DAI - PW3 - API documentation

Pedro Alves da Silva Gonçalo Carvalheiro Heleno

Contents

Endpoint template

This section contains the template that is used in the endpoints list section. It describes what each part does, and how to fill it.

`POST /example_endpoint`

A description of this endpoint will be available here. The requirements are listed in this description, as well as what its goal is.

Parameters

The table below lists all the mandatory and optional requirements, with a brief description of each one of them. **Typically, a path parameter is mandatory, while query parameters are optional.**

Parameter	Optional?	Description
`username`	✓	Username to be used for this dummy request. Does not affect output.

Responses

This section lists the possible HTTP status codes the clients may receive when using this endpoint. Server-side status codes (`5xx`) are excluded from the list.

When using both the `POST`:

- `200`: Call was successful
- `400`: Invalid parameters
- `401`: User not logged

Endpoints summary

This table summarizes the accepted HTTP request methods by endpoint.

Endpoint URL	GET	POST	PUT	DELETE
`/users`	✓	✓		
`/users/{username}`	✓		✓	✓
`/gpg-keys`	✓	✓		
`/gpg-keys/{fingerprint}`	✓		✓	✓
`/emails`	✓			
`/emails/{username}`	✓	✓		✓

Users

This section lists all the endpoints available to access, update, delete, and / or create resources related to users.

`GET /users`

Allows a client to obtain the list of users that are known by the server.

Parameters

Parameter	Optional?	Description
`firstName`	✓	Filters the list of users by their first name.
`lastName`	✓	Filters the list of users by their last name.

The search is case-insensitive and partial matches are allowed. If both `firstName` and `lastName` are provided, the server will return only the users that match both criteria.

Responses

The server will always return `200`. It returns the list of users with their username and names in a JSON list format or an empty list is returned if no users are found.

Example usage

Request:

```
GET /users
```

Response:

```
200 OK
```

```json
[
  {
    "username": "john.doe",
    "firstName": "John",
    "lastName": "Doe"
  },
  {
    "username": "jane.doe",
    "firstName": "Jane",
    "lastName": "Doe"
  }
]
```

Request:

```
GET /users?firstName=John
```

Response:

```
200 OK
```

```json
[
  {
    "username": "john.doe",
    "firstName": "John",
    "lastName": "Doe"
  }
]
```

`POST /users`

Creates a new user in the server.

Parameters

No HTTP path or query parameters are needed.

Request

The request body must contain a JSON object with the following fields:

- `username`: username to be used for this user
- `firstName`: user’s first name
- `lastName`: user’s last name

Responses

The server returns the following status codes:

- `201`: user created successfully
- `400`: username is malformed (must contain only alphanumeric characters, dots, and underscores)
- `409`: username has already been taken

Only a status code with an empty body is returned.

Example with valid data

Request:

```
POST /users
[...]

{
    "username": "john.doe",
    "firstName": "John",
    "lastName": "Doe"
}
```

Response:

```
201 Created
```

Example with malformed username

```
POST /users
[...]

{
    "username": "j$hn:doe",
    "firstName": "John",
    "lastName": "Doe"
}
```

Response:

```
400 Bad Request
```

Example with username that has already been taken (attempt to recreate)

```
POST /users
[...]

{
    "username": "john.doe",
    "firstName": "John",
    "lastName": "Doe"
}
```

Response:

```
409 Conflict
```

`GET /users/{username}`

Allows a client to obtain information about the provided user.

Parameters

Parameter	Optional?	Description
`username`		Shows information related to this user.

Responses

The server returns the following status codes:

- `200`: user was found
- `400`: username is malformed
- `404`: user does not exist

When successful, it returns a JSON object containing the user's information:

```
{
    "username": "john.doe",
    "firstName": "John",
    "lastName": "Doe"
}
```

Example with valid username

Request:

`GET /user/john.doe`

Response:

`200 OK`

```
{
  "username": "john.doe",
  "firstName": "John",
  "lastName": "Doe"
}
```

Example with malformed username

`GET /users/malf:or$med`

Response:

`400 Bad Request`

Example with username that doesn't exist

`GET /users/unknown.user`

Response:

`404 Not Found`

`PUT /users/{username}`

Allows updating the `firstName` or `lastName` attributes of specific user.

