResponse(status_code=status.HTTP_200_OK, content=user)

@app.get("/user/{sub}", response_description="Get a user", response_model=UserModel)
async def get_user(sub: str):
    user = await mongo_users.find_one({"email": sub})
    if not user:
        return JSONResponse(status_code=status.HTTP_404_NOT_FOUND, content="No user
found")
    return JSONResponse(status_code=status.HTTP_200_OK, content=user)

@app.post('/user/delete', response_description="delete a user", response_model=UserModel)
async def delete_user(user: UserModel = Body(...)):
    res = await mongo_users.delete_one({"sub": user["sub"]})
    if res.deleted_count >= 1:
        return JSONResponse(status_code=status.HTTP_200_OK, content={"success": "true"})
    else:
        return JSONResponse(status_code=status.HTTP_304_NOT_MODIFIED)

@app.put("/user/update/{id}", response_description="update a users info")
async def update_user(id: str, user : UpdateUserModel = Body(...)):
    user = {key: value for key, value in user.dict().items() if value is not None}
    if len(user) >= 1:
        update_result = await mongo_users.update_one({"_id": id}, {"$set": user})
        if update_result.modified_count == 1:
            return JSONResponse(status_code=status.HTTP_200_OK)
    return JSONResponse(status_code=status.HTTP_404_NOT_FOUND)
```

Zauthly did not find a reason that a client should be mutated as the primary function is to validate client metadata and secrets. The token endpoints are: get a token, create a token, and delete a token. It was determined that tokens should not be updated and a new token should be issued if the old is invalidated. The authorization endpoints are: get an authorization code, create an authorization code, and delete an authorization code. Zauthly found that there was no reason to have mutable authorization codes. By utilizing the API the objects returned can be strictly

typed and validated before being sent to the caller. The objects returned from FastAPI endpoints are in JSON form which is a standard notation that enables simple integration to other systems besides Python based applications.

The final component of Zauthly is the Zaulthy_client, which is a collection of Python scripts that enable the creation of clients and administration of the authorization of users. Currently the system supports granting users any permissions they want but they must be evaluated by the PA within the Authorization API. The client is deployed as a docker container within the Zauthly tool. The process of creating a client requires sharing a one time secret securely, and the secret is printed within a docker container and the responsibility of the secret is passed to the administrator that can access the docker containers.

Deployment

Zauthly is deployed using containerization. The microservices have been converted to Docker files to provide flexibility and validation of configuration. By leveraging Docker containers the system can be deployed on any x86 based machine that can install Docker.

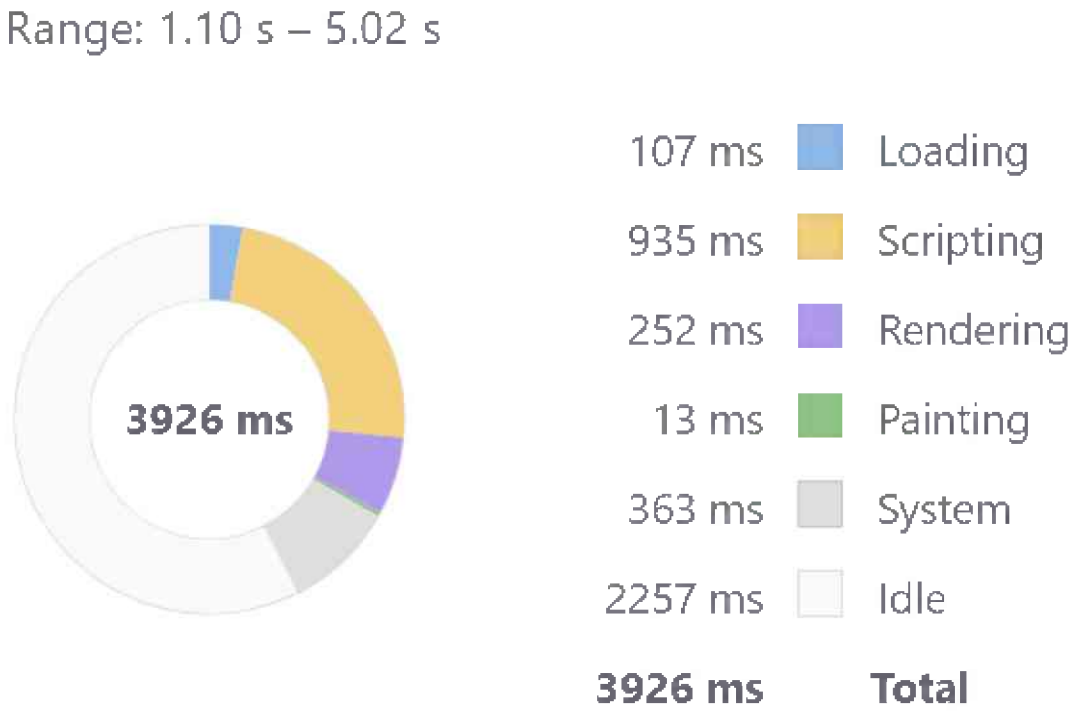
```
version: '3.1'

