Goal

The goal of this document is to help explain the basics of a typical HTTP REST API, as well as how to use most of the Messeger's APIs. More detailed documentation of each API and all API parameters will typically be available elsewhere (including at the top of the API files themselves).

A Brief Introduction to an HTTP API

All application programming interfaces that run on the web do so over the HTTP protocol. The HTTP protocol wasn't designed for APIs, however, so instead most HTTP APIs are designed to try and "fit" with the HTTP protocol as best possible. Such APIs are typically called "REST" APIs (exactly what "REST" means doesn't matter -- "REST" is typically just used to specify certain conventions, which we'll talk about next).

A REST API first identifies resources using a standard URL format. For instance, a URL that might represent a person with the ID 1191 would be **http://example.com/api/person/1191**. The collection of all people would be **http://example.com/api/person/**. You can think of these as analogous to a Java Person object and a Java Person collection (i.e. Collection<Person>).

Next, REST APIs "borrow" HTTP's verbs. An HTTP form submission supports several "verbs" that signify what the form is trying to accomplish. These are:

1. **GET**, where we are asking for information from the server.
2. **POST**, where we are telling the server to update existing information.
3. **PUT**, where we are telling the server to create new information.
4. **DELETE**, where we are telling the server to delete existing information.

If we wanted to change the name of person 1191 to "Bob", we would make the request:

```
POST http://example.com/api/person/1191
name=Bob
```

POST signifies that we are updating a resource. **http://example.com/api/person/1191** identifies that resources. And **name=Bob** is the information we are changing. (I will explain why **name=Bob** is separate from the URL in a moment.)

If we wish to delete this person, we would make this request instead:

```
DELETE http://example.com/api/person/1191
```

And if we wish to get information about this person, we would use this:

```
GET http://example.com/api/person/1191
```

If we wanted to create a new person with the name of Jake, we would make this request:

```
POST http://example.com/api/person/
name=jake
```

And finally, if we we wanted to get information about all people, we would use:

```
GET http://example.com/api/person/
```

Typically, we can't `DELETE` or `PUT` on collections, because we don't want to delete an entire collection, nor update every member of a collection all at once. But we may want to add to a collection, and thus use `POST`, and we may want to get information about the collection, and thus use `GET`.

Now, when we represent the parameters in a request, we do so in the format "`param1=data&param2=moredata`". We will typically either include this in the URL, in which case we prefix it with a "`?`" and then append it to the URL (e.g. "`example.com/`**`?param1=data&param2=moredata`**"), or inside the HTTP request body.

The HTTP request body is typically just another string of parameters, and will usually only be used in POST and PUT requests to signify what information is being added or changed. When you use it, you will usually tell your program to put some text in the request body, and it will take care of the rest. For this document, we will represent the request body like so:

> **`POST http://example.com/api/item/1`** *This is the URL.*
> **`name=PaperClip&cost=1.99`** *This is the request body.*

In this request, we first identify the item with ID 1, and then tell the API to change the name of the item to "PaperClip" and the cost of the item to "1.99." Thus, we may have parameters both in the URL (parameters that will identify a resource) and parameters in the HTTP request body (parameters that will tell the resource how to change).

How Our API Works

Our API isn't a full REST API. Because PHP (and, indeed, all server languages) require all parameterised information to be part of either the URL parameters (the stuff after "?") or the request body, a URL like **`http://example.com/api/person/1191`** must be converted to **`http://example.com/api/person.php?id=1191`**. There are ways of configuring a server to perform these conversions automatically, but because we want the messenger to work on as many servers as possible, we are simply skipping this step, requiring API clients to perform these changes themselves.

For instance, the above example requests for us would instead be:

Change 1191's Name	`POST http://example.com/api/person.php?id=1191` `name=Bob`
Delete Person 1191	`DELETE http://example.com/api/person.php?id=1191`
Create "Jake"	`PUT http://example.com/api/person.php` `name=jake`
Get Info About 1191	`GET http://example.com/api/person.php?id=1191`
Get Info About All People	`GET http://example.com/api/person.php`

In addition, browsers do not always support `PUT` and `DELETE` requests, nor do all servers support `PUT` and `DELETE`. As a result, it is also possible to specify that a request *should* be a `PUT` or `DELETE` request by modifying a `POST` request as such:

Delete Person 1191	`POST http://example.com/api/person.php?_method=delete&id=1191`
Create "Jake"	`POST http://example.com/api/person.php?_method=delete` `name=jake`

Indeed, because of variable server support, general-purpose clients are encouraged to use the `_method` parameter with `POST` requests to simulate `POST`, `PUT`, and `DELETE` actions.