The username is not allowed to be changed and if required, instead a new user should be created, followed by moving the emails from the old user to the new one, then deleting the old user.

Parameters

Parameter	Optional?	Description
`username`		Username of the user to be edited.

Request

The request body must contain a JSON object with the following fields:

- `firstName`: user's first name
- `lastName`: user's last name

Responses

The server returns the following status codes:

- 200: user was edited
- 400: username is malformed
- 404: user does not exist

Only a status code with an empty body is returned.

Example with valid data

Request:

```
PUT /users/john.doe
[...]

{
    "firstName": "John",
    "lastName": "DoeNew"
}
```

Response:

```
200 OK
```

```
{
    "username": "john.doe",
    "firstName": "John",
    "lastName": "DoeNew"
}
```

Example with malformed username

```
PUT /users/malf:or$med
[...]

{
    "firstName": "John",
    "lastName": "Doe"
}
```

Response:

```
400 Bad Request
```

Example with user that does not exist

```
PUT /users/unknown.user
[...]

{
    "firstName": "John",
    "lastName": "Doe"
}
```

Response:

```
404 Not Found
```

`DELETE /users/{username}`

Deletes a specific user.

Deleting a user also deletes all emails associated to this email, as well as any key - email pairs that were exclusively used attributed to this user's email addresses.

Parameters

Parameter	Optional?	Description
`username`		Username of the user to be deleted.

Responses

The server returns the following status codes:

- `200`: user was deleted
- `400`: username is malformed
- `404`: user does not exist

Only a status code with an empty body is returned.

Example with valid username

Request:

```
DELETE /users/john.doe
```

Response:

```
200 OK
```

```json
{
    "username": "john.doe",
    "firstName": "John",
    "lastName": "Doe"
}
```

Example with malformed username

```
DELETE /users/malf:or$med
```

Response:

```
400 Bad Request
```

Example with user that does not exist

```
DELETE /users/unknown.user
```

Response:

```
404 Not Found
```

GPG keys

This section lists and describes the endpoints that allows a client to interact with the GPG keys stored in the server.

`GET /gpg-keys`

Shows a list of all stored public GPG keys, along with the respective fingerprint.

Parameters

No HTTP path or query parameters are needed.

Responses

Endpoint may only return `200` to confirm the request was successful.

Response is in a JSON format. Example request and response is:

Request:

```
GET /gpg-keys
```

Response:

```
200 OK
```

```json
[
  {
    "fingerprint": "AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTT",
    "key": "-----BEGIN PGP PUBLIC KEY BLOCK-----\n\n[...]\n-----END PGP PUBLIC KEY BLOCK-----"
  },
  {
    "fingerprint": "UUVVWWXXYYZZ00112233445566778899AABBCCDD",
    "key": "-----BEGIN PGP PUBLIC KEY BLOCK-----\n\n[...]\n-----END PGP PUBLIC KEY BLOCK-----"
  }
]
```

POST /gpg-keys

Adds a new key to the database.

Parameters

No HTTP path or query parameters are needed.

Request

The request body must contain a JSON object with the following fields:

- `fingerprint`: the fingerprint of the key in hexadecimal format
- `key`: the key itself in ASCII-armored format

Responses

- `201`: request was successful and the key was created
- `400`: either the fingerprint or the key is invalid
- `409`: the fingerprint is already present in the database

Only a status code with an empty body is returned.

Example with valid key

Request:

```
POST /gpg-keys
[...]

{
    "fingerprint": "AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTT",
    "key": "-----BEGIN PGP PUBLIC KEY BLOCK-----\n\n[...]\n-----END PGP PUBLIC KEY BLOCK-----"
}
```

Response:

```
201 Created
```

Example with invalid key

Request:

```
POST /gpg-keys
[...]

{
    "fingerprint": "AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTT",
    "key": "toto--titi-tata"
}
```

Response:

```
400 Bad Request
```

Example with key that's already present

Request:

```
POST /gpg-keys
[...]

{
    "fingerprint": "AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTT",
    "key": "-----BEGIN PG PUBLIC KEY BLOCK-----\n\n[...]\n-----END PGP PUBLIC KEY BLOCK-----"
}
```

Response:

```
409 Conflict
```

GET /gpg-keys/{fingerprint}

Allows a client to obtain the private key associated with the provided fingerprint.

