Authentication Protocol54 documentation

Protocol54 Glossary

In the Protocol54 documentation we use the following concepts:

1. **Biome** is a set of services united by one Auth service.
2. **Auth service** is the main authentication service. The service is used to register all users and get an authentication token (APT54), also to manage permissions for all services.
3. **APT54 (Authentication Protocol 54)** is Authentication Protocol token for authorization/registration on any service.
4. **Actor** is a single biome object that is able to authorize and use resources on various services, with the exception of the actor group. Types of actors:
   - **Service** is a type of Ecosystem54 actor.
   - **User** is a type of actor that fully uses the authentication system, i.e. registered via QR code and using APT54 for authorization.
   - **Classic user** is a type of actor that uses login/password form for registration and authorization.
   - **Group** is a type of actor that groups a set of actors with uniform permissions.
5. **Client service** is the final service for the user to use. This service is one of the biome members. A service can only be attached to one biome.
6. **Auth SSO (Authorization Single Sign-On)** - a method of authorization/registration through the Auth service on any of the client services.
7. **Assistant services** are a unique type of actor service, which is designed to reduce the load on the main service, provide additional opportunities for the main service. Moreover, its presence in the system is optional.
8. **Permissions** - access rights to the resources of client service. They are needed to allow or deny actor to use the resources. Permissions can be set individually or for groups of users. Initially, there are three groups: "Default", "Ban" and "Admin". For example, a user who is in the "Ban" group has no access to any service resources. A user who is in the "Admin" group is an administrator on all services.

Protocol54 Philosophy

The unique architecture of the APT54 protocol provides:

1. Single authorization management system / Single permission system
   - Fast authentication system
   - Built-in two-factor authentication
   - Highest level of security
2. Easy maintenance and management of the protocol
   - Use of open-source technology only
   - Rapid deployment and connectivity to existing applications
   - Simplicity of the solution ensures faster onboarding of new staff
3. Modularity, scalability of the protocol
   - Possibility of unlimited improvement of the protocol
4. High stress resistance of the entire system
   - Ensured through the independence of individual components from each other
   - Use of fast encryption algorithms

Introduction to Protocol54

The AP54 protocol is designed specifically for the implementation of service architecture. For convenient support and development of a service system based on this protocol, a development team of no more than 10 people is required. Since the protocol is written in Python, it is impossible to expand the development team to a large size (as, for example, in projects in Java).

AP54 protocol allows implementing a distributed service architecture of the project, in which services are not connected to each other, but have a common authentication system.

One of the most important properties of this design approach is the modularity of such a system. This property distinguishes such an approach from architectures in which the project is implemented as one indivisible part. Each individual service based on AP54 protocol performs its task independently of the other, and if one service fails, the rest of the system continues to work as usual.

This architecture also provides great scalability potential since it is possible to add as many services as needed.

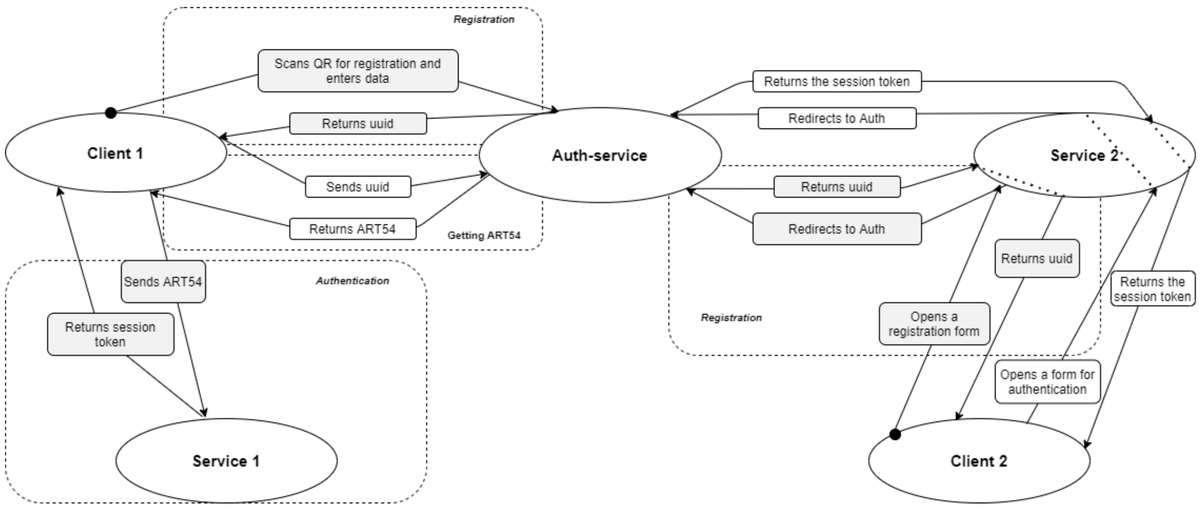
Another advantage of this architecture is that it requires a relatively small team of programmers (up to 10 people) to support a system based on the AP54 protocol regardless of its size (i.e. the number of services). This reduces money and time costs for software development and testing, unlike, for example, microservice architecture, which becomes extremely complex and costly as its size increases.

A common authentication system (i.e., only one account is needed to access all services within this system) is also an undoubted advantage of this architecture since it reduces the time that a user can spend on registration and logging into an account. The use of the APT54 token, in turn, increases the security of the system.

Problems that Protocol54 solves

1. **Scalability**. You can add as many separate services to the system as you like and maintain the speed of the entire system at a high level.
2. **Computing power**. Thanks to excellent scalability, the computing power of the entire system can grow almost indefinitely.
3. **Development**. The development of the system does not require a large group of developers and large labor costs because each service in the system can be developed separately from each other.
4. **Support**. The support of such a system, as well as the development, is not complicated and does not require large labor costs.
5. **Fault tolerance**. Such a system provides the highest level of fault-tolerance. If a problem occurs on one of the services, the rest of the system continues to operate normally.
6. **Security**. Thanks to the AP54 protocol, which implements two-factor authentication using the APT54 token, the high level of system security is ensured.