services:
  mongo:
    container_name: authz-mongo
    image: mongo
    restart: always
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: example
      MONGO_INITDB_DATABASE: authz
    expose:
      - "27017"
    volumes:
      - ./data:/data/db
  authz:
    container_name: authz-auth
    build:
      context: ./
      dockerfile: ./authz/Dockerfile
    ports:
      - "80:80"
    restart: unless-stopped
  dbpi:
    container_name: authz-dbapi
    build:
      context: ./
      dockerfile: ./db_api/Dockerfile
    expose:
      - "80"
    restart: unless-stopped
  client:
    container_name: authz-client
    build:
      context: ./
      dockerfile: ./client_helper/Dockerfile
    volumes:
      - ./client_helper/src/:/src/
    restart: unless-stopped
```

The dependencies are locked to ensure the system remains functional. The project includes a docker compose file for container orchestration. The Docker compose file exposes the internal Docker network between the Authorization API and Database API. The only externally exposed endpoints are the Authorization API's. This decision was important to increase the security of Zauthly, by limiting what parts of the application are exposed to the word. With the restriction of communication between the Authorization API and Database API to only the Docker network Zauthly can bypass the need for TLS and API verification. This is because the network security inside the Docker container is

only as secure as the system that is running Docker. If the communication between containers is compromised then the system running Docker must be compromised, as root is required to attach to the network. Because the container to container communication does not have any additional security requirements, simple HTTP can be used to increase the performance. An example of the deployment configuration file is shown above.

Performance

When introducing additional steps and a database call the performance of the Zauthly proof of concept was an important metric to show that user's experience and server throughput is not greatly affected. The performance tests look to provide a lower and upper bound of the server response time. The tests were performed with a server that is hosted in the cloud in Oregon and the requests originating from Alaska. The upper bound test was timing the complete time to log in and be granted access. This test was performed using Google Chrome's ability to measure performance.



This test was timed from the time login was clicked until the secret page was shown to the user. The total time including the Google Consent was approximately 4 seconds.
This shows the upper bound of 4 seconds to be granted access.

The lower bound of the system was calculated using the internal container to container communication. This was done over HTTP making 100 requests which were averaged to find the approximate time for the Database API to respond. Total time for 100 requests was 300 milliseconds, which gives a derived cost per call: 3 milliseconds. This performance metric represents the best case performance of the container to container communication.

Overall the performance metrics that were measured were encouraging and indicate that the additional logic of Zauthly does not impact server throughput or user experience.

Future Work For Zauthly

The main future work for the Zauthly project are the important parts to taking this project to a true open source tool that more people can utilize, including additional configuration options. Allowing configuration of token life, and encryption methods could provide greater control of the security aspect of tokens. Additionally a full admin user interface instead of a Docker container that relies on scripts. Another important feature that should be added into Zauthly is managing multiple IdPs in one application. Providing more than Google as a login option would provide greater utility for users of this tool. Lastly, a validation that the system can scale horizontally would be important as the system in theory should but would need to be tested. This could be done by providing a load balancer to help distribute traffic evenly to multiple Authorization API's.

Conclusion

Zauthly shows that a ZT Oauth2 authentication service is possible and performance is acceptable. The clear isolation of authentication and authorization follow the Zero Trust paradigms. Zauthly shows that an Oauth2 flow is still possible if a PE and PA are introduced to the system. This additional logic has a minimal impact on performance of a similar Oauth2 client to server interaction. The clear separation of responsibilities also provides a simple path to microservice deployment. The Authorization API communicates directly to the SSO client Vouch and a token flow is executed. Zauthly has shown the ability to provide developers a way to integrate more secure authorization and authentication into the applications they develop. It has the potential to be applied at large scales given its design and could become a widely used tool for custom authorization management.

References

1. Berners-Lee, T., Cailliau, R., Luotonen, A., Nielsen, H. and Secret, A. (1994). The World-Wide Web. *Communications of the ACM*, [online] 37(8), pp.76-82. Available at: <https://dl.acm.org/doi/pdf/10.1145/179606.179671>.
2. Campbell, B., Identity, P., Mortimore, C., Jones, M., Salesforce and Microsoft. (2015). *RFC 7522 - Security Assertion Markup Language (SAML) 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants*. [online] Available at: <https://datatracker.ietf.org/doc/html/rfc7522>.
3. Empower. (2021, February 8). TechEmpower framework benchmarks. *TechEmpower Framework Benchmarks*. Retrieved March 23, 2022, from https://www.techempower.com/benchmarks/#section=data-r20&hw=ph&test=query&l=zijzen-7&a=2
4. Flask, F. (2022). Design decisions in flask. *Flask Documentation (2.0.x)*. Retrieved March 23, 2022, from https://flask.palletsprojects.com/en/2.0.x/design/
5. Grinberg, M. (2018). *Flask web development: developing web applications with python*. " O'Reilly Media, Inc."
6. IETF, Datatracker.ietf.org. 2006. *RFC 4511 - Lightweight Directory Access Protocol (LDAP): The Protocol.* [online] Available at: <https://datatracker.ietf.org/doc/html/rfc4511>.
7. IETF, Datatracker.ietf.org. 2010. *RFC 5849 - The OAuth 1.0 Protocol*. [online] Available at: <https://datatracker.ietf.org/doc/html/rfc5849>.
8. IETF, Datatracker.ietf.org. 2012. *RFC 6749 - The OAuth 2.0 Authorization Framework*. [online] Available at: <https://datatracker.ietf.org/doc/html/rfc6749>.
9. JumpCloud, J. C. (2021). Roadmap to the DOMAINLESS enterprise. *JumpCloud*. Retrieved March 23, 2022, from https://jumpcloud.com/resources/domainless-enterprise-roadmap
10. Madden, J., & TechTarget. (2016). *Azure Active Directory, identity and access management* ... Retrieved March 24, 2022, from https://info.microsoft.com/rs/157-GQE-382/images/EN-CNTNT-Whitepaper-JMActiveDirectoryandIdentityWhitepaper.pdf
11. Neuman, B. C. and T. Ts'o (1994), "Kerberos: an authentication service for computer networks," in IEEE Communications Magazine, vol. 32, no. 9, pp. 33-38, Sept. 1994, doi: 10.1109/35.312841.
12. Nginx, N. (2022). If is evil... when used in location context. *Web Server Load Balancing with NGINX Plus*. Retrieved March 23, 2022, from https://www.nginx.com/resources/wiki/start/topics/depth/ifisevil/
13. Okta, O. (2021). Okta Security Technical Whitepaper. *Okta*. Retrieved March 23, 2022, from https://www.okta.com/resources/whitepaper/okta-security-technical-white-paper/

14. Oppliger, R. (1996) *Authentication systems for secure networks*. Boston, Artech House, 1996.
15. Colvin, S., et al. (2022). Overview. *pydantic Developers Manual*. Retrieved March 23, 2022, from https://pydantic-docs.helpmanual.io/
16. Ramírez, S. (2018). About FastAPI. *FASTAPI*. Retrieved March 23, 2022, from https://fastapi.tiangolo.com/
17. Rescorla, E. and RTFM, Inc., (2000). *RFC 2818 - HTTP Over TLS*. [online] Datatracker.ietf.org. Available at: <https://datatracker.ietf.org/doc/html/rfc2818>.
18. Rose, S., Borchert, O., Mitchell, S. and Connelly, S., (2020). Zero Trust Architecture. *NIST Special Publication* 800-207, [online] Available at: <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-207.pdf>.
19. Thönes, J. (2015), "Microservices," in *IEEE Software*, vol. 32, no. 1, pp. 116-116, Jan.-Feb. 2015, doi: 10.1109/MS.2015.11.

Appendix

Github

https://github.com/mattp0/Zero-Trust-Authz

Authorization API

Oauth.py

