# DA: Rest API (Falcon-Gunicorn-Nginx)

Applications for mobile devices - Theory - Unit 5

Didac Florensa Cazorla

**Any**: 2021-2022
**Curs**: 102386
**Institut**: University of Lleida (Campus Igualada)
**Titulació**: Bachelor’s degree in Digital Interaction and Computing Techniques (GTIDIC)

# Agenda

## Introduction

## Warm-up

- Dr. Jordi Mateo Fornés
- **Office**:
  - Office A.12 (Campus Igualada)
  - Office 3.08 (EPS Lleida)
- **Email**: jordi.mateo@udl.cat
- **Twitter**: https://twitter.com/MatForJordi
- **Github**: https://github/JordiMateoUdL

- Ph.D Dídac Florensa Cazorla
- **Office**:
  - Office A.12 (Campus Igualada)
  - Office 3.08 (EPS Lleida)
- **Email**: didac.florensa@udl.cat

## REST APIs

## Application Programming Interfaces (API)

- **APIs** allow software running on different systems to communicate.
- **APIs** allow programs to read, write or delete variables or information from another system.
- It is common to use standard **API** interfaces for interoperability and avoid reinventing the wheel.

### REMEMBER, remember the Fifth of November, gunpowder, treason and plot...

In the operating system course, we learn that the **API** provides an interface for applications to use hardware subsystems without knowing every detail of the hardware.

# REpresentational State Transfer (REST) API Attributes

**REST** is an architectural style for providing standards between computer systems on the web, making it easier for systems to communicate with each other. These systems are characterized by being stateless and separating the concerns of client and server.

1. **Uniform Interface**: Decouple principel. You must define a contract between the client and the service using standards.
2. **Client-Server**: Enforce separation of concerns. Client and server evolve independently.
3. **Stateless**: each request from the client to the server must contain all of the information necessary to understand and complete the request. So, The server cannot take advantage of any previously stored context information.
4. **Cacheable**: The client application gets the right to reuse the response data later for equivalent requests and a specified period.
5. **Layered System**: The client is agnostic as to how many layers, if any, there are between the client and the actual server responding to the request (secure layer, cache layer, load balancer,...)
6. **Code-on-demand**: REST also allows client functionality to extend by downloading and executing code in the form of applets or scripts.

## Communication between Client and Server

In the **REST** architectures, it is required that a client requests the server to retrieve or modify data on the server.

### HTTP Request

HTTP stands for Hypertext Transfer Protocol and structures requests and responses over the internet. HTTP requires data to be transferred over the network from one point to another.

A request generally consists of:

- An HTTP verb, which defines what kind of operation to perform. (GET,PUT,POST,DELETE).
- A header allows the client to pass along information about the request.
- A path to a resource is an optional message body containing data.

# REST Client Requests: CRUD

1. **CREATE**: Create a data structure or a variable and set this initial value.
2. **READ**: Query server for a variable.
3. **UPDATE**: Update content of a variable on the server.
4. **DELETE**: Delete the content of a variable on the server.

```
"book": {
  "id": <Integer>,
  "title": <String>,
  "author": <String>,
  "isbn": <Integer>
}
```

## CREATE

The client will supply (title, author and ISBN), and the server will create an instance of the book and add it to the library.

## READ

The client wants to see a list of all the books in the library. Also, we would have a function to filter book by author or retrieve a single book.

## UPDATE

The client wants to supply new values (title, author and ISBN).

## REMOVE

The client will supply the value to identify a book and delete it from the library.