Protocol54 scheme



Protocol54 Phases

Protocol54 contains two large phases called biome and client.

Biome phase

The biome phase involves a full cycle of registration, authentication and user registration.

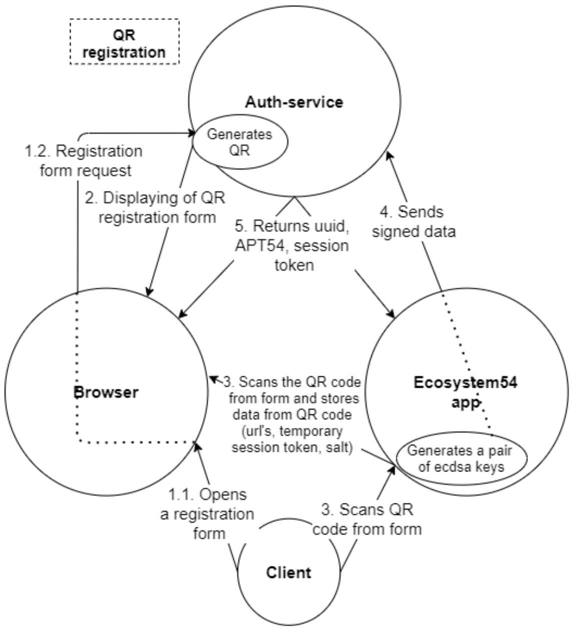
Registration

The user can register in two ways:

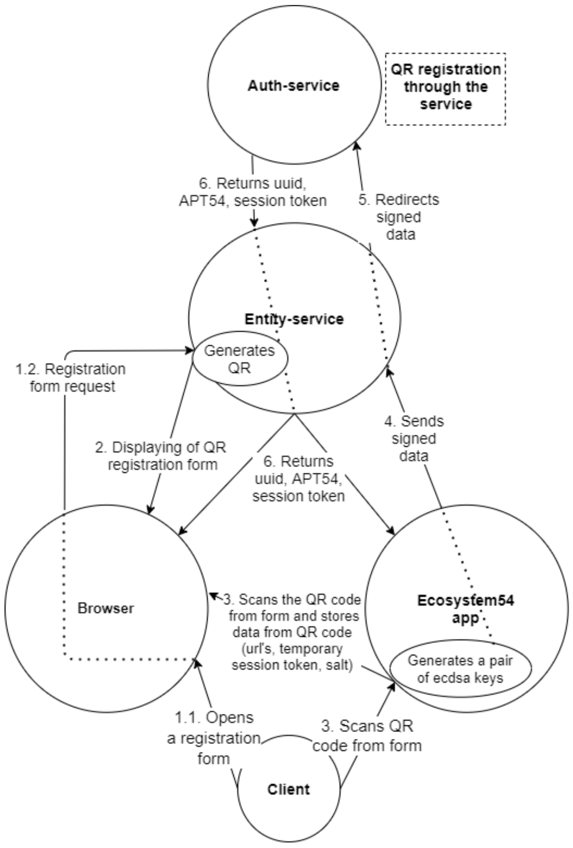
- Scanning QR code is the most reliable way.
  If the user uses QR code, he needs to install a special application Ecosystem54. This application is necessary to work with QR code, generating data, sending requests.

  When scanning QR code, the application generates a pair of ECDSA keys, which will be used as a proof of identity. After generation the application signs the salt with the generated private key and sends a request to the server with the signed salt and public key. The server returns the UUID (unique identifier) of the user in response.

  1. Register with QR using Ecosystem54 app:
     1. The client opens the registration form by QR code.
     2. The service generates QR code and displays QR code registration form in the browser.
     3. The client starts the process of scanning QR code, opens the registration in Ecosystem54 app. The client scans the generated QR code using Ecosystem54 app. The app stores metadata from QR code (URLs, QR token, salt).
     4. The app generates a pair of ECDSA keys and signs the salt with a private key. Then the app offers the user to enter his data and click a button to send the data. After clicking the app sends the data (pub_key, qr_token, signed_salt) to the service. (The service address is taken from qr_code_metadata).
     5. In response, Ecosystem54 receives and writes UUID generated by the Auth service, APT54 and session token.

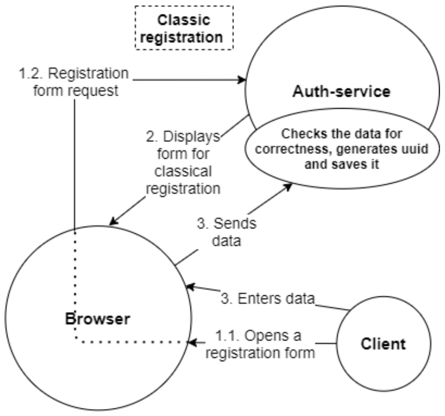

QR registration
Auth-service
Generates QR
1.2. Registration form request
2. Displaying of QR registration form
5. Returns uuid, APT54, session token
4. Sends signed data
Browser
3. Scans the QR code from form and stores data from QR code (url's, temporary session token, salt)
Ecosystem54 app
Generates a pair of ecdsa keys
1.1. Opens a registration form
3. Scans QR code from form
Client

  2. Registration with QR using Ecosystem54 app:
     1. The client opens the registration form by QR code.
     2. The service generates QR code and displays QR code registration form in the browser.
     3. The client starts the process of scanning QR code, opens the registration in Ecosystem54 app. The client scans the generated QR code using Ecosystem54 app. The app stores metadata from QR code (URLs, QR token, salt).
     4. The app generates a pair of ECDSA keys and signs the salt with a private key. Then the app offers the user to enter his data and click a button to send the data. After clicking the app sends the data (pub_key, qr_token, signed_salt) to the service (The service address is taken from qr_code_metadata).
     5. The service redirects the signed data to Auth.
     6. Auth generates and writes UUID and sends UUID, APT54 and session token to the Entity service, which redirects the data to Ecosystem54 app.