```
from authlib.integrations.flask_oauth2 import (
    AuthorizationServer,
    ResourceProtector,
)
from authlib.oauth2.rfc6749.grants import (
    AuthorizationCodeGrant as _AuthorizationCodeGrant,
)
from authlib.oidc.core import UserInfo
from authlib.oidc.core.grants import OpenIDCode as _OpenIDCode
from mongomixin import Oauth2AuthorizationCodeMixin
from helper import (
    create_bearer_token_validator,
    query_client,
    save_token,
    create_authz_code,
    get_authz_code,
    delete_authz_code,
    get_user_by_id,
    create_revocation_endpoint

)
from model import User
import json
import secrets
import time
from config import DUMMY_JWT_CONFIG, allowed_permission

def create_authorization_code(client, grant_user, request):
    data = {
        "client_id": client.client_id,
        "redirect_uri": request.redirect_uri,
        "response_type":request.response_type,
        "scope": request.scope,
        "grant_user": grant_user.id,
```

```
        "nonce": secrets.token_urlsafe(16),
        "auth_time": int(time.time())
    }
    info = json.loads(create_authz_code(data))
    item = Oauth2AuthorizationCodeMixin(info)
    return item.get_code()

class AuthorizationCodeGrant(_AuthorizationCodeGrant):
    def create_authorization_code(self, client, grant_user, request):
        return create_authorization_code(client, grant_user, request)

    def parse_authorization_code(self, code, client):
        info = json.loads(get_authz_code(code))
        item = Oauth2AuthorizationCodeMixin(info)
        if item and not item.is_expired():
            return item

    def delete_authorization_code(self, authorization_code):
        res = delete_authz_code(authorization_code.get_code())

    def authenticate_user(self, authorization_code):
        info = json.loads(get_authz_code(authorization_code.get_code()))
        user_data = get_user_by_id(info['grant_user'])
        user = User(json.loads(user_data))
        return user

def exists_nonce(nonce, req):
    return True

def generate_user_info(user, scope):
    token_user = get_user_by_id(user)
    token_user = User(json.loads(token_user))
    permissions = token_user.get_permissions()
    if allowed_permission not in permissions:
        return UserInfo(name=token_user.get_name(), email="bad@bad.com", team="bad")
    id_email = str(token_user.get_user_id()) + "@mperry.io"
    return UserInfo(name=token_user.get_name(), email=id_email,
team=token_user.get_permissions())

class OpenIDCode(_OpenIDCode):
    def exists_nonce(self, nonce, request):
        return exists_nonce(nonce, request)

    def get_jwt_config(self, grant):
        return DUMMY_JWT_CONFIG
```

```
    def generate_user_info(self, user, scope):
        return generate_user_info(user.get_user_id(), scope)


authorization = AuthorizationServer(
    query_client=query_client,
    save_token=save_token,
)
require_oauth = ResourceProtector()


def config_oauth(app):
    authorization.init_app(app)

    authorization.register_grant(AuthorizationCodeGrant, [
        OpenIDCode(require_nonce=False),
    ])

    revocation_cls = create_revocation_endpoint()
    authorization.register_endpoint(revocation_cls)
    # protect resource
    bearer_cls = create_bearer_token_validator()
    require_oauth.register_token_validator(bearer_cls())
```

Mongomixin.py

```
from authlib.common.encoding import json_loads, json_dumps
from authlib.oauth2.rfc6749 import ClientMixin, TokenMixin, AuthorizationCodeMixin
from authlib.oauth2.rfc6749.util import scope_to_list, list_to_scope
import time

class Oauth2ClientMixin(ClientMixin):
    "client mixin definition"
    def __init__(self, info : dict):
        self.client_id: str = info['_id']
        self.client_secret: str = info['client_secret']
        self.client_id_issued_at: int = info['client_id_issued_at']
        self.client_secret_expires_at: int = info['client_secret_expires_at']
        self._client_metadata = info['client_metadata']

    #based on requirements for the client mixin from authlib
    @property
    def client_info(self):
        return dict(
```