Parameters

Parameter	Optional?	Description
`fingerprint`		Fingerprint of the GPG key the client want to obtain.

Responses

- 200: request was successful
- 400: fingerprint is invalid
- 404: fingerprint does not exist

When successful, it returns a JSON object containing the key's information:

```json
{
    "fingerprint": "AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTT",
    "key": "-----BEGIN PGP PUBLIC KEY BLOCK-----\n\n[...]\n-----END PGP PUBLIC KEY BLOCK-----"
}
```

Example with a valid fingerprint

Request:

```
GET /gpg-keys/AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTT
```

Response:

```
200 OK
```

```json
{
    "fingerprint": "AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTT",
    "key": "-----BEGIN PGP PUBLIC KEY BLOCK-----\n\n[...]\n-----END PGP PUBLIC KEY BLOCK-----"
}
```

Example with invalid fingerprint

Request:

```
GET /gpg-keys/invalid-fingerprint
```

Response:

```
400 Bad Request
```

Example with fingerprint that doesn't exist

Request:

```
GET /gpg-keys/unknown-fingerprint
```

Response:

```
404 Not Found
```

`PUT /gpg-keys/{fingerprint}`

Allows updating the `key` attributes of a specific key.

Parameters

Parameter	Optional?	Description
`fingerprint`		Fingerprint of the GPG key the client want to obtain.

Request

The request body must contain a JSON object with the following fields:

- `key`: the new key

Responses

The server returns the following status codes:

- `200`: key was edited
- `400`: fingerprint is malformed
- `404`: key does not exist

Only a status code with an empty body is returned.

Example with valid data

Request:

```
PUT /gpg-keys/AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTT
[...]

{
    "key": "-----BEGIN PGP PUBLIC KEY BLOCK-----\n\n[...]\n-----END PGP PUBLIC KEY BLOCK-----"
}
```

Response:

```
200 OK
```

Example with invalid fingerprint

```
PUT /fingerprint/invalid-fingerprint
[...]

{
    "key": "-----BEGIN PGP PUBLIC KEY BLOCK-----\n\n[...]\n-----END PGP PUBLIC KEY BLOCK-----"
}
```

Response:

```
400 Bad Request
```

Example with key that doesn't exist

```
PUT /gpg-keys/unknown-fingerprint
[...]

{
    "key": "-----BEGIN PGP PUBLIC KEY BLOCK-----\n\n[...]\n-----END PGP PUBLIC KEY BLOCK-----"
}
```

Response:

```
404 Not Found
```

`DELETE /gpg-keys/{fingerprint}`

Deletes a key from the server.

Parameters

Parameter	Optional?	Description
`fingerprint`		Fingerprint of the GPG key to delete.

Responses

- `200`: key was deleted
- `400`: fingerprint is invalid
- `404`: key does not exist

Only a status code with an empty body is returned.

Example with valid GPG key

Request:

```
DELETE /gpg-keys/AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTT
```

Response:

```
200 OK
```

Example with invalid fingerprint

Request:

```
DELETE /gpg-keys/invalid-fingerprint
```

Response:

```
400 Bad Request
```

Example with key that doesn't exist

Request:

```
DELETE /gpg-keys/unknown-fingerprint
```

Response:

```
404 Not Found
```

Emails

This last section lists all the endpoints that allow users to attach emails to, or detach emails from them.

`GET /emails`

Allows a client to obtain the list of all or part of all known emails.

Parameters

No HTTP path or query parameters are needed.

Responses

Server always replies with `200` to indicate the list retrieved successfully, even if no emails are known.

Response is in a JSON list format like the one below:

```json
[
  {
    "email": "john.doe@example.com",
    "username": "john.doe"
  },
  {
    "email": "john.doe@home-email.com",
    "username": "john.doe"
  },
  {
    "email": "jane.doe@example.com",
    "username": "jane.doe"
  }
]
```

Example usage

Retrieves all emails

Request:

```
GET /emails
```

Response:

```
200 OK
```

```json
[
  {
    "email": "john.doe@example.com",
    "username": "john.doe"
  },
  {
    "email": "john.doe@home-email.com",
    "username": "john.doe"
  },
  {
    "email": "jane.doe@example.com",
    "username": "jane.doe"
  }
]
```

`GET /emails/{username}`

Shows a list of all the emails addresses associated with a specific user.