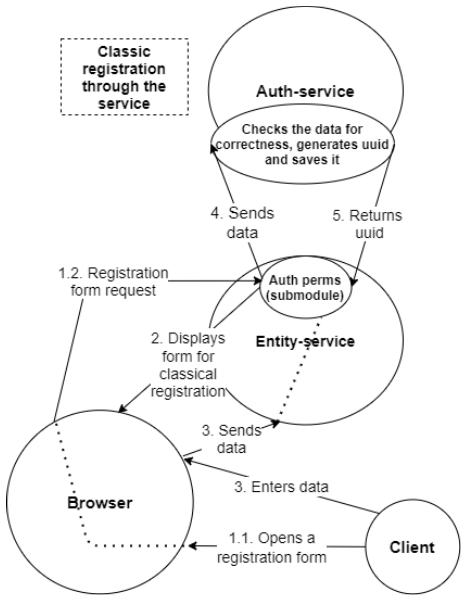

Auth-service
QR registration through the service
6. Returns uuid, APT54, session token
5. Redirects signed data
Entity-service
Generates QR
1.2. Registration form request
2. Displaying of QR registration form
6. Returns uuid, APT54, session token
4. Sends signed data
Browser
3. Scans the QR code from form and stores data from QR code (url's, temporary session token, salt)
Ecosystem54 app
Generates a pair of ecdsa keys
1.1. Opens a registration form
3. Scans QR code from form
Client

- Using the registration form.
  If the user uses the registration form, the server checks the entered data and also returns the user's UUID. Registration by form is possible only on the Auth service. When choosing a form for registration on any client service, the user will be automatically redirected to the Auth service by Auth SSO.

1. Classic registration with a form:
   1. The client opens a registration form.
   2. Browser displays the registration form.
   3. The client enters his data and sends it to the service. Auth checks the data for correctness, generates UUID and saves it.


Classic registration
1.2. Registration form request
Auth-service
Checks the data for correctness, generates uuid and saves it
2. Displays form for classical registration
3. Sends data
Browser
3. Enters data
1.1. Opens a registration form
Client

2. Classic registration with a form through the service:
   1. The client opens a registration form.
   2. Browser displays the registration form.
   3. The client enters his data and sends it to the service. The service checks the data for correctness.
   4. Auth perms sends the data to the Auth service. Auth checks the data for correctness, generates UUID and saves it.
   5. The Auth service sends UUID to the Entity service and it stores the user.


Classic registration through the service
Auth-service
Checks the data for correctness, generates uuid and saves it
4. Sends data
5. Returns uuid
1.2. Registration form request
Auth perms (submodule)
2. Displays form for classical registration
Entity-service
3. Sends data
3. Enters data
Browser
1.1. Opens a registration form
Client

Getting APT54

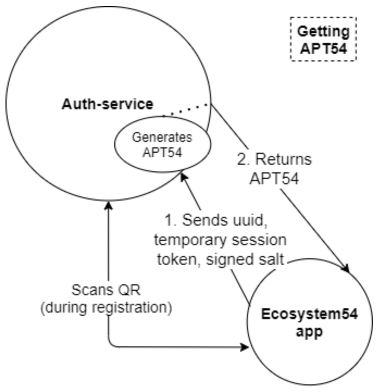
It is possible to get APT54 only when registering via QR token, i.e. with a pair of ECDSA keys.

Ecosystem54 app sends the received after registration UUID and the signed salt to the service. In response, the service generates APT54 (token), which contains the full information of the user, a list of his groups and a list of the result of checking permissions.

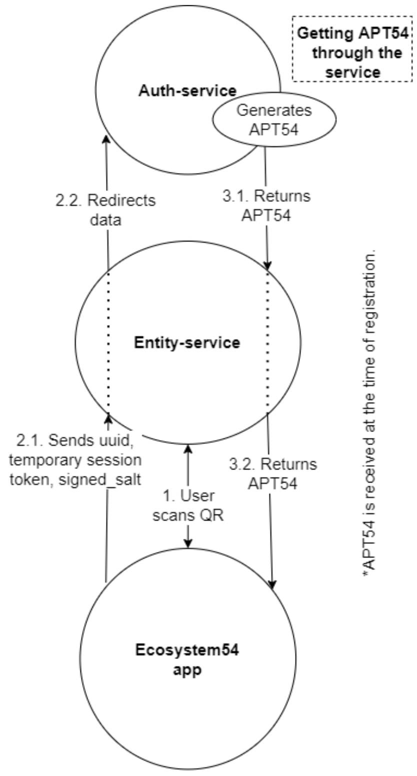
All data is signed by the Auth service, so an attempt to falsify the data will result in a signature mismatch => the client service will not accept it.

With this token the user can visit any client service, authorize there, confirm his identity and create the user on the client service.

1. Getting APT54:
   1. Ecosystem54 app sends UUID, qr_token and signet_salt to the service (the service address was in QR code).
   2. The Auth service generates APT54, which contains all information about the user, signs it with a private key and returns it to the app.


Getting APT54
Auth-service
Generates APT54
2. Returns APT54
1. Sends uuid, temporary session token, signed salt
Scans QR (during registration)
Ecosystem54 app

2. Getting APT54 through the service:
   APT54 is received at the time of registration by QR code.
   1. The user scans QR code.
   2. Ecosystem54 app sends UUID, temporary session token and signet_salt to the service (the service address was in QR code).
   3. The Entity service redirects the data to Auth.
   4. The Auth service generates APT54, which contains all information about the user, signs it with a private key and returns it to the Entity service.
   5. The Entity service receives the data and redirects it to Ecosystem54 app.