```
        client_id=self.client_id,
        client_secret=self.client_secret,
        client_id_issued_at=self.client_id_issued_at,
        client_secret_expires_at=self.client_secret_expires_at,
        )
@property
def client_metadata(self):
    if 'client_metadata' in self.__dict__:
        return self.__dict__['client_metadata']
    if self._client_metadata:
        data = self._client_metadata.replace("'", "\"")
        data = json_loads(data)
        self.__dict__['client_metadata'] = data
        return data
    return {}

def set_client_metadata(self, value):
    self._client_metadata = json_dumps(value)

@property
def redirect_uris(self):
    return self.client_metadata.get('redirect_uris', [])

@property
def token_endpoint_auth_method(self):
    return self.client_metadata.get(
        'token_endpoint_auth_method',
        'client_secret_basic'
    )

@property
def grant_types(self):
    return self.client_metadata.get('grant_types', [])

@property
def response_types(self):
    return self.client_metadata.get('response_types', [])

@property
def client_name(self):
    return self.client_metadata.get('client_name')

@property
def client_uri(self):
    return self.client_metadata.get('client_uri')
```

```
@property
def logo_uri(self):
    return self.client_metadata.get('logo_uri')

@property
def scope(self):
    return self.client_metadata.get('scope')

@property
def contacts(self):
    return self.client_metadata.get('contacts', [])

@property
def tos_uri(self):
    return self.client_metadata.get('tos_uri')

@property
def policy_uri(self):
    return self.client_metadata.get('policy_uri')

@property
def jwks_uri(self):
    return self.client_metadata.get('jwks_uri')

@property
def jwks(self):
    return self.client_metadata.get('jwks', [])

@property
def software_id(self):
    return self.client_metadata.get('software_id')

@property
def software_version(self):
    return self.client_metadata.get('software_version')

def get_client_id(self):
    return self.client_id

def get_default_redirect_uri(self):
    if self.redirect_uris:
        return self.redirect_uris[0]

def get_allowed_scope(self, scope):
    if not scope:
        return ''
```

```
        allowed = set(self.scope.split())
        scopes = scope_to_list(scope)
        return list_to_scope([s for s in scopes if s in allowed])

    def check_redirect_uri(self, redirect_uri):
        return redirect_uri in self.redirect_uris

    def has_client_secret(self):
        return bool(self.client_secret)

    def check_client_secret(self, client_secret):
        return self.client_secret == client_secret

    def check_token_endpoint_auth_method(self, method):
        return self.token_endpoint_auth_method == method

    def check_response_type(self, response_type):
        return response_type in self.response_types

    def check_grant_type(self, grant_type):
        return grant_type in self.grant_types


class Oauth2AuthorizationCodeMixin(AuthorizationCodeMixin):
    def __init__(self, info: dict):
        self.code:str=info["_id"]
        self.client_id:str=info["client_id"]
        self.redirect_uri:str=info["redirect_uri"]
        self.response_type:str=info["response_type"]
        self.scope:str=info["scope"]
        self.nonce:str=info["nonce"]
        self.auth_time:int=info["auth_time"]

    def is_expired(self):
        return self.auth_time + 300 < time.time()

    def get_redirect_uri(self):
        return self.redirect_uri

    def get_scope(self):
        return self.scope

    def get_auth_time(self):
        return self.auth_time

    def get_nonce(self):
```

```
        return self.nonce

    def get_code(self):
        return self.code

class Oauth2TokenMixin(TokenMixin):
    def __init__(self, info: dict):
        self.id:str=info["_id"]
        self.client_id:str=info["client_id"]
        self.user_id:str=info["user_id"]
        self.token_type:str=info["token_type"]
        self.access_token:str=info["access_token"]
        self.scope:str=info["scope"]
        self.issued_at:int=info["issued_at"]
        self.access_token_revoked_at:int=info["access_token_revoked_at"]
        self.expires_in:int=info["expires_in"]

    def get_id(self):
        return self.id

    def check_client(self, client):
        return self.client_id == client.get_client_id()

    def get_scope(self):
        return self.scope

    def get_expires_in(self):
        return self.expires_in

    def get_expires_at(self):
            return self.issued_at + self.expires_in

    def is_revoked(self):
        return self.access_token_revoked_at

    def is_expired(self):
        if not self.expires_in:
            return False

        expires_at = self.issued_at + self.expires_in
        return expires_at < time.time()
```

App.py

```
import secrets
from flask import Flask, redirect, session, url_for, request, jsonify, render_template
from dotenv import load_dotenv
import os
from flask_dance.contrib.google import make_google_blueprint, google
from authlib.oauth2 import OAuth2Error
from oauth import authorization, require_oauth, generate_user_info, config_oauth
from model import User
from helper import user_exists, create_json_user
import json
from authlib.integrations.flask_oauth2 import current_token
import time
from config import custom_redirect_url

load_dotenv()
app = Flask(__name__)
client_id = os.getenv('GOOGLE_CLIENT_ID')
client_secret = os.getenv('GOOGLE_CLIENT_SECRET')
app.secret_key = secrets.token_urlsafe(32)

app.config.from_pyfile("settings.py")
config_oauth(app)
os.environ['OAUTHLIB_INSECURE_TRANSPORT']='1'
os.environ['OAUTHLIB_RELAX_TOKEN_SCOPE']='1'
os.environ['AUTHLIB_INSECURE_TRANSPORT']='1'

blueprint = make_google_blueprint(
    client_id=client_id,
    client_secret=client_secret,
    reprompt_consent=True,
    scope=["profile", "email", "openid"],
    redirect_url=custom_redirect_url
    )

app.register_blueprint(blueprint, url_prefix="/login")