## Combining REST and HTTP requests

- GET — retrieve a specific resource (by id) or a collection of resources
- POST — create a new resource
- PUT — update a specific resource (by id)
- DELETE — remove a specific resource by id

| CRUD | REST |
|---|---|
| CREATE | POST |
| READ | GET |
| UPDATE | PUT,PATCH |
| DELETE | DELETE |

## Combining REST and HTTP responses

| Status | Meaning |
|---|---|
| 200 (OK) | This is the standard response for successful HTTP requests. |
| 201 (CREATED) | This is the standard response for an HTTP request that resulted in an item being successfully created. |
| 204 (NO CONTENT) | This is the standard response for successful HTTP requests, where nothing is being returned in the response body. |
| 400 (BAD REQUEST) | The request cannot be processed because of wrong request syntax, excessive size, or another client error. |
| 403 (FORBIDDEN) | The client does not have permission to access this resource. |
| 404 (NOT FOUND) | The resource could not be found at this time. It is possible it was deleted or does not exist yet. |
| 500 (INTERNAL SERVER ERROR) | The generic answer for an unexpected failure if there is no more specific information available. |

## Example: Creating a book

```
POST http://damcore/book/
Body:
{
  "book": {
      "title": "One Piece Volume 1",
      "author": "Eiichiro Oda",
      "isbn": 0782009136637
  }
}
```

```
Status Code - 201 (CREATED)
Body -
{
  "book": {
      "id":1,
      "title": "One Piece Volume 1",
      "author": "Eiichiro Oda",
      "isbn": 0782009136637
  }
}
```

## Design TIPs

- What routes would you need to implement to provide your workout class model with this CRUD functionality, and what are their corresponding HTTP verbs?
- What effect would each route have on the database?
- What response body would each route return?
- What response code would each route return?

## Falcon

## Introduction

**Definition** minimalist Python web API framework for building reliable app backends and microservices.

- Highly-optimized, extensible codebase
- Intuitive routing via URI templates and REST-inspired resource classes
- Easy access to headers and bodies through request and response classes
- DRY request processing via middleware components and hooks
- Idiomatic HTTP error responses
- Straightforward exception handling
- Snappy unit testing through WSGI helpers and mocks
- CPython 2.6-2.7 and 3.4+, or PyPy 2.7 and 3.5+
- Cython support for an extra speed boost under CPython

The Falcon web framework encourages the **REST** architectural style. Resource classes implement HTTP method handlers that resolve requests and perform state transitions.

## Why I choose it?

- We have been using Falcon as a replacement for [framework], and we simply love the performance (three times faster) and codebase size (easily half of our original [framework] code).
- Falcon is rock solid, and it's fast. Falcon looks excellent so far. I hacked together a quick test for a tiny server of mine and was ~40% faster with only 20 minutes of work.
- I'm loving #falconframework! Super clean and simple, I finally have the speed and flexibility I need!
- I feel like I'm just talking HTTP at last, with nothing in the middle. Falcon seems like the request of the backend.
- The source code for Falcon is so good I almost prefer it to documentation. It basically can't be wrong.

## On summary

I chose and loved falcon because it is simple, extendable, solid, flexible and minimalist. Moreover, it is python-based ^^.

## REST architectural style

```
class QuoteResource:
    def on_get(self, req, resp):
        """Handles GET requests"""
        quote = {
            'quote': (
                "I've always been more interested in "
                "the future than in the past."
            ),
            'author': 'Grace Hopper'
        }
        resp.media = quote
api = falcon.API()
api.add_route('/quote', QuoteResource())
```

## Middleware and Hooks

### Hooks

Falcon supports before and after **hooks**. You install a hook simply by applying one of the decorators below, either to an individual responder or an entire resource.

### Middleware

**Middleware** components provide a way to execute logic before the framework routes each request after each request is routed but before the target responder is called or just before the response is returned for each request. Components are registered with the middleware kwarg when instantiating Falcon's API class.

### Hooks vs Middleware

Unlike hooks, middleware methods apply globally to the entire API.

# Decorators in python

**Decorators** allow you to inject or modify code in functions or classes.

```python
class my_decorator(object):

    def __init__(self, f):
        print("inside my_decorator.__init__()")
        f()

    def __call__(self):
        print("inside my_decorator.__call__()")

@my_decorator
def my_function():
    print("inside my_function()")

print("Finished decorating my_function()")
my_function()
```

inside my_decorator.__init__()

*inside my_function()*

Finished decorating my_function()

*inside my_decorator.__call__()*

```python
class entry_exit(object):

    def __init__(self, f):
        self.f = f

    def __call__(self):
        print("Entering", self.f.__name__)
        self.f()
        print("Exited", self.f.__name__)

@entry_exit
def func1():
    print("inside func1()")

@entry_exit
def func2():
    print("inside func2()")

func1()
func2()
```