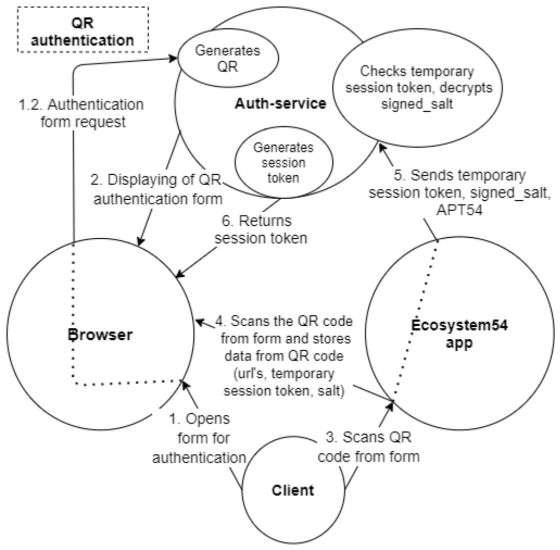

Getting APT54 through the service
Auth-service
Generates APT54
2.2. Redirects data
3.1. Returns APT54
Entity-service
2.1. Sends uuid, temporary session token, signed_salt
1. User scans QR
3.2. Returns APT54
Ecosystem54 app
*APT54 is received at the time of registration.

Authentication

Authentication is required to obtain a session token. This token is a pass to service resources that require identifying the actor or checking permissions.

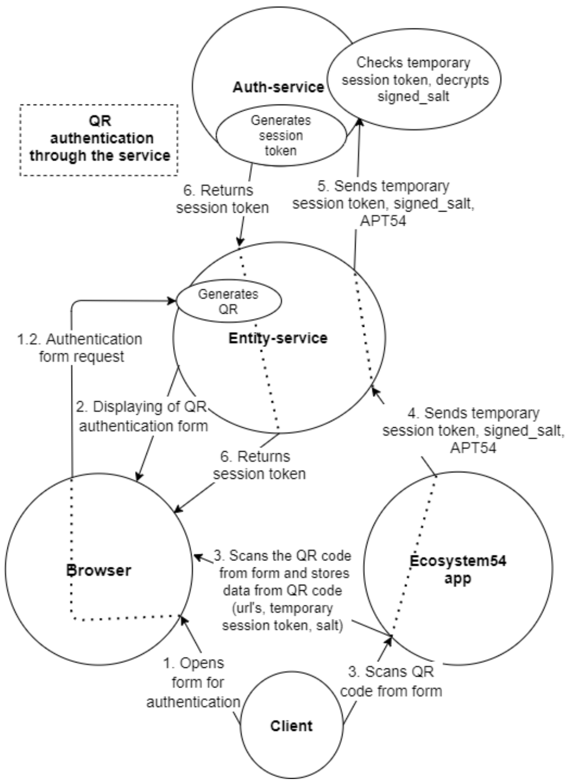
When authenticating using QR code, the user sends his APT54 and the signed salt. In response, the user receives a session token. If there is no user account on the client service, it will be created on the basis of APT54.

1. Using QR:
   1. The client opens the authentication form by QR code.
   2. The service generates QR code and displays QR code registration form in the browser.
   3. The client starts the process of scanning QR code, opens the authentication in Ecosystem54 app. The client scans the generated QR code using Ecosystem54 app. The app stores metadata from QR code (URL s, QR token, salt), signs salt (from QR code) with its private key.
   4. The app sends the client service request containing qr_token, signet_salt and APT54.
   5. The service checks qr_token, decrypts signet_salt and compares it with the one in the generated QR code. Then APT54 is decrypted and a session token is generated based on its permissions, which are stored in APT54. The service returns the session_token to the user browser and to the app. With this token the client can access the resources of the service.


QR authentication
Generates QR
1.2. Authentication form request
Auth-service
Checks temporary session token, decrypts signed_salt
Generates session token
2. Displaying of QR authentication form
5. Sends temporary session token, signed_salt, APT54
6. Returns session token
Browser
4. Scans the QR code from form and stores data from QR code (url's, temporary session token, salt)
Ecosystem54 app
1. Opens form for authentication
3. Scans QR code from form
Client

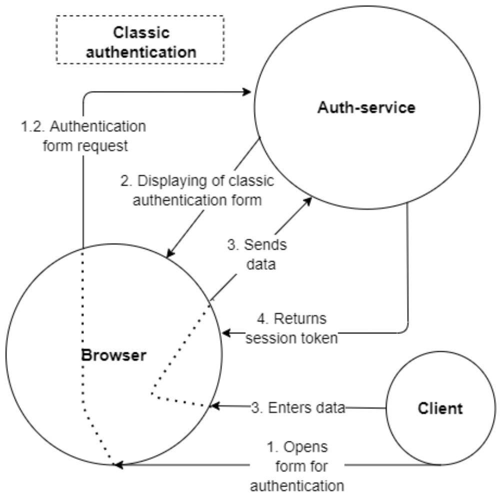
2. Using QR through the service:
   1. The client opens the authentication form by QR code.
   2. The service generates QR code and displays QR code registration form in the browser.

3. The client starts the process of scanning QR code, opens the authentication in Ecosystem54 app. The client scans the generated QR code using Ecosystem54 app. The app stores metadata from QR code (URL s, QR token, salt), signs salt (from QR code) with its private key.
4. The app sends the client service request containing qr_token, signet_salt and APT54.
5. The service redirects data to the Auth service.
6. The Auth service checks qr_token, decrypts signet_salt and compares it with the one in the generated QR code. Then APT54 is decrypted and a session token is generated based on its permissions, which are stored in APT54. The service returns the session_token to the user browser and to the app. With this token the client can access the resources of the service.