@app.route('/login')
def login():
    return redirect(url_for('google.login'))

@app.route('/logout')
def logout():
    if blueprint.token is not None:
        token = blueprint.token["access_token"]
        resp = google.post(
            "https://accounts.google.com/o/oauth2/revoke",
```

```
            params={"token": token},
            headers={"Content-Type": "application/x-www-form-urlencoded"}
        )
        del blueprint.token
    session.pop('User', None)

    return render_template('loggedout.html')

@app.route('/userinfo')
@require_oauth('profile')
def permissions():
    return jsonify(generate_user_info(current_token.user_id, current_token.scope))

@app.route('/authorize', methods=['GET', 'POST'])
def authorize():
    user_info_endpoint = '/oauth2/v2/userinfo'
    if not google.authorized:
        session['query_str'] = request.query_string
        return redirect(url_for("google.login", next=request.url))
    elif request.query_string == b":
        request.query_string = session['query_str']
    token_expire_time = blueprint.token['expires_at']
    if int(time.time()) >= token_expire_time:
        return redirect(url_for("logout"))
    session['User'] = google.get(user_info_endpoint).json()
    authz_user = user_exists(session['User'])
    if authz_user is not None:
        user = User(json.loads(authz_user))
    else:
        user = User(json.loads(create_json_user(session['User'])))
    if request.method == 'GET':
        try:
            _ = authorization.validate_consent_request(end_user=user)
        except OAuth2Error as error:
            return jsonify(dict(error.get_body()))
    return authorization.create_authorization_response(grant_user=user)

@app.route('/revoke', methods=['POST'])
def revoke_token():
    return authorization.create_endpoint_response('revocation')

@app.route('/token/', methods=['POST'])
def token():
    return authorization.create_token_response()

if __name__ == "__main__":
```

```
app.run(host='0.0.0.0', port=8060)
```

Helper.py

```
import requests
import json
from config import db_api_url, base_permissions, domain
import time
from mongomixin import Oauth2ClientMixin, Oauth2TokenMixin

def build_user_json(user):
    json_user = {
        "email": f"{user['email']}",
        "name": f"{user['name']}",
        "hd": domain,
        "permissions": base_permissions
    }
    return json_user

def create_json_user(user) -> dict:
    create_user_endpoint = db_api_url+"/user/create"
    json_user = build_user_json(user)
    response = requests.post(create_user_endpoint, data=json.dumps(json_user))
    if response.status_code == 200:
        return response.content
    return None

def user_exists(user) -> dict:
    user_endpoint = db_api_url + "/user/" + str(user['email'])
    response = requests.get(user_endpoint)
    if response.status_code == 200:
        return response.content
    elif response.status_code == 404:
        return None
    else:
        raise Exception("Unknown Error as occurred")

def get_user_by_id(id) -> dict:
    user_endpoint = db_api_url + "/user/id/" + id
    response = requests.get(user_endpoint)
    if response.status_code == 200:
        return response.content
```

```
    elif response.status_code == 404:
        return None
    else:
        raise Exception("Unknown Error as occurred")

def create_authz_code(data):
    update_endpoint = db_api_url + "/authcode/create"
    response = requests.post(update_endpoint, data=json.dumps(data))
    if response.status_code == 200:
        return response.content
    return None

def get_authz_code(code):
    update_endpoint = db_api_url + "/authcode/" + code
    response = requests.get(update_endpoint)
    if response.status_code == 200:
        return response.content
    return None

def delete_authz_code(code):
    update_endpoint = db_api_url + "/authcode/delete/" + code
    response = requests.get(update_endpoint)
    if response.status_code == 200:
        return True
    return False

def query_client(client_id):
    client_endpoint = db_api_url + "/client/" + str(client_id)
    response = requests.get(client_endpoint)
    client = Oauth2ClientMixin(json.loads(response.content))
    return client

def save_token(token, request):
    client_endpoint = db_api_url + "/token/create"
    if request.user:
        user_id = request.user.get_user_id()
    else:
        user_id = None
    client = request.client
    item ={
        "client_id": client.client_id,
        "user_id": user_id,
        "issued_at": int(time.time()),
        "expires_in": 300,
        "access_token_revoked_at": 0,
        **token
```

```
    }
    response = requests.post(client_endpoint, data=json.dumps(item))


def create_query_token_func():
    """Create an ``query_token`` function for revocation, introspection
    token endpoints.
    """
    def query_token(token, _):
        client_endpoint = db_api_url + "/token/" + token
        response = requests.get(client_endpoint)
        return Oauth2TokenMixin(json.loads(response.content))
    return query_token


def create_revocation_endpoint():
    """Create a revocation endpoint class
    """
    from authlib.oauth2.rfc7009 import RevocationEndpoint
    query_token = create_query_token_func()

    class _RevocationEndpoint(RevocationEndpoint):
        def query_token(self, token, token_type_hint):
            return query_token(token, token_type_hint)

        def revoke_token(self, token, request):
            client_endpoint = db_api_url + "/token/update/" + token.get_id()
            now = int(time.time())
            token.access_token_revoked_at = now
            response = requests.post(client_endpoint, data=json.dumps(token))



    return _RevocationEndpoint


def create_bearer_token_validator():
    """Create an bearer token validator class
    """
    from authlib.oauth2.rfc6750 import BearerTokenValidator

    class _BearerTokenValidator(BearerTokenValidator):
        def authenticate_token(self, token_string):
            client_endpoint = db_api_url + "/token/" + token_string
            response = requests.get(client_endpoint)
            return Oauth2TokenMixin(json.loads(response.content))
```

```
    def request_invalid(self, request):
        return False

    def token_revoked(self, token):
        revoke_time = token.is_revoked()
        if revoke_time != 0:
            if revoke_time > int(time.time()):
                return True
        return False

return _BearerTokenValidator
```

Config.py

```
db_api_url = "http://authz-dbapi"
base_permissions = ["test"]
domain = "mperry.io"
allowed_permission = "web"
custom_redirect_url = "http://auth.mperry.io/authorize"

DUMMY_JWT_CONFIG = {
    'key': 'secret-key',
    'alg': 'HS256',
    'iss': 'http://mperry.io',
    'exp': 3600,
}
```

Dockerfile

```
FROM python:3.9

WORKDIR /src

COPY ./authz/requirements.txt .

