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

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
`/gpg/list`	✓			
`/gpg/by-email`	✓	✓		✓
`/gpg/by-username`	✓			
`/users`	✓			
`/user/by-username`	✓	✓	✓	✓
`/emails`	✓			
`/email/by-username`	✓	✓		✓

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/list`

Shows a list of all stored public GPG keys, along with the user and email they belong to.

Parameters

This endpoint does not take any parameters.

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/list
```

Response:

```
200 OK
```

```
[
  {
    "username": "john.doe",
    "user": "John Doe",
    "email": "john.doe@example.com",
    "key": "-----BEGIN PGP PUBLIC KEY BLOCK-----\n\n[...]\n-----END PGP PUBLIC KEY BLOCK-----"
  },
  {
    "username": "jane.doe",
    "user": "Jane Doe",
    "email": "jane.doe@example.com",
    "key": "-----BEGIN PGP PUBLIC KEY BLOCK-----\n\n[...]\n-----END PGP PUBLIC KEY BLOCK-----"
  }
]
```

`GET /gpg/by-email`

Shows a list of all the stored public GPG keys belonging to a specific email address.

Parameters

Parameter	Optional?	Description
`email`		Only shows GPG keys belonging to this email when retrieving the keys associated with it

Responses

- `200`: Request was successful
- `400`: Invalid email (malformed email)

Response is in a JSON format.

Example with a valid email

Request:

```
GET /gpg/by-email?email=john.doe@example.com
```

Response:

```
200 OK
```

```
[
  {
    "username": "john.doe",
    "user": "John Doe",
    "email": "john.doe@example.com",
    "key": "-----BEGIN PGP PUBLIC KEY BLOCK-----\n\n[...]\n-----END PGP PUBLIC KEY BLOCK-----"
  },
  {
    "username": "john.doe",
    "user": "John Doe",
    "email": "john.doe@example.com",
    "key": "-----BEGIN PGP PUBLIC KEY BLOCK-----\n\n[...]\n-----END PGP PUBLIC KEY BLOCK-----"
  }
]
```

Example with email that has no keys attached to it

If the provided email address is valid, but has no GPG keys attributed to it, an empty JSON list is returned:

Request:

```
GET /gpg/by-email/?email=unknown.user@example.com
```

Response:

```
200 OK
```

```
[]
```

Example with malformed email

Request:

```
GET /gpg/by-email/?email=unknown.user@
```

Response:

```
400 BAD
```

```
[]
```

`POST /gpg/by-email`

Creates a key - email pair.

Parameters

Parameter	Optional?	Description
`email`		The email to attach the key to
`key`		Key to attach to the email

Responses

- `201`: Request was successful, and pair was created
- `400`: Invalid email (malformed email) or GPG key (malformed key)

Response is in the form of a JSON dictionary. The dictionary is empty on successful requests. It contains the error message in case of an unsuccessful request:

```
{
  "error": "Error message"
}
```

Example with valid key

Request:

```
POST /gpg/by-email
[...]

{
    "email": "john.doe@example.com",
    "key": "-----BEGIN PGP PUBLIC KEY BLOCK-----\n\n[...]\n-----END PGP PUBLIC KEY BLOCK-----"
}
```

Response:

```
201 CREATED
```

```
{}
```

Example with invalid email

Request:

```
POST /gpg/by-email
[...]

{
    "email": "john.doe@",
    "key": "-----BEGIN PGP PUBLIC KEY BLOCK-----\n\n[...]\n-----END PGP PUBLIC KEY BLOCK-----"
}
```

Response:

```
400 BAD
```

```
{
    "error": "Invalid email address"
}
```

Example with invalid key

Request:

```
POST /gpg/by-email
[...]

{
    "email": "john.doe@example.com",
    "key": "-----BEGIN PUBLIC KEY BLOCK-----\n\n[...]\n-----END PGP PUBLIC KEY BLOCK-----"
}
```

Response:

```
400 BAD
```

```
{
    "error": "Invalid GPG key"
}
```

`DELETE /gpg/by-email`

Deletes a key - email pair from the server.

Parameters

Parameter	Optional?	Description
`email`		Email to detach a key from
`key`		Key to detach

Responses

- 200: Request was successful
- 404: Key does not exist, or is not attributed to the provided email

Example with valid GPG key

Request:

```
DELETE /gpg/by-email
[...]