Checks temporary session token, decrypts signed_salt
Auth-service
QR authentication through the service
Generates session token
6. Returns session token
5. Sends temporary session token, signed_salt, APT54
Generates QR
1.2. Authentication form request
Entity-service
2. Displaying of QR authentication form
4. Sends temporary session token, signed_salt, APT54
6. Returns session token
3. Scans the QR code from form and stores data from QR code (url's, temporary session token, salt)
Browser
Ecosystem54 app
1. Opens form for authentication
3. Scans QR code from form
Client

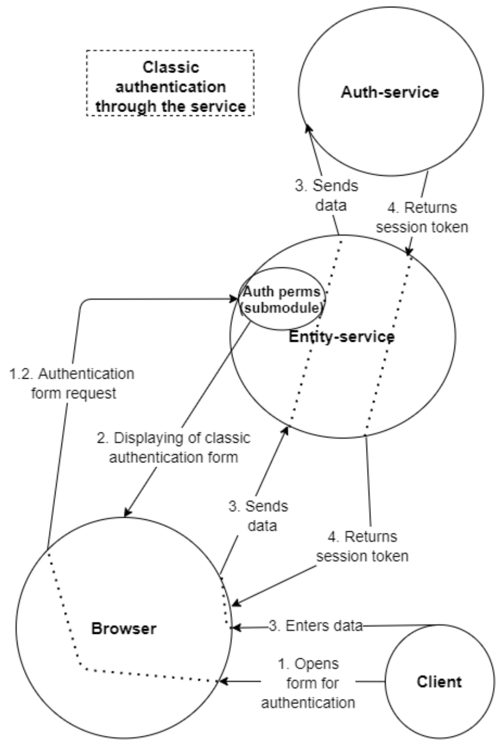
When authenticating a classic user, the user enters a login (email) and password. The Auth service checks the data and gives the user a session on its service. If the user uses SSO system, the session will be also created on the client service and the user will be redirected to the client service.

1. Classic Form Authentication:
   1. The client opens a classic form for authentication.
   2. Browser displays the registration form.
   3. The client enters his data and sends it to the service.
   4. Auth checks the data for correctness and, if successful, returns session_token, which is stored in the browser cookie. With this token the client can access the resources of the service.


Classic authentication
Auth-service
1.2. Authentication form request
2. Displaying of classic authentication form
3. Sends data
4. Returns session token
Browser
3. Enters data
Client
1. Opens form for authentication

2. Classic Form Authentication through the service:
   1. The client opens a classic form for authentication.

2. Browser sends an authentication form request to the service.
3. Browser displays the registration form.
4. The client enters his data and sends it to the service. The service redirects data to the Auth service.
5. Auth checks the data for correctness and, if successful, returns session_token, which is stored in the browser cookie. With this token the client can access the resources of the service.


Classic authentication through the service
Auth-service
3. Sends data
4. Returns session token
Auth perms (submodule)
Entity-service
1.2. Authentication form request
2. Displaying of classic authentication form
3. Sends data
4. Returns session token
Browser
3. Enters data
1. Opens form for authentication
Client

Actors

When updating actors, it will be necessary to inform the services in biome about their changes, for example, if we have added a user to any group or the user's personal information has changed. Also, when creating a new actor, such as a group, it is necessary to inform the service that a new group has appeared or the weight of the group has changed.

Client phase

The client phase involves a cycle of authorization and distribution of permissions.

Authorization

React and jinja template technologies are used as frontend applications for the service. In this case, the standard flask session module is used to manage sessions. The module encrypts token using a secret key and stores it in cookies.

Also Ecosystem54 allows storing frontend apps on a separate service, but then the app has to give the session token in Headers of the request with the Session-Token key.

Permissions

When adding, changing or deleting permissions of actor on the Auth service, the client service must be informed about these changes.

Protocol messages (Auth service)

Messages are sent over the HTTPs protocol, mostly in JSON format. Socket IO is used to get the session token.

Requests to the server are exclusively content with the right request body.

Requests on the example of user registration phases:

- Classic registration

```
Request JSON
    {
        "data": {
            "actor_type": "classic_user",
            "first_name": "Mortaldo",
            "last_name": "Gonzalez",
            "login": "morty.gonz@gmail.com",
            "password": "*****",
            "password_confirmation": "*****",
        }
    }
Response JSON
    {
        "user": "df7ea002-5edd-4a8d-b9fb-ca62a4ff6fef"
    }
```

- QR registration

```
Request JSON
    {
        "data": {
        "pub_key": "04faafa18fad0.. ..d20044",
        "qr_token": "bWTkDlHiLeHSaIMzkvQdCesFucoWqGPd",
        "signed_salt": "30450220215447bb19c420053.. ..dec"
    }
    }
Response JSON
    {
        "user": "35cf9eed-2571-4d62-9266-c01106b95343"
    }
```

- Getting APT54

```
Request JSON
    {
        "data": {
            "uuid": "35cf9eed-2571-4d62-9266-c01106b95343",
            "qr_token": "bWTkDlHiLeHSaIMzkvQdCesFucoWqGPd",
            "signed_salt": "30450220215447bb19c420053.. ..dec"
        }
    }
Response JSON
    {
        "APT54": {
            "signature": "30450.. ..edec",
            "user_data": {
                "uuid": "35cf9eed-2571-4d62-9266-c01106b95343",
                "uinfo": {
                    "email": "morty.gonz@gmail.com",
                    "groups": ["b3b803f8-ce81-4327-8b59-45550b73c65c"],
                    "weight": 4294967297,
                    "last_name": "Gonzalez",
                    "first_name": "Mortaldo"
                },
                "created": "2019-12-06 13:46:48.318271",
                "actor_type": "user",
                "initial_key": "040a5.. ..9cdf",
                "secondary_keys": null,
                "root_perms_signature": null
            },
            "expiration": "2020-04-24 12:42:10"
        },
    }
```

- Classic authentication

```
Request JSON
    {
        "data": {
            "actor_type": "classic_user",
            "email": "morty.gonz@gmail.com",
            "password": "*****"
        }
    }
Response JSON
    {
        "session_token": "5YfpZfIQEKU9Rkvnfc1IEoqYtyeXyaGL"
    }
```