*Entering func1 inside func1() Exited func1*

*Entering func2 inside func2() Exited func2*

## Hosting and deploying

Gunicorn **Green Unicorn** is a Python WSGI HTTP Server for UNIX. It's a pre-fork worker model ported from Ruby's Unicorn project. The Gunicorn server is broadly compatible with various web frameworks, simply implemented, light on server resource usage, and fairly speedy.

```
$ source .venv/bin/activate
$ pip install gunicorn
# Reload  option reloads the app whenever its code changes
$ gunicorn --reload myapi.app --port=8000
$ curl -v localhost:8000
```

## DamCore

## Init the API and the middlewares

```
# FALCON
app = application = falcon.API(
    middleware=[
        middlewares.DBSessionManager(),
        middlewares.Falconi18n(),
        MultipartMiddleware()
    ]
)
```

```
class DBSessionManager(object):
    def __init__(self):
        self.db_factory = db.DB_SCOPED_SESSION_FACTORY

    def process_resource(self, req, resp, resource, params):
        resource.db_session = db.create_db_session()

    def process_response(self, req, resp, resource, req_succeeded):
        if (resource is not None) and (resource.db_session is not None):
            resource.db_session.close()
```

## Creating Resources

Falcon uses Python classes to represent resources. In practice, these classes act as controllers in your application. They convert an incoming request into one or more internal actions and then compose a response back to the client based on the results of those actions.

```
class Resource:
    def on_get(self, req, resp):
        doc = {
            'images': [
                {
                    'href': '/images/1eaf6ef1-7f2d-4ecc-a8d5-6e8adba7cc0e.png'
                }
            ]
        }
        # Create a JSON representation of the resource
        resp.text = json.dumps(doc, ensure_ascii=False)
        resp.status = falcon.HTTP_200
```

Campus Universitari Igualada - UdL

Universitat de Lleida

## Creating a User (Example):

```
class ResourceRegisterUser(DAMCoreResource):
    def on_post(self, req, resp, *args, **kwargs):
        super(ResourceRegisterUser, self).on_post(req, resp, *args, **kwargs)

        aux_user = User()
        try:
            try:
                aux_genere = GenereEnum(req.media["genere"].upper())
            except ValueError:
                raise falcon.HTTPBadRequest(description=messages.genere_invalid)

            aux_user.username = req.media["username"]
            aux_user.password = req.media["password"]
            aux_user.email = req.media["email"]
            aux_user.name = req.media["name"]
            aux_user.surname = req.media["surname"]
            aux_user.genere = aux_genere

            self.db_session.add(aux_user)

            try:
                self.db_session.commit()
            except IntegrityError:
                raise falcon.HTTPBadRequest(description=messages.user_exists)

        except KeyError:
            raise falcon.HTTPBadRequest(description=messages.parameters_invalid)
        resp.status = falcon.HTTP_200
```

## DamCore

```python
# LOGGING
mylogger = logging.getLogger(__name__)
configure_logging()
# DEFAULT 404
# noinspection PyUnusedLocal
def handle_404(req, resp):
    resp.media = messages.resource_not_found
    resp.status = falcon.HTTP_404
# FALCON
app = application = falcon.API(
    middleware=[
        middlewares.DBSessionManager(),
        middlewares.Falconi18n(),
        MultipartMiddleware()
    ]
)
application.add_route("/", common_resources.ResourceHome())
# ACCOUNT
application.add_route("/account/profile", account_resources.ResourceAccountUserProfile())
application.add_route("/account/profile/update_profile_image", account_resources.ResourceAccountUpdateProfileImage())
application.add_route("/account/create_token", account_resources.ResourceCreateUserToken())
application.add_route("/account/delete_token", account_resources.ResourceDeleteUserToken())
# USERS
application.add_route("/users/register", user_resources.ResourceRegisterUser())
application.add_route("/users/show/{username}", user_resources.ResourceGetUserProfile())
# EVENTS
application.add_route("/events", event_resources.ResourceGetEvents())
application.add_route("/events/show/{id:int}", event_resources.ResourceGetEvent())
# OTHERS
application.add_sink(handle_404, "")
```

## URI Param and Query Param

What is the difference between:

1. GET `/events/show/{id:int}`

Unique Resource Identifier (URI) is a resource identifier passed as a parameter in the Uniform Resource Locator (URL). Unique Resource Identifier (URI) identifies a unique instance of a particular resource type. The URI parameter should be a unique identifier.