{
"email": "john.doe@example.com",
"key": "-----BEGIN PUBLIC KEY BLOCK-----\n\n[...]\n-----END PGP PUBLIC KEY BLOCK-----"
}
```

Response:

```
200 OK
```

```
[]
```

Example with invalid email - key pair

Request:

```
DELETE /gpg/by-email
[...]

{
"email": "john.doe@example.com",
"key": "-----BEGIN GPG PUBLIC KEY BLOCK-----\n\n[...]\n-----END PGP PUBLIC KEY BLOCK-----"
}
```

Response:

```
400 BAD
```

```
{
    "error": "GPG key not attributed to provided email"
}
```

`GET /gpg/by-username/{username}`

Shows a list of all the stored public GPG keys belonging to a specific email address.

Parameters

Parameter	Optional?	Description
`username`		Only shows GPG keys belonging to this user

Responses

- 200: Request was successful
- 400: Invalid username (malformed username)

Response is in a JSON format.

Example with valid user

Request:

```
GET /gpg/by-username/john.doe
```

Response:

```
200 OK
```

```
[
  {
    "username": "john.doe",
    "user": "John Doe",
    "email": "john.doe@example.com",
    "key": "-----BEGIN PGP PUBLIC KEY BLOCK-----\n\n[...]\n-----END PGP PUBLIC KEY BLOCK-----"
  },
  {
    "username": "john.doe",
    "user": "John Doe",
    "email": "john.doe@home-email.com",
    "key": "-----BEGIN PGP PUBLIC KEY BLOCK-----\n\n[...]\n-----END PGP PUBLIC KEY BLOCK-----"
  }
]
```

Example with valid user, no emails

If the provided username is valid, but user has no emails, an empty JSON list is returned:

Request:

```
GET /gpg/by-username/unknown.user
```

Response:

```
200 OK
```

```
[]
```

Example with invalid username

As described, if the username is malformed, a specific HTTP status code is returned. Furthermore, and empty JSON list returned:

Request:

```
GET /gpg/by-username/malf:or$med
```

Response:

```
400 BAD
```

```
[]
```

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 user.

Parameters

This endpoint does not take any parameters

Responses

The server will always return `200`. It returns the list of users with their username and full name in a JSON list format.

Example usage

Request:

```
GET /users
```

Response:

```
200 OK
```

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

`GET /user/by-username/{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 dictionary containing the user's information:

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

Example with valid username

Request:

```
GET /user/by-username/john.doe
```

Response:

```
200 OK
```

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

Example with username that doesn't exist

```
GET /user/by-username/unknown.user
```

Response:

```
404 NOT FOUND
```

```
{}
```

Example with malformed email

```
GET /user/by-username/malf:or$med
```

Response:

```
400 BAD
```

```
{}
```

`POST /user/by-username/{username}`

Creates a user so emails can be attached to it.

Parameters

Parameter	Optional?	Description
`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
- `409`: Username has already been taken

It always returns an empty JSON list: `[]`

Example with valid data

Request:

```
POST /user/by-username/john.doe
[...]

{
    "firstname": "John",
    "lastname": "Doe"
}
```

Response:

```
200 OK

[]
```

Example with malformed username

```
POST /user/by-username/malf:or$med
[...]

{
    "firstname": "John",
    "lastname": "Doe"
}
```

Response:

```
400 BAD

[]
```

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

```
POST /user/by-username/jane.doe
[...]

{
    "firstname": "John",
    "lastname": "Doe"
}
```

Response:

```
409 CONFLICT

[]
```

PUT /user/by-username/{username}

Updates data about a specific user

Parameters

Parameter	Optional?	Description
`username`		Username to be used for this user
`firstname`		Updated user's first name
`lastname`		Updated 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

Always returns an empty JSON list: `[]`

Example with valid data

Request:

```
PUT /user/by-username/john.doe
[...]

{
    "firstname": "John",
    "lastname": "Doe"
}
```

Response:

```
200 OK

[]
```

Example with malformed username

```
PUT /user/by-username/malf:or$med
[...]