- QR authentication

```
Request JSON
    {
        "data": {
            "qr_token": "bWTkDlHiLeHSaIMzkvQdCesFucoWqGPd",
            "signed_salt": "30450220215447bb19c420053.. ..dec"
            "APT54": {
                "signature": "30450.. ..edec",
                "user_data": {
                    "uuid": "35cf9eed-2571-4d62-9266-c01106b95343",
                    "uinfo": {
                        "email": "morty.gonz@gmail.com",
                        "groups": ["b3b803f8-ce81-4327-8b59-45550b73c65c"],
                        "weight": 4294967297,
                        "last_name": "Gonzalez",
                        "first_name": "Mortaldo"
                    },
                    "created": "2019-12-06 13:46:48.318271",
                    "actor_type": "user",
                    "initial_key": "040a5.. ..9cdf",
                    "secondary_keys": null,
                    "root_perms_signature": null
                },
                "expiration": "2020-04-24 12:42:10"
            }
        }
    }
Response JSON
    {
        "session_token": "5YfpZfIQEKU9Rkvnfc1IEoqYtyeXyaGL"
    }
```

- Authorization (if frontend works separately on a separate server)

```
Request headers
    {
        "Session-Token": "5YfpZfIQEKU9Rkvnfc1IEoqYtyeXyaGL"
    }
```

- Authorization (if frontend works separately on the same server)
  The session is created automatically.

Client service messages

Messages are sent over the HTTPs protocol, mostly in JSON format. Socket IO is used to get the session token.

Requests on the example of user registration phases:

- Classic registration

```
Request GET /auth_authorization/ Optional params:
    {
        "redirect_url": "/admin/auth/sign-up" or "admin/auth/sign-up"
    }
Response JSON
    {
        "domain": "https://auth.p.54origins.com/auth_sso/?uuid=SERVICE_UUID&session=TEMPORARY_SESSION&red
        "temporary_session": "iLu7WHGPBYm6XphM7PhKrHg5GsXFYqbZ"
    }
```

  Then frontend redirects to domain from response and the registration form of the classic user opens on the Auth service.

- QR registration

```
Request JSON
    {
        "data": {
            "pub_key": "04faafa18fad0.. ..d20044",
            "qr_token": "bWTkDlHiLeHSaIMzkvQdCesFucoWqGPd",
            "signed_salt": "30450220215447bb19c420053.. ..dec"
        }
    }
Response JSON
    {
        "user": "35cf9eed-2571-4d62-9266-c01106b95343"
    }
```

The registration is performed this way:

- The request is sent to the client service
- The client service does not validate whether the service is Auth service.
- Client service data is added to the request and sent to the Auth service.
- Registration is performed at the Auth service and response is returned to the client service.
- On the client service, the user is created from response that is received from Auth.
- The user receives response with UUID of the created user.

- Getting APT54

```
Request JSON
    {
        "data": {
            "uuid": "35cf9eed-2571-4d62-9266-c01106b95343",
            "qr_token": "bWTkDlHiLeHSaIMzkvQdCesFucoWqGPd",
            "signed_salt": "30450220215447bb19c420053.. ..dec"
        }
    }
Response JSON
    {
        "APT54": {
            "signature": "30450.. ..edec",
            "user_data": {
                "uuid": "35cf9eed-2571-4d62-9266-c01106b95343",
                "uinfo": {
                    "email": "morty.gonz@gmail.com",
                    "groups": ["b3b803f8-ce81-4327-8b59-45550b73c65c"],
                    "weight": 4294967297,
                    "last_name": "Gonzalez",
                    "first_name": "Mortaldo"
                },
                "created": "2019-12-06 13:46:48.318271",
                "actor_type": "user",
                "initial_key": "040a5.. ..9cdf",
                "secondary_keys": null,
                "root_perms_signature": null
            },
            "expiration": "2020-04-24 12:42:10"
        },
    }
```

To get APT54, the client service sends a request to a special URL of the Auth service, which is available only to services registered on the Auth service. In response, the client service receives APT54.

- Classic authentication

```
Request GET /auth_authorization/ Optional params:
    {
        "redirect_url": "/admin/auth/sign-in" or "admin/auth/sign-in"
    }
Response JSON
    {
        "domain": "https://auth.p.54origins.com/auth_sso/?uuid=SERVICE_UUID&session=TEMPORARY_SESSION&red
        "temporary_session": "iLu7WHGPBYm6XphM7PhKrHg5GsXFYqbZ"
    }
```

Then frontend redirects to domain from response and the authentication form of the classic user opens on the Auth service. After successful authentication a session is created for the user on backend on Auth and the client service. The user is redirected back to the client service.

- QR authentication

```
Request JSON
    {
        "data": {
            "qr_token": "bWTkDlHiLeHSaIMzkvQdCesFucoWqGPd",
            "signed_salt": "30450220215447bb19c420053.. ..dec",
            "apt54": {
                "signature": "30450.. ..edec",
                "user_data": {
                    "uuid": "35cf9eed-2571-4d62-9266-c01106b95343",
                    "uinfo": {
                        "email": "morty.gonz@gmail.com",
                        "groups": ["b3b803f8-ce81-4327-8b59-45550b73c65c"],
                        "weight": 4294967297,
                        "last_name": "Gonzalez",
                        "first_name": "Mortaldo"
                    },
                    "created": "2019-12-06 13:46:48.318271",
                    "actor_type": "user",
                    "initial_key": "040a5.. ..9cdf",
                    "secondary_keys": null,
                    "root_perms_signature": null
                },
                "expiration": "2020-04-24 12:42:10"
            }
        }
    }
Response JSON
    {
        "session_token": "5YfpZfIQEKU9Rkvnfc1IEoqYtyeXyaGL"
    }
```