2. GET `/events?status="O"`

Query parameters are used to filter the collection resources type. The query parameters are passed at the end of the URL after a question mark to sort, filter or paginate the resource.

### Take Home Message

- Use query parameters or query string to sort, filter and paginate resources.
- Use the URI parameter to get a unique resource type.

## Using URI parameters (Events)

```
class ResourceGetEvent(DAMCoreResource):
# GET /events/show/{id:int}
    def on_get(self, req, resp, *args, **kwargs):
        super(ResourceGetEvent, self).on_get(req, resp, *args, **kwargs)

        if "id" in kwargs:
            try:
                response_event = self.db_session.query(Event)
                .filter(Event.id == kwargs["id"]).one()
                resp.media = response_event.json_model
                resp.status = falcon.HTTP_200
            except NoResultFound:
                raise falcon.HTTPBadRequest(description=messages.event_doesnt_exist)
        else:
            raise falcon.HTTPMissingParam("id")
```

## Using Query Param (Events)

```
class ResourceGetEvents(DAMCoreResource):
# GET /events?status=?
    def on_get(self, req, resp, *args, **kwargs):
        super(ResourceGetEvents, self).on_get(req, resp, *args, **kwargs)

        request_event_status = req.get_param("status", False)
        if request_event_status is not None:
            request_event_status = request_event_status.upper()
            if (len(request_event_status) != 1) or (
                    request_event_status not in [i.value for i in EventStatusEnum.__members__.values()]):
                raise falcon.HTTPInvalidParam(messages.event_status_invalid, "status")

        response_events = list()

        aux_events = self.db_session.query(Event)

        if request_event_status is not None:
            aux_events = \
                aux_events.filter(
                    Event.status == EventStatusEnum(request_event_status))

        if aux_events is not None:
            for current_event in aux_events.all():
                response_events.append(current_event.json_model)

        resp.media = response_events
        resp.status = falcon.HTTP_200
```

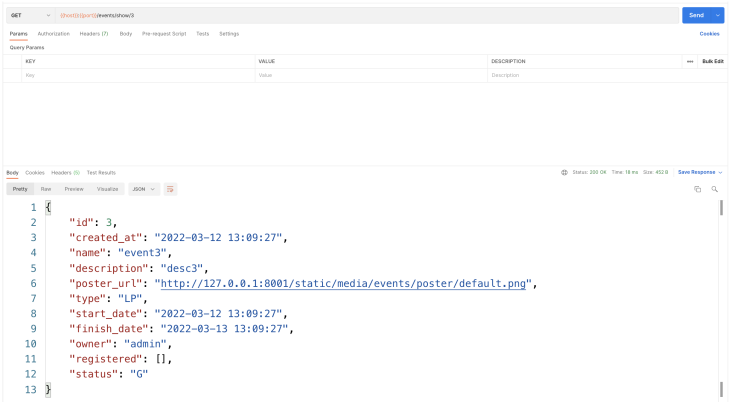
## Demo

## Demo

- Launch the services using '`docker-compose`.
- Test the API using PostMan or a terminal curl.

```
curl --location --request GET '127.0.0.1:8000/events?status=0'
curl --location --request GET '127.0.0.1:8000/events'
curl --location --request GET '127.0.0.1:8000/events/show/1'
```