RUN pip install -r requirements.txt

COPY ./authz/src/ .
```

```
CMD ["gunicorn", "app:app", "-b", "0.0.0.0:80"]
```

Database API

App.py

```
from fastapi import FastAPI, Body, status
from fastapi.responses import JSONResponse
from fastapi.encoders import jsonable_encoder
from fastapi import FastAPI
from typing import List
import motor.motor_asyncio

from model import UpdateTokenModel, UserModel, UpdateUserModel, ClientModel,
AuthCodeModel, TokenModel

app = FastAPI()
client =
motor.motor_asyncio.AsyncIOMotorClient("mongodb://root:example@authz-mongo:27017/?a
uthSource=admin")
db = client.authn
mongo_users = db['users']
mongo_clients = db['clients']
mongo_tokens = db['tokens']
mongo_authcodes = db['authcodes']

@app.post("/user/create", response_description="Add new user",
response_model=UserModel)
async def create_user(user: UserModel = Body(...)):
    user = jsonable_encoder(user)
    new_user = await mongo_users.insert_one(user)
    created_user = await db['users'].find_one({"_id": new_user.inserted_id})
    return JSONResponse(status_code=status.HTTP_200_OK, content=created_user)

@app.get('/user/list', response_description="List all users", response_model=List[UserModel])
async def list_users():
    users = await mongo_users.find().to_list(1000)
    return JSONResponse(status_code=status.HTTP_200_OK, content=users)

@app.get("/user/id/{id}", response_description="Get a user", response_model=UserModel)
async def get_user_by_id(id: str):
    user = await mongo_users.find_one({"_id": id})
```

```
    if not user:
        return JSONResponse(status_code=status.HTTP_404_NOT_FOUND, content="No user
found")
    return JSONResponse(status_code=status.HTTP_200_OK, content=user)

@app.get("/user/{sub}", response_description="Get a user", response_model=UserModel)
async def get_user(sub: str):
    user = await mongo_users.find_one({"email": sub})
    if not user:
        return JSONResponse(status_code=status.HTTP_404_NOT_FOUND, content="No user
found")
    return JSONResponse(status_code=status.HTTP_200_OK, content=user)

@app.post('/user/delete', response_description="delete a user", response_model=UserModel)
async def delete_user(user: UserModel = Body(...)):
    res = await mongo_users.delete_one({"sub": user["sub"]})
    if res.deleted_count >= 1:
        return JSONResponse(status_code=status.HTTP_200_OK, content={"success": "true"})
    else:
        return JSONResponse(status_code=status.HTTP_304_NOT_MODIFIED)

@app.put("/user/update/{id}", response_description="update a users info")
async def update_user(id: str, user : UpdateUserModel = Body(...)):
    user = {key: value for key, value in user.dict().items() if value is not None}
    if len(user) >= 1:
        update_result = await mongo_users.update_one({"_id": id}, {"$set": user})
        if update_result.modified_count == 1:
            return JSONResponse(status_code=status.HTTP_200_OK)
    return JSONResponse(status_code=status.HTTP_404_NOT_FOUND)

@app.post("/client/create", response_description="Add new client",
response_model=ClientModel)
async def create_client(client: ClientModel = Body(...)):
    client = jsonable_encoder(client)
    new_client = await mongo_clients.insert_one(client)
    created_client = await db['clients'].find_one({"_id": new_client.inserted_id})
    return JSONResponse(status_code=status.HTTP_200_OK, content=created_client)

@app.get("/client/{id}", response_description="Get a client", response_model=ClientModel)
async def get_client(id: str):
    client = await mongo_clients.find_one({"_id": id})
    if not client:
        return JSONResponse(status_code=status.HTTP_404_NOT_FOUND, content="No client
found")
    return JSONResponse(status_code=status.HTTP_200_OK, content=client)
```

```
@app.post('/client/delete', response_description="delete a client",
response_model=ClientModel)
async def delete_client(client: ClientModel = Body(...)):
    res = await mongo_clients.delete_one({"sub": client["sub"]})
    if res.deleted_count >= 1:
        return JSONResponse(status_code=status.HTTP_200_OK, content={"success": "true"})
    else:
        return JSONResponse(status_code=status.HTTP_304_NOT_MODIFIED)

@app.post("/token/create", response_description="Add new token",
response_model=TokenModel)
async def create_token(token: TokenModel = Body(...)):
    token = jsonable_encoder(token)
    new_token = await mongo_tokens.insert_one(token)
    created_token = await db['tokens'].find_one({"_id": new_token.inserted_id})
    return JSONResponse(status_code=status.HTTP_200_OK, content=created_token)

@app.get("/token/{token_str}", response_description="Get a token",
response_model=TokenModel)
async def get_token(token_str: str):
    token = await mongo_tokens.find_one({"access_token": token_str})
    if not token:
        return JSONResponse(status_code=status.HTTP_400_BAD_REQUEST, content="No
token found")
    return JSONResponse(status_code=status.HTTP_200_OK, content=token)