- Authorization (if frontend works separately on a separate server)

```
Request headers
    {
        "Session-Token": "5YfpZfIQEKU9Rkvnfc1IEoqYtyeXyaGL"
    }
```

- Authorization (if frontend works separately on the same server)
  The session is created automatically.
- Actor update
  Available only for the Auth service because the signature is verified with the public Auth key. METHOD - PUT

```
Request JSON
    {
        "actor": {
            "uuid": "0f7aa635-53dc-4ed3-a55f-dd8a9c8c757d",
            "root_perms_signature": None,
            "initial_key": None,
            "secondary_keys": None,
            "uinfo": {
                "weight": "105",
                "group_name": "Test"
            },
            "actor_type": "group"
        },
        "service_uuid": "647d834e-11c2-4e47-a8ba-57ffc305186c",
        "signature": "3046022100c6e252a97a35225931a578335a31fba94d84c19446b2ab8fcc0468642320fbce022100e54
    }
Response JSON
    {
        "message": "Actor successfully updated"
    }
```

- Actor deleting
  Available only for the Auth service because the signature is verified with the public Auth key. METHOD - DELETE

```
Request JSON
    {
        "uuid": "4caac0cd-6927-4c15-88b4-2c7a34f5b71b",
        "actor_type": "group",
        "service_uuid": "647d834e-11c2-4e47-a8ba-57ffc305186c",
        "signature": "3046022100e1215484c9c90fb7c84ada980a7b994533bb6b7d891f3a71b679abf9ff87d79a022100a4a
    }
Response JSON
    {
        "message": "Actor successfully deleted"
    }
```

- Creation (installation, updating) of a permission
  Available only for the Auth service because the signature is verified with the public Auth key. METHOD - content

```
Request JSON
    {
        "perm": {
            "action_id": ""39adc9dc-c008-11e9-9cb5-2a2ae2dbcce4/UpdateEntityAction",
            "actor_id": "a8511292-5db6-42c4-b10f-9977adfd9759",
            "default_value": 0,
            "description": "Change entity owner",
            "perm_id": "39adc9dc-c008-11e9-9cb5-2a2ae2dbcce4/UpdateEntityAction/biom_perm001",
            "perm_type": "check",
            "perm_value": 0,
            "service_id": "39adc9dc-c008-11e9-9cb5-2a2ae2dbcce4"
        },
        "actor": {
            "uuid": "a8511292-5db6-42c4-b10f-9977adfd9759",
            "created": "2020-04-15 10:24:51.117446",
            "root_perms_signature": None,
            "initial_key": None,
            "secondary_keys": None,
            "uinfo": {
                "weight": "105",
                "group_name": "Test"
            },
            "actor_type": "group"
        },
        "service_uuid": "647d834e-11c2-4e47-a8ba-57ffc305186c",
        "signature": "3045022100cc17b20f4413060f6fc0213029968042a12b910fc87780ac3dc3c4f69439d0e802205067f
    }
Response JSON
    {
        "message": "Permissions successfully updated"
    }
```

- Removing permission
  Available only for the Auth service because the signature is verified with the public Auth key. METHOD - DELETE

```
Request JSON
    {
        "permission": {
            "service_id": "39adc9dc-c008-11e9-9cb5-2a2ae2dbcce4",
            "perm_id": "39adc9dc-c008-11e9-9cb5-2a2ae2dbcce4/UpdateEntityAction/biom_perm001",
            "actor_id": "a8511292-5db6-42c4-b10f-9977adfd9759",
            "perm_value": 0,
            "default_value": 0,
            "perm_type": "check",
            "action_id": "39adc9dc-c008-11e9-9cb5-2a2ae2dbcce4/UpdateEntityAction",
            "description", "Change entity owner",
            "created", None
        },
        "service_uuid": "647d834e-11c2-4e47-a8ba-57ffc305186c",
        "signature": "304402202c9c8f980aef3012dccb6978eefaa00f6099b119baf59a0949bb0016f200d9a0022034bde52
    }
Response JSON
    {
        "message": "Permissions successfully deleted"
    }
```

- Auth SSO
  Getting URL for redirect to Auth with METHOD GET information is described in points 1 and 4 (registration, authentication) of the current block. METHOD - content
  Creation of a session on the client service by Auth service.

```
Request JSON
    {
        "apt54": {
            "signature": "30450.. ..edec",
            "user_data": {
                "uuid": "35cf9eed-2571-4d62-9266-c01106b95343",
                "uinfo": {
                    "email": "morty.gonz@gmail.com",
                    "groups": ["b3b803f8-ce81-4327-8b59-45550b73c65c"],
                    "weight": 4294967297,
                    "last_name": "Gonzalez",
                    "first_name": "Mortaldo"
                },
                "created": "2019-12-06 13:46:48.318271",
                "actor_type": "user",
                "initial_key": "040a5.. ..9cdf",
                "secondary_keys": null,
                "root_perms_signature": null
            },
            "expiration": "2020-04-24 12:42:10"
        },
        "signature": "30450.. ..asdg",
        "temporary_session": "Pk03V2ANPBKPsYmzOMFHNlIUE8DYQBUW"
    }
```

- QR code generator
  Built-in to the Ecosystem54 service ability to generate an individual QR code of the service for registration/authorization on request. The result is an image in base64 format and QR token. METHOD - GET

```
Response JSON
    {
        "qr_code": "base64",
        "qr_token": "VeFwJTRXwS21p5RUNEt7k0rnZipRjD0g"
    }
```

- Session saving
  This method encrypts and saves the encrypted session in the client browser cookies for subsequent authorization on services. It uses session_token received after authentication as a parameter. METHOD - content

```
Request JSON
    {
        "session_token": "VeFwJTRXwS21p5RUNEt7k0rnZipRjD0g"
    }
Response JSON
    {
        "message": "Session token saved"
    }
```

- Description method
  Allows you to find out brief information about the current service, where the request was received. As a result of the request you can get the name of the service, its UUID and the name of the current biome. METHOD - GET

```
Response JSON
    {
        "service_name": "client_service",
        "service_uuid": "78de748a-6922-4f04-90fc-a189d9051db7",
        "biom_name": "ecosystem54_demo"
    }
```

- Built-in login/registration template
  Provides a basic template for user registration and authorization, with js libraries and debugged css styles. If necessary, it can be redefined, modified or refused. The template uses libraries jquery, socketio, notifyjs. METHOD - GET

```
Response JSON
Endpoint '/authorization/'
```

  Will display the auth.html template with reported parameters (qr_code, qr_token, services, scripts).