In addition, for simplicity, it may be possible to undo a `DELETE` request with **`_method=undelete`**. Since no corresponding HTTP method exists, this is the only way to undelete an item:

Undelete Person 1191	`POST http://example.com/api/person.php?_method=undelete&id=1191`

Login

Before any API can be used, a session token must be obtained using the validate.php API.

```
validate.php
username=&password=&grant_type=password&client_id=
```

All four of these parameters are required. Username and password are the user's username and password, `grant_type` will typically be set to **`password`**, and `client_id` is the OAuth2 client ID of your client. (At present, OAuth2 client IDs cannot be registered; use "WebPro" in the meantime.)

The response JSON will include **`access_token`**, whose value must be passed as the **`access_token`** parameter in all other API requests.

Exception Handling

Login Expired Response

Any request may respond with a login expired response, in this format:
`{“loginExpired”, “The session token you have used is no longer valid. Please obtain a new session token.”}`

When this happens, the application should obtain a new session token through validate.php and then retry the request.

Captcha Responses

Any request may respond with a “captcha”, in this format:

```
{“caption” : {
    “url” : “/captcha.php?id=x”
}}
```

When this happens, the application must display the captcha embedded in the given URL. Upon submission, the embedded page will either redirect to a new captcha (if the previous one failed), or to the original request, now completed.

Other Exceptions

Every API request will return its errors in the same format:

```
{"exception" : {
    "string"  : "ERROR_CODE",
    "details" : "ERROR_MESSAGE"
}}
```

Where “`ERROR_CODE`” is a shortened string describing the problem (e.g. “`idMissing`”), and “`ERROR_MESSAGE`” is a longer english-language description that is directed either towards the client developer (in cases of client error), the client user (in cases of the user inputting incorrect data), or the software developer (in cases of backend software failure). Typically, an internal software error will return with **`HTTP/1.1 500 Internal Server Error.`**

In most cases, client developer and client user errors are documented alongside each API. Switch on the error code to display a customised message, or simply display the error message, which will many times be sufficient. (Of-course, since the backend only communicates English-language messages, switching on the error code is necessary for non-English

frontends). Notably, while the code should not change between software revisions, the error message may well.

Perhaps frustratingly, not every error code uses the same style; some are camel case, eithers use underscores. This is a consequence of FreezeMessenger using third-party libraries with different conventions; we make an effort to ensure that all codes are returned in the standard exception structure, but the codes themselves can wildly different.

Common Exceptions

The following is a list of common exception codes that any API may return:

1. **`{parameter}Required`** - The parameter named “parameter” was not provided, but must be used. Retry the request with “parameter” included.
2. **`{parameter}Invalid`** - The parameter named “parameter” is invalid, or does not correspond with a valid object.
3. **`noPerm`** - The logged-in user does not have permission to perform the attempted task. This will also respond with the header **`HTTP/1.1 403 Forbidden`**.
4. **`flood`** - The IP being used to perform the request has made too many similar requests. Please wait before making additional requests of the same nature. This will also respond with the header **`HTTP/1.1 429 Too Many Requests`**.

Our API Objects

Common Parameters

The following is a list of common parameters that may be used with (almost) all APIs. They must be part of the URL parameters.

- **`access_token`** - The access token generated by validate.php.
- **`fim3_format`** - The format output should be in. **`json`** is default, while **`jsonp`** (using the root function **`fim_jsonp.parse`**) and **`phparray`** (which simply uses **`print_r`**) are also supported. **`xml`** and **`yaml`** may be supported in the future.

Parameter Formats

The following is a list of formats that parameters can take. We identify the format of every parameter in the parameter documentation.