# Get event by id

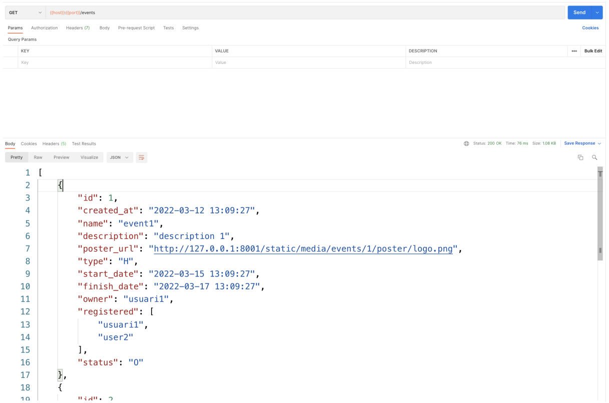


```
{{host}}:{{port}}/events/show/3
```

```json
{
    "id": 3,
    "created_at": "2022-03-12 13:09:27",
    "name": "event3",
    "description": "desc3",
    "poster_url": "http://127.0.0.1:8001/static/media/events/poster/default.png",
    "type": "LP",
    "start_date": "2022-03-12 13:09:27",
    "finish_date": "2022-03-13 13:09:27",
    "owner": "admin",
    "registered": [],
    "status": "G"
}
```

Figure 1: GET 127.0.0.1:8000/events/show/3

# Get Events



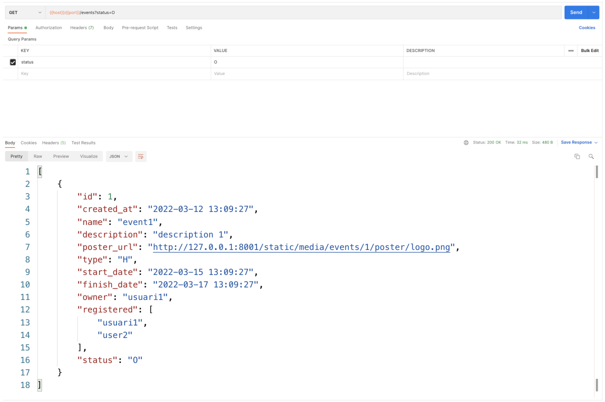
GET {{host}}:{{port}}/events

Status: 200 OK Time: 76 ms Size: 1.08 KB

```
[
    {
        "id": 1,
        "created_at": "2022-03-12 13:09:27",
        "name": "event1",
        "description": "description 1",
        "poster_url": "http://127.0.0.1:8001/static/media/events/1/poster/logo.png",
        "type": "H",
        "start_date": "2022-03-15 13:09:27",
        "finish_date": "2022-03-17 13:09:27",
        "owner": "usuari1",
        "registered": [
            "usuari1",
            "user2"
        ],
        "status": "0"
    },
    {
        "id": 2
```

Figure 2: GET 127.0.0.1:8000/events

# Filtering Events



{{host}}:{{port}}/events?status=O

Query Params

| KEY | VALUE | DESCRIPTION |
|---|---|---|
| status | O | |

Status: 200 OK Time: 32 ms Size: 480 B

```
[
    {
        "id": 1,
        "created_at": "2022-03-12 13:09:27",
        "name": "event1",
        "description": "description 1",
        "poster_url": "http://127.0.0.1:8001/static/media/events/1/poster/logo.png",
        "type": "H",
        "start_date": "2022-03-15 13:09:27",
        "finish_date": "2022-03-17 13:09:27",
        "owner": "usuari1",
        "registered": [
            "usuari1",
            "user2"
        ],
        "status": "O"
    }
]
```

Figure 3: GET 127.0.0.1:8000/events?status=O

## Serving Static Contents

## Implementation

- Build a container **aimed** to serve *static contents*.
- We are going to use **NGINX**. More information NGINX
- We will use this server to exchange (upload/store and retrieve) pictures.

### Services

- Backend using Gunicorn to deploy Falcon API, port 8000.
- NGINX to serve static content, port 8001.
- MySqlServer, port 3306.

## Dockerfile

```
# docker/docker-compose.yml
static-file-server:
    image: nginx:latest
    ports:
      - 8001:80
    volumes:
      - ../../static/.:/usr/share/nginx/html/static
```

⇒ You will need to create the *static* folder at the correct level. In this folder, **../../static/.** , we will create our static resources structure to serve the requests. We are copying this folder into the default Nginx static folder in the container **/usr/share/nginx/html/static**.