03_auth_perms submodule

Most of the protocol functions are performed by Submodule project 03_auth_perms Ecosystem54.

Basic route and view

/reg/ - RegistrationView

/apt54/ - APT54View

/perms/ - PermissionView

/auth/ - ClientAuthView

/actor/ - ActorView

/get_qr_code/ - QRCodeView

/about/ - AboutView

/auth_authorization/ - AuthSSOView

/get_invite_link_info/ - GetInviteLinkInfoView

/save_session/ - SaveSession

/authorization/ - AuthorizationView

RegistrationView

Registration with Auth service, responsible for the registration process in the biome.

APT54View

Authentication with getting APT54, here the authentication process takes place through APT54.

ClientAuthView

Here the authentication process on the client service is implemented, using QR code.

PermissionView

It is responsible for creating permissions in database that was set on Auth.

ActorView

Endpoint for create/update/delete actors that was changed on Auth service.

QRCodeView

This data is stored in a QR image:

```
data = dict(
    qr_token=qr_token,                  # QR token
    salt=salt,                              # random generated hex string
    domain=service_domain,              # service domain where authentication takes place
    biom_uuid=auth_uuid.get("uuid")      # uuid current biome
)
```

As a result, qr image is converted to a base64 string

Then this response is formed:

```
response = {
    "qr_code": base64.b64encode(img_io.getvalue()).decode(), # qr is converted to a base64 string
    "qr_token": qr_token                                     # generated qr token
 }
```

AboutView

Provides biome and current service information

This is the response:

```
response = {
    "service_name": service_info.get("service_name", "Unknown"),  # current service name from db
    "service_uuid": service_info.get("uuid", "Unknown"),          # current service uuid from db
    "biom_name":  service_info.get("biom_name", "Unknown")        # current ecosystem54 biome name
}
```

AuthSSOView

Creation of a redirect URL and temporary session for authorization through the Auth service.

```
response = {
    “domain”: Урл на аус сервис с параметрами,
    “temporary_session”: Временная сессия.
}
```

GetInviteLinkInfoView

Delete?

SaveSession

Entry of received session in cookie with token encryption, which is performed by the standard flask session module using APP_SECRET_KEY. Used with jinja template.

AuthorizationView

General login template.

Flask application configuration

The Ecosystem54 service requires a specific set of variables and their values, which are in the Flask application configuration. The settings.py file acts as a repository for the variables. It is recommended to create your own local_settings.py to describe the values of the variables, in turn, settings.py inherits their values.

It is important to note!

local_settings.py is in .gitignore.

Below is a complete list of variables and their detailed descriptions.

local_settings.py example:

```
# Database credentials.
# ! Required
DATABASE = {
"ENGINE": "contentgresql",
"NAME": "",
"USER": "",
"PASSWORD": "",
"HOST": "localhost",
}
# Database URI for database connection.
# ! Optional
DATABASE_URI = "{ENGINE}://{USER}:{PASSWORD}@{HOST}/{NAME}".format(**DATABASE)
# App secret key for signing session for example.
# ! Required
APP_SECRET_KEY = None
# Debug mode
DEBUG = True
# Service name. Need for register service in database by control service.
# ! Required
SERVICE_NAME = "YOUR SERVICE NAME"
# Auth service public key. This public key will be    automatically added in local_settings by control serv
# For local purposes use public key from your local auth service.
# ! Required
AUTH_PUBLIC_KEY = ""
# !!! Note !!!
# SERVICE_PRIVATE_KEY and SERVICE_PUBLIC_KEY should be a pair of ecdsa secp256k1.

# Your service private key.
# ! Required
SERVICE_PRIVATE_KEY = ""
# Your service public key.
# ! Required
SERVICE_PUBLIC_KEY = ""
# Current service uuid with which one it was registered on auth service. This will be automatically added by
# service.
# ! Required
SERVICE_UUID = "bb789f26-5376-4dc1-8885-686d4d8dcb15"
# This flag used to choose what key to use in signature verification (initial_key or secondary_keys).
# If set True use only initial_key for signature verification (used on important services)
# If set False check with initial_key first, then if verification failed, check with secondary_keys.
# ! Required
PRIMARY_KEY_ONLY = False
# In what group after registration user should be in. If no value will use auth default group. Group should
# on auth and on your service. Can not be ADMIN group.
# ! Optional
DEFAULT_GROUP_NAME = "DEFAULT"
# Where session will be saved and service will try get it. Can be "HEADERS", "SESSION", None.
# If set "HEADERS" - session token should be send in request.headers with key Session-Token in every request
# If set "SESSION" - use default flask session method by saving signed token with APP_SECRET_KEY, by key "se
# If set None - will check request.headers first, then flask session by key "session_token"
# ! Optional
SESSION_STORAGE = "SESSION"
# Auth domain what is used to send data on getting apt54/sending service permissions on auth. Will be added
# service.
# TODO: will be removed soon and getting from database.
# ! Required
AUTH_DOMAIN = None
# Need to save in database when your service register on your service. Will be added by control service.
# ! Required
SERVICE_DOMAIN = ""
# What socket mode need to use. Set gevent if deploying by control service on live (uwsgi or gunicorn).
# Set None if using Werkzeug (local)
# ! Required
SOCKET_ASYNC_MODE = "gevent"
```