@app.put("/token/update/{id}", response_description="update a tokens info")
async def update_token(id: str, token : UpdateTokenModel = Body(...)):
    token = {key: value for key, value in token.dict().items() if value is not None}
    if len(token) >= 1:
        update_result = await mongo_tokens.update_one({"_id": id}, {"$set": token})
        if update_result.modified_count == 1:
            return JSONResponse(status_code=status.HTTP_200_OK)
    return JSONResponse(status_code=status.HTTP_404_NOT_FOUND)

@app.post('/token/delete', response_description="delete a token",
response_model=TokenModel)
async def delete_token(token: TokenModel = Body(...)):
    res = await mongo_tokens.delete_one({"_id": token["_id"]})
    if res.deleted_count >= 1:
        return JSONResponse(status_code=status.HTTP_200_OK, content={"success": "true"})
    else:
        return JSONResponse(status_code=status.HTTP_304_NOT_MODIFIED)

@app.post("/authcode/create", response_description="Add new authcode",
response_model=AuthCodeModel)
```

```
async def create_authcode(authcode: AuthCodeModel = Body(...)):
    authcode = jsonable_encoder(authcode)
    new_authcode = await mongo_authcodes.insert_one(authcode)
    created_authcode = await db['authcodes'].find_one({"_id": new_authcode.inserted_id})
    return JSONResponse(status_code=status.HTTP_200_OK, content=created_authcode)

@app.get("/authcode/{code}", response_description="Get a authcode",
response_model=AuthCodeModel)
async def get_authcode(code: str):
    authcode = await mongo_authcodes.find_one({"_id": code})
    if not authcode:
        return JSONResponse(status_code=status.HTTP_400_BAD_REQUEST, content="No
token found")
    return JSONResponse(status_code=status.HTTP_200_OK, content=authcode)

@app.get('/authcode/delete/{authcode}', response_description="delete a authcode")
async def delete_authcode(authcode: str):
    res = await mongo_authcodes.delete_one({"_id": authcode})
    if res.deleted_count >= 1:
        return JSONResponse(status_code=status.HTTP_200_OK, content={"success": "true"})
    else:
        return JSONResponse(status_code=status.HTTP_304_NOT_MODIFIED)
```

Model.py

```
from pydantic import BaseModel, Field
from bson import ObjectId
from typing import Optional, List

class UserObject(ObjectId):
    @classmethod
    def __get_validators__(cls):
        yield cls.validate

    @classmethod
    def validate(cls, v):
        if not ObjectId.is_valid(v):
            raise ValueError("Invalid Object")
        return ObjectId(v)

    @classmethod
    def __modify_schema__(cls, field_schema):
        field_schema.update(type="string")
```

```
class UserModel(BaseModel):
    id: UserObject = Field(default_factory=UserObject, alias="_id")
    email: str = Field(...)
    name: str = Field(...)
    hd: str = Field(...)
    permissions: List[str] = Field(...)

    class Config:
        allow_population_by_field_name = True
        arbitrary_types_allowed = True
        json_encoders = {ObjectId: str}
        schema_extra = {
            "example": {
                "email": "mperry37@alaska.edu",
                "name": "joe bob",
                "hd": "mperry.io",
                "permissions": ["web", "admin", "chat"],
            }
        }

class UpdateUserModel(BaseModel):
    email: Optional[str]
    name: Optional[str]
    hd: Optional[str]
    permissions: Optional[List[str]]
    class Config:
        allow_population_by_field_name = True
        arbitrary_types_allowed = True
        json_encoders = {ObjectId: str}
        schema_extra = {
            "example": {
                "email": "mperry37@alaska.edu",
                "name": "joe bob",
                "hd": "mperry.io",
                "permissions": ["web", "admin", "chat"],
            }
        }

class ClientObject(ObjectId):
    @classmethod
    def __get_validators__(cls):
        yield cls.validate

    @classmethod
    def validate(cls, v):
```

```
        if not ObjectId.is_valid(v):
            raise ValueError("Invalid Object")
        return ObjectId(v)

    @classmethod
    def __modify_schema__(cls, field_schema):
        field_schema.update(type="string")

class ClientModel(BaseModel):
    client_id: ClientObject = Field(default_factory=ClientObject, alias="_id")
    client_secret: str = Field(...)
    client_id_issued_at: int = Field(...)
    client_secret_expires_at: int = Field(...)
    client_metadata: str = Field(...)

    class Config:
        allow_population_by_field_name = True
        arbitrary_types_allowed = True
        json_encoders = {ObjectId: str}
        schema_extra = {
            "example": {
                "client_secret": "joe bob",
                "client_id_issued_at": 45123,
                "client_secret_expires_at": 84000,
                "client_metadata": "stuff",
            }
        }

class TokenObject(ObjectId):
    @classmethod
    def __get_validators__(cls):
        yield cls.validate

    @classmethod
    def validate(cls, v):
        if not ObjectId.is_valid(v):
            raise ValueError("Invalid Object")
        return ObjectId(v)

    @classmethod
    def __modify_schema__(cls, field_schema):
        field_schema.update(type="string")

class TokenModel(BaseModel):
    id: TokenObject = Field(default_factory=TokenObject, alias="_id")
    client_id: str = Field(...)
```

```
    user_id: str = Field(...)
    token_type: str = Field(...)
    scope: str = Field(...)
    access_token: str = Field(...)
    expires_in: int = Field(...)
    issued_at: int = Field(...)
    access_token_revoked_at: int = Field(...)