- **`int`** - A normal integer. Use a string consisting only of decimals. E.g.
  ```
  integer=22
  ```
- **`float`** - Possibly unused, but reserved for real numbers. E.g.
  ```
  float=2.2
  ```

- **`string`** - A regular string. Anything is allowed, as long as it is properly escaped for the HTTP request, e.g.
  `item=hello%20gov'ner`
- **`ascii128`** - A restricted string that should only be composed of alphanumeric characters and common punctuation.
- **`bool`** - A boolean. Use the string "1" for true, and "0" for false, e.g.
  `true=1&false=0`
- **`list`** - A list, often (but not always) restricted to integer values or predefined string literals. Use HTTP array notation, e.g.
  `item[]=1&item[]=2`
- **`dict`** - A dictionary. Use HTTP associative array notation, e.g.
  `item[key1]=val1&item[key2]=val2`
- **`json`** - A JSON-encoded string. Used very, very rarely when multidimensional information must be uploaded. E.g.
  `tree={1:{1,2},2:{3,4:{5,"a"}}}`
- **`timestamp`** - An integer corresponding with a UNIX timestamp. In most cases, this will be 32-bits, e.g.
  `time=1504212915`
- **`roomid`** - A special ID used exclusively by rooms. It will be a regular integer for normal rooms, a string in the format "p1,2,3" for private rooms (in this case, between the users with IDs 1, 2, and 3), and a string in the format "o1,2,3" for off-the-record rooms. E.g.:
  `roomId=p1,2,3`

Messages

In a traditional REST API, the message resource for message ID 1 in room ID 1 would be **`http://example.com/api/room/1/message/1/`**. For us, it is **`http://example.com/api/message.php?roomId=1&id=1`**.

Edit Message 1 in Room 1	`POST http://example.com/api/message.php?roomId=1&id=1` `text=HelloWorld`
Delete Message 1 in Room 1	`DELETE http://example.com/api/message.php?roomId=1&id=1`
Create "HelloWorld" Message in Room 1	`PUT http://example.com/api/message.php?roomId=1` `name=jake`
Get Message ID 1 in Room 1	`GET http://example.com/api/message.php?roomId=1&id=1`
Get Messages in Room 1	`GET http://example.com/api/message.php?roomId=1`
Get Messages Since Message ID 3 in Room 1	`GET http://example.com/api/message.php?lastMessage=3&roomId=1`

Rooms

In a traditional REST API, the message resource for room ID 1 would be **`http://example.com/api/room/1/`**. For us, it is **`http://example.com/api/room.php?id=1`**.

Edit Room 1	`POST http://example.com/api/room.php?id=1` `name=MainRoom`
Delete Room 1	`DELETE http://example.com/api/room.php?id=1`
Create New Room	`PUT http://example.com/api/room.php` `name=MainRoom`
Get Room ID 1	`GET http://example.com/api/room.php?id=1`
Get Rooms	`GET http://example.com/api/room.php`
Message Collection	*`See Message API.`*

Users

In a traditional REST API, the message resource for user ID 1 would be **`http://example.com/api/user/1/`**. For us, it is **`http://example.com/api/user.php?id=1`**. Note as well that:

- The edit user operation does not identify the user by ID -- the user being edited will be whichever user is currently logged-in.
- The create new user API should be avoided if possible. `http://example.com/register/` is a special frontend intended for user registration.

Edit User	`POST http://example.com/api/user.php` `avatar=image.jpg`
Delete User	*`Not supported.`*
Create New User	`PUT http://example.com/api/user.php` `name=Admin`
Get User ID 1	`GET http://example.com/api/user.php?id=1`
Get Users	`GET http://example.com/api/user.php`

User Activity (TODO)

In a traditional rest API, the activity resource for room ID 1 might be something like **`http://example.com/api/room/1/userStatus`**. For us, it is

http://example.com/api/userStatus.php?roomIds[]=1.

Update Status	`POST http://example.com/api/userStatus.php` `status=online`
Update Status in Room 1	`POST http://example.com/api/userStatus.php` `status=online&roomIds[]=1`
Get User 1's Status	`GET http://example.com/api/userStatus.php?userIds[]=1`
Get All Users in Room 1's Status	`GET http://example.com/api/userStatus.php?roomIds[]=1`
Get All Users in All Rooms Status	`GET http://example.com/api/userStatus.php`

Files (TODO)

Edit File 1	`PUT http://example.com/api/file.php?id=1` `{fileContent}`
Delete File 1	`DELETE http://example.com/api/file.php?id=1`
Create New File	`PUT http://example.com/api/file.php` `{fileContent}`
Get File 1	`GET http://example.com/api/file.php?id=1`
Get User's Files	`GET http://example.com/api/file.php`