{
    "firstname": "John",
    "lastname": "Doe"
}
```

Response:

```
400 BAD

[]
```

Example with user that does not exist

```
PUT /user/by-username/unknown.user
[...]

{
    "firstname": "John",
    "lastname": "Doe"
}
```

Response:

```
404 NOT FOUND

[]
```

DELETE /user/by-username/{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 to be deleted

Responses

The server returns the following status codes:

- `200`: User was deleted
- `400`: Username is malformed
- `404`: User does not exist

Always returns an empty JSON list: `[]`

Example with valid data

Request:

```
DELETE /user/by-username/john.doe
```

Response:

```
200 OK

[]
```

Example with malformed username

```
DELETE /user/by-username/malf:or$med
```

Response:

```
400 BAD

[]
```

Example with user that does not exist

```
DELETE /user/by-username/unknown.user
```

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

Endpoint does not take any parameters.

Responses

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

Response if 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 /email/by-username/{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

Response, if an error occurred, is an empty JSON list: `[]`. If successful, response if in a JSON list format like the one below:

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

Example with a valid username

Request:

```
GET /email/by-username/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 /email/by-username/empty.user
```

Response:

```
200 OK
```

```
[]
```

Example with malformed username

Request:

```
GET /email/by-username/malf:or$med
```

Response:

```
400 BAD
```

```
[]
```

Example with an unknown user specified

Request:

```
GET /email/by-username/unknown.user
```

Response:

```
404 NOT FOUND
```

```
[]
```

`POST /email/by-username/{username}`

Associates an email address to a user

Parameters

Parameter	Optional?	Description
`username`		Username to associate an email to
`email`		Email to associate to user

Responses

- `200`: Request was successful
- `400`: Invalid username (malformed username)
- `404`: Unknown username
- `410`: Email address has already been associated to a user

Response is in the form of a JSON dictionary. The dictionary is empty on successful requests. It contains the error message in case of an unsuccessful request:

```json
{
  "error": "Error message"
}
```

Example with a valid username and email

Request:

```
POST /email/by-username/john.doe
[...]

email:john.doe@example.com
```

Response:

```
200 OK

{}
```

Example with an invalid username

Request:

```
POST /email/by-username/malf:or$med
[...]

email:john.doe@example.com
```

Response:

```
400 BAD

{
    "error": "Invalid username"
}
```

Example with an invalid email address

Request:

```
POST /email/by-username/john.doe
[...]

email:john.doe@
```

Response:

```
400 BAD

{
    "error": "Invalid email address"
}
```

Example with an unknown user

Request:

```
POST /email/by-username/unknown.user
[...]

email:john.doe@example.com
```

Response:

```
404 NOT FOUND

{
    "error": "Invalid username"
}
```

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

Request:

```
POST /email/by-username/john.doe
[...]

email:jane.doe@example.com
```

Response:

```
410 CONFLICT

{
    "error": "Email address has already been attached to another user"
}
```

DELETE /email/by-username/{username}

Detaches an email address to 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
email		Email to detach from user

Responses

- 200: Request was successful
- 400: Invalid username (malformed username)
- 404: Unknown username or email address is not attached to user

Response is in the form of a JSON dictionary. The dictionary is empty on successful requests. It contains the error message in case of an unsuccessful request:

```
{
  "error": "Error message"
}
```

Example with a valid username and email

Request:

"'DELETE /email/by-username/john.doe [...]

email:john.doe@example.com

```
**Response**:
```

200 OK

{}

```
### Example with an invalid username

**Request**:

`DELETE /email/by-username/malf:or$med
[...]

email:john.doe@example.com
`

**Response**:
```

400 BAD

{ "error": "Invalid username" }

```
### Example with an invalid email address

**Request**:

`DELETE /email/by-username/john.doe
[...]

email:john.doe@
`

**Response**:
```

400 BAD

{ “error”: “Invalid email address” }

```
### Example with an unknown user

**Request**:

`DELETE /email/by-username/unknown.user
[...]

email:john.doe@example.com
`

**Response**:
```

404 NOT FOUND

{ “error”: “Invalid username” }

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

**Request**:

`DELETE /email/by-username/john.doe
[...]

email:jane.doe@example.com
`

**Response**:
```

404 NOT FOUND

{ “error”: “Email is not attached to user” } “‘