    class Config:
        allow_population_by_field_name = True
        arbitrary_types_allowed = True
        json_encoders = {ObjectId: str}
        schema_extra = {
            "example": {
                "client_id": "1231232145",
                "user_id": "1231232",
                "token_type": "Bearer",
                "access_token": 'swerasdfwewdasdf',
                "scope": "openid profile email",
                "issued_at": 12314,
                "access_token_revoked_at": 123124,
                "expires_in": 0,
            }
        }

class UpdateTokenModel(BaseModel):
    client_id: Optional[str]
    user_id: Optional[str]
    token_type: Optional[str]
    scope: Optional[str]
    access_token: Optional[str]
    expires_in: Optional[int]
    issued_at: Optional[int]
    access_token_revoked_at: Optional[int]

    class Config:
        allow_population_by_field_name = True
        arbitrary_types_allowed = True
        json_encoders = {ObjectId: str}
        schema_extra = {
            "example": {
                "client_id": "1231232145",
                "user_id": "1231232",
                "token_type": "Bearer",
                "access_token": 'swerasdfwewdasdf',
                "scope": "openid profile email",
```

```
            "issued_at": 12314,
            "access_token_revoked_at": 123124,
            "expires_in": 0,
        }
    }

class AuthCodeObject(ObjectId):
    @classmethod
    def __get_validators__(cls):
        yield cls.validate

    @classmethod
    def validate(cls, v):
        if not ObjectId.is_valid(v):
            raise ValueError("Invalid Object")
        return ObjectId(v)

    @classmethod
    def __modify_schema__(cls, field_schema):
        field_schema.update(type="string")

class AuthCodeModel(BaseModel):
    code: AuthCodeObject = Field(default_factory=AuthCodeObject, alias="_id")
    client_id: str = Field(...)
    redirect_uri: str = Field(...)
    response_type: str = Field(...)
    scope: str = Field(...)
    grant_user: str = Field(...)
    nonce : str = Field(...)
    auth_time: int = Field(...)

    class Config:
        allow_population_by_field_name = True
        arbitrary_types_allowed = True
        json_encoders = {ObjectId: str}
        schema_extra = {
            "example": {
                "client_id": "45123",
                "redirect_uri": "http://mperry.io",
                "response_type": "code",
                "scope": "openid profile email",
                "grant_user": "21323423245",
                "nonce": "1212134",
                "auth_time": 12314512312
            }
        }
```

Dockerfile

```
FROM python:3.9

WORKDIR /src

COPY ./db_api/requirements.txt .

RUN pip install -r requirements.txt

COPY ./db_api/src/ .

CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "80"]
```

Docker-Compose

Docker-compose.yml

```
version: '3.1'

services:
  mongo:
    container_name: authz-mongo
    image: mongo
    restart: always
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: example
      MONGO_INITDB_DATABASE: authz
    expose:
      - "27017"
    volumes:
      - ./data:/data/db
  authz:
    container_name: authz-auth
    build:
```

```
    context: ./
    dockerfile: ./authz/Dockerfile
  ports:
    - "80:80"
  restart: unless-stopped
 dbpi:
  container_name: authz-dbapi
  build:
    context: ./
    dockerfile: ./db_api/Dockerfile
  expose:
    - "80"
  restart: unless-stopped
 client:
  container_name: authz-client
  build:
    context: ./
    dockerfile: ./client_helper/Dockerfile
  volumes:
    - ./client_helper/src/:/src/
  restart: unless-stopped
```

Zauthly Client

Create_client.py

```
import secrets
import json
import requests
import time
db_api_url = "http://authz-dbapi"

def create_client():
    data = {
        "client_secret": secrets.token_urlsafe(32),
        "client_id_issued_at": int(time.time()),
        "client_secret_expires_at": int(time.time()) + int(84_000),
        "client_metadata": "{'redirect_uris':
['http://secret.mperry.io/auth'],'token_endpoint_auth_method':
'client_secret_basic','grant_types': ['authorization_code'],'response_types':
['code'],'client_name': 'The Main Frame','client_uri': 'http://secret.mperry.io','scope': 'openid
profile email'}"
```

```
    }
    update_endpoint = db_api_url + "/client/create"
    print("preparing to send post request")
    print(json.dumps(data))
    response = requests.post(update_endpoint, data=json.dumps(data))
    if response.status_code == 202 or response.status_code == 200:
        return True, response.content
    return False, None


if __name__ == "__main__":
    print("Creating a client")
    res, content = create_client()
    print(res, content)
```

Update_client.py

```
import json
import requests

db_api_url = "http://authz-dbapi"
new_perms = {
    "permissions": ["web"]
}


def update_user(user, new_user):
    print(f"testing update user, with info:{user}, to {new_user}")
    update_endpoint = db_api_url + "/user/update/" + user["_id"]
    response = requests.put(update_endpoint, data=json.dumps(new_user))
    if response.status_code == 200:
        return True
    return False

def user_exists(user):
    print("testing user exists")
    user_endpoint = db_api_url + "/user/" + str(user['email'])
    response = requests.get(user_endpoint)
    if response.status_code == 200:
        return True, response.content
    elif response.status_code == 404:
        return False, None
```

```
    else:
        raise Exception("Unknown Error as occurred")

if __name__ == "__main__":
    user = {"email": "mperry37@alaska.edu"}
    res, content = user_exists(user)
    user_info = json.loads(content)
    res = update_user(user_info, new_perms)
    print(res)
```

Dockerfile

```
FROM python:3.9

WORKDIR /src

COPY ./client_helper/requirements.txt .

RUN pip install -r requirements.txt

ENTRYPOINT ["tail", "-f", "/dev/null"]
```