⇒ We redirect the *container* port **80** (nginx default) to *localhost* **8001**.

## Rules

- We are going to create a rule to store our static content (images) using this pattern:

```
${host}${static_folder}${media_folder}${resource-url}${image_filename}
```

- Example: We want the picture of the user

```
http://localhost:8001/static/media/users/2/photo/image.png
```

## Pattern Implementation: URL generation (1)

```
def _generate_media_url(class_instance,
     class_attibute_name, default_image=False):
        class_base_url = urljoin(
          urljoin(urljoin("http://{}"
            .format(
                settings.STATIC_HOSTNAME),
                settings.STATIC_URL),
                settings.MEDIA_PREFIX),
                class_instance.__tablename__ + "/"
            )
        class_attribute = getattr(class_instance,
            class_attibute_name)
```

## Pattern Implementation: URL generation (2)

```
if class_attribute is not None:
  return urljoin(urljoin(urljoin(urljoin(
    class_base_urlclass_attribute),
    str(class_instance.id) + "/"),
    class_attibute_name + "/"),
    class_attribute)
    else:
        if default_image:
          return urljoin(urljoin(class_base_url,
            class_attibute_name + "/"),settings.DEFAULT_IMAGE_NAME)
        else:
          return class_attribute
```

# Pattern Implementation: PATH generation

```python
def _generate_media_path(class_instance, class_attibute_name):
    class_path = "/{0}{1}{2}/{3}/{4}/".format(
      settings.STATIC_URL,
      settings.MEDIA_PREFIX,
      class_instance.__tablename__,
      str(class_instance.id),
      class_attibute_name)
    return class_path
```

## Update User model (models.py)

```python
class User(SQLAlchemyBase, JSONModel):
    __tablename__ = "users"
    photo = Column(Unicode(255))

    @hybrid_property
    def photo_url(self):
        return _generate_media_url(self, "photo")

    @hybrid_property
    def photo_path(self):
        return _generate_media_path(self, "photo")
```

## App

```python
# GET
application.add_route("/account/profile",
account_resources.ResourceAccountUserProfile())

#POST
application.add_route("/account/profile/update_profile_image",
account_resources.ResourceAccountUpdateProfileImage())
```

## GET

Just ensure that in json_model in models.py:

```
@hybrid_property
    def json_model(self):
        return {
          ...
          "photo": self.photo_url
        }
```

## POST (1)

```python
# @falcon.before(requires_auth)
class ResourceAccountUpdateProfileImage(DAMCoreResource):
    def on_post(self, req, resp, *args, **kwargs):
        super(ResourceAccountUpdateProfileImage, self)
          .on_post(req, resp, *args, **kwargs)
        # Get the user from the token
        current_user = req.context["auth_user"]
        resource_path = current_user.photo_path
        # Get the file from form
        incoming_file = req.get_param("image_file")
```

## POST (2)

```
# Run the common part for storing
filename = utils
.save_static_media_file(incoming_file, resource_path)
# Update db model
current_user.photo = filename
self.db_session.add(current_user)
self.db_session.commit()
```

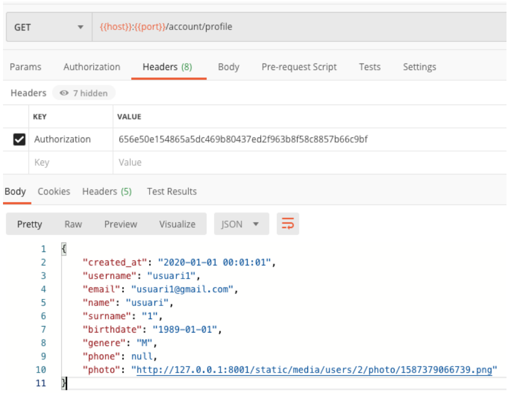
## save_static_media_file (utils.py) (1)

```python
def save_static_media_file(incoming_file, resource_path):
    # Generate and id with the currrent timestamp
    imgId = str(int(datetime.datetime.now().timestamp() * 1000))
    # Build filename using the id generated
    filename = imgId + "." + incoming_file.filename.split(".")[-1]
    # Check if folder exists or not in the server
    if not os.path.exists(resource_path):
        os.makedirs(resource_path)
```