Parameters

Parameter	Optional?	Description
`username`		Username to use to retrieve all emails associated with.

Responses

- `200`: request was successful
- `400`: invalid username (malformed username)
- `404`: unknown username

Only a status code with an empty body is returned.

```
[
  {
    "email": "john.doe@example.com",
    "username": "john.doe"
  },
  {
    "email": "john.doe@home-email.com",
    "username": "john.doe"
  }
]
```

Example with a valid username

Request:

```
GET /emails/john.doe
```

Response:

```
200 OK
```

```
[
  {
    "email": "john.doe@example.com",
    "username": "john.doe"
  },
  {
    "email": "john.doe@home-email.com",
    "username": "john.doe"
  }
]
```

Example with a username that has no emails attached to it

Request:

```
GET /emails/empty.user
```

Response:

```
200 OK
```

```
[]
```

Example with malformed username

Request:

```
GET /emails/malf:or$med
```

Response:

```
400 BAD
```

```
Bad Request
```

Example with an unknown user specified

Request:

```
GET /emails/unknown.user
```

Response:

```
404 NOT FOUND
```

```
Not Found
```

`POST /emails/{username}`

Associates an email address to a user.

Parameters

Parameter	Optional?	Description
`username`		Username to associate email to.

Request

The request body must contain a JSON object with the following fields:

- `email`: user's email address

Responses

- 201: association created
- 400: invalid username (malformed username)
- 404: unknown username
- 409: email address has already been associated to a user

Only a status code with an empty body is returned.

Example with a valid username and email

Request:

```
POST /emails/john.doe
[...]

{
  "email": "john.doe@example.com"
}
```

Response:

```
200 OK
```

Example with an invalid username

Request:

```
POST /emails/malf:or$med
[...]

{
  "email": "john.doe@example.com"
}
```

Response:

```
400 Bad Request
```

Example with an invalid email address

Request:

```
POST /emails/john.doe
[...]

{
  "email": "john.doe@"
}
```

Response:

```
400 Bad Request
```

Example with an unknown user

Request:

```
POST /emails/unknown.user
[...]

{
  "email": "john.doe@example.com"
}
```

Response:

```
404 Not Found
```

Example with an email that's already been attached to another user

Request:

```
POST /emails/john.doe
[...]

{
  "email": "john.doe@example.com"
}
```

Response:

```
409 Conflict
```

`DELETE /emails/{username}`

Detaches an email address from a user. This also removes any key - email pairs for keys that are no longer attached to any email address.

Parameters

Parameter	Optional?	Description
`username`		Username to detach an email from.

Request

The request body must contain a JSON object with the following fields:

- `email`: user's email address to be deleted

Responses

- `204`: Email has been detached from user
- `400`: Invalid username (malformed username)
- `404`: Unknown username or email address is not attached to user

Only a status code with an empty body is returned.

Example with a valid username and email

Request:

```
DELETE /emails/john.doe
[...]

{
  "email": "john.doe@example.com"
}
```

Response:

```
204 NO CONTENT
```

Example with an invalid username

Request:

```
DELETE /emails/malf:or$med
[...]

{
  "email": "john.doe@example.com"
}
```

Response:

```
400 BAD
```

```
Bad Request
```

Example with an invalid email address

Request:

```
DELETE /emails/john.doe
[...]

{
  "email": "john.doe@"
}
```

Response:

```
400 BAD
```

```
Bad Request
```

Example with an unknown user

Request:

```
DELETE /emails/unknown.user
[...]

{
  "email": "john.doe@example.com"
}
```

Response:

```
404 NOT FOUND
```

```
Not Found
```

Example with an email that's not attached to the user

Request:

```
DELETE /emails/john.doe
[...]

{
  "email": "jane.doe@example.com"
}
```

Response:

```
404 NOT FOUND
```

```
Bad Request
```