## save_static_media_file (utils.py) (2)

```
# Create a file path
file_path = resource_path + filename
# Write to a temporary file to prevent
# incomplete files from being used
temp_file_path = file_path + "~"
with open(temp_file_path, "wb") as f:
    f.write(incoming_file.file.read())
# File has been fully saved to disk move it into place
os.rename(temp_file_path, file_path)
return filename
```

## Testing (1)

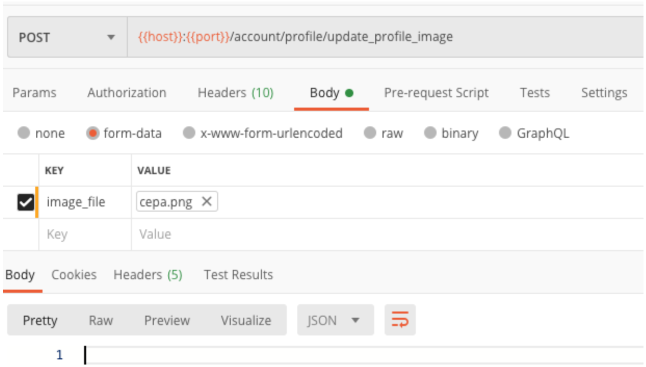


| KEY | VALUE |
|---|---|
| Authorization | 656e50e154865a5dc469b80437ed2f963b8f58c8857b66c9bf |
| Key | Value |

```
{
    "created_at": "2020-01-01 00:01:01",
    "username": "usuari1",
    "email": "usuari1@gmail.com",
    "name": "usuari",
    "surname": "1",
    "birthdate": "1989-01-01",
    "genere": "M",
    "phone": null,
    "photo": "http://127.0.0.1:8001/static/media/users/2/photo/1587379066739.png"
}
```

Figure 4: Screenshot from Postman, operation getAccountProfile()

## Testing (2)


POST {{host}}:{{port}}/account/profile/update_profile_image

| KEY | VALUE |
|---|---|
| ✓ image_file | cepa.png |
| Key | Value |

Figure 5: Screenshot from Postman, operation updateAccountProfile()

## Testing(3)

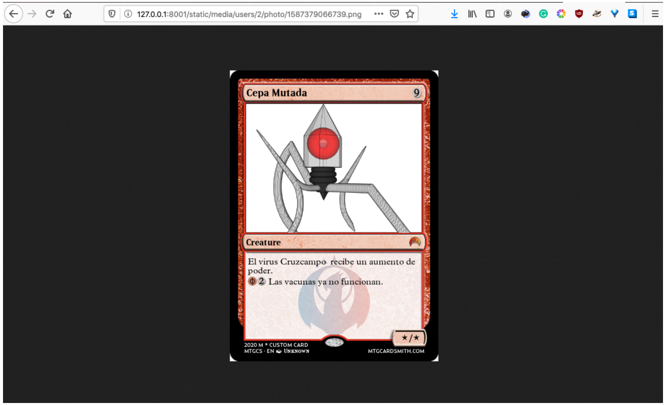

127.0.0.1:8001/static/media/users/2/photo/1587379066739.png
Cepa Mutada
9
Creature
El virus Cruzcampo recibe un aumento de poder.
2 Las vacunas ya no funcionan.
★/★
2020 M * CUSTOM CARD
MTGCS · EN UNKNOWN
MTGCARDSMITH.COM

Figure 6: Screenshot from browser, getting the uploaded image

## Homework

## Task A: Books

- Create book model.
- Create book CRUD.

⇒ Tutorial Book - Jordi Mateo **Tutorial. Book. Part 1**

- Design a hook from the book.
- Add a book image. (Put and Retrieve some image).

## That’s all

QUESTIONS?

### About me

**www** — jordimateofornes.com
**github** — github.com/JordiMateo
**twitter** — @MatForJordi
**gdc** — Distributed computation group