# RESTful Web APIs

REpresentational State Transfer Web APIs: p*aradigm to implement Web APIs*

Guidelines:

- APIs → ***M2M communications***
- Manipulation of text resources represented by URIs based on ***HTTP methods***
- Uses HTTP response status codes to denote the outcome of the operation
- ***Stateless***

- Usually based on JSON data, but might support content negotiation
- Input variables might be included in the **Resource URI**

- Many times applications *do not strictly comply* to all guidelines

# RESTful Web APIs

Manipulation of text resources based on ***HTTP methods***

- CRUD operations mapped to HTTP methods
  - CREATE → POST
  - READ → GET
  - UPDATE → PUT/PATCH
  - DELETE → DELETE
  - https://restfulapi.net/http-methods/

- Use HTTP response status code to denote the outcome. *Examples:*
  - 200 OK → request handled successfully
  - 400 Bad Request → input parameters errors
  - 415 Unsupported media type → invalid requested response format

# Example: Input variables

*Handle manipulation of colors based on their RGB values*

- Example API solution:

  /api/v0.1/color/yellow

  - POST: create a new color given RGB values
  - GET: return RGB values of the color named yellow
  - PUT: updated the existing color named yellow with new RGB values

- *Typical common variants that are not strictly REST-compliant:*
  - *Use query parameters to specify the object*
    - *Example: /api/v0.1/colors?name=yellow*
  - *Support both create and update operations by using a POST*

# Example: Input data

**/api/v0.1/color/yellow**

- **POST**: create a new color given RGB values.

  *How to send data?*

  - send values in the payload as JSON data

    *'Content-type': 'application/json'*

    *{'r': 255, 'g': 255, 'b': 0}*

  - send values in the payload as Form data

    *'Content-type': 'application/ x-www-form-urlencoded'*

    *r=255&g=255&b=0*

  *Other input data formats might also be used (e.g., other forms input data the we already discussed)*

# Example: Error management

- Normally, wrong input parameters would generate a *400 response status code. Examples:*

*Request body*

*{'name': 'yellow', 'r': 255, 'g':300}*

*Response status code and body variants*

*400 Bad request*
*{'message': 'invalid request parameters'}*

REST-compliant, HTTP already shows the application outcome

*OR*

*200 Ok*
*{'status': 'ko', 'message': 'invalid request parameters'}*

Error only appears in the application data, not in the HTTP protocol (note: Facebook implements this)

# Example: Content negotiation

- Using the due request and response headers is very important
  - the request header "*Accept*" can specify the requested format
- The content type might also specified as in input in the URI
- Example:
  - /api/v0.1/json/color/<name> , /api/v0.1/xml/color/<name>

*OR*

  - */api/v0.1/color/<name>?json* , */api/v0.1/color/<name>?xml*

***Responses***

```
{
  'name': 'yellow',
  'r': 255,
  'g': 255,
  'b': 0
}
```

```
<xml>
  <name>'yellow'</name>
  <r>255</r>
  <g>255</g>
  <b>0</b>
</xml>
```

# Authentication in RESTful APIs

- ***M2M communication*** → there is no reason to use **passwords**
  - i.e., a password is a secret that ***easy to remember for humans***

- API authentication keys → ***secure random tokens***
  - *e.g., 16 random bytes*

- *Usually sent through headers*

- *May also use JWT to authenticate or encrypt JSON data, but usually rely on **https** for that*

# Example: Authentication

- See e.g. Mashape
  - https://market.mashape.com/
    - Maket for APIs offered through RESTful Web APIs
  - All APIs can be accessed by using an API key set in the *'X-Mashape-Key'* header

We will use Mashape services in the **lab exam**, so **(at home) create an account and try using these services**

# Exposing RESTful APIs with Flask

- Can implement RESTful APIs by using the plain Flask library
  - take care of all the due characteristics (e.g., CSRF, inputs, response and errors formats)

- Can use an additional dedicated library → ***flask_restful***

  - *We discuss how to expose REST APIs by using the library*
    - *We give examples and hints, for **all options** read the **documentation** **https://flask-restful.readthedocs.io/en/latest/***

      - Create API handlers
      - Handle inputs

# Using flask restful [2]

- Create the restful app wrapper
  - exempt api handlers from the CSRF token
    - NOTE: only if you comply to the stateless constraint, i.e., do not use any session

```
…
from flask_restful import Api
…
csrf_protect = CSRFProtect(app)
api = Api(app, decorators=[csrf_protect.exempt])
```

# Using flask restful [3]

- Create handlers
  - return objects are Python structures that are automatically **converted to JSON data** (Note: all flask methods, e.g. abort, are wrapped to support this behavior)

```
from flask_restful import Resource
class Color(Resource):
    def get(self, name):
        return {}
    def post(self, name):
        return {}
    def put(self, name):
        return {}
api.add_resource(Color, '/api/v0.1/color/<string:name>')
```

# Using flask restful [4]

- Handle inputs with an *argparse-like* parser
  - approach similar to flask_wtf ORM for forms
  - can easily verify parameters ***location (e.g., URI, payload, query)***, *enums*, *optional*, *types*, *and other options*

Can set where the parser look for parameters in the request object (e.g., args, headers, body, json, ...)

http://flask-restful.readthedocs.io/en/0.3.5/reqparse.html#argument-locations

```
from flask_restful import reqparse
parser = reqparse.RequestParser()
parser.add_argument('r', type=int, required=True, location='json')

…

class Color(Resource):
    def get(self, name):
        args = parser.parse_args()
        …
        return { … }

    …
api.add_resource(Color, '/api/v0.1/color/<string:name>')
```

Can define custom callable as for argparse data parsing

# Exercise

- Implement the Color application and API described before
  - We use the Google Datastore to maintain information

*/api/v0.1/colors/<name>*

- GET: return RGB values of the color <name>
- POST: create a new color given RGB values from JSON data
  {'r': <red>, 'b': <blue>, 'g': <green>}
- PUT: updated the existing color named yellow with new RGB values (same JSON data accepted by the POST request)

# Swagger and OpenAPI

*"Swagger is the world's largest framework of API developer tools for the OpenAPI Specification (OAS), enabling development across the entire API lifecycle, from design and documentation, to test and deployment."*

*https://swagger.io/*

- Defines a language to formally describe Web APIs
  - **v2.0 → Swagger** format

*Moved to the Open API Initiative (OAI) consortium in Jan. 1st 2016, the Swagger Specification has been **renamed** to OpenAPI Specification*

  - **V3.0 → OpenAPI** format https://openapis.org
    - still called Swagger format most of the times

# Swagger/OpenAPI specification file

- Define an API specification file (Open API specification - OAS)
  - v2.0 → *swagger.yaml*
  - v3.0 → *openapi.yaml*

  usually placed in the Web service root, that describes the black-box behavior of the Web service

- The standard describes how to write the

  https://swagger.io/specification/ (current v3.0.1)

# swagger.yaml specification file

```
swagger: <swagger-version>
info:
  version: <api-version>
  title: <service-name>
paths:
  <resource-path>:
    <method>:
      description: <method-description>
      parameters:
       - name: <field-name>
         in: <input-type>
         type: <value-type>
         required: {true|false}
         description: <parameter-description>
      responses:
        <http-status-code>:
          description: <response-description>
          schema: <returned-data-schema>
```

*see http://swagger.io/specification*

# swagger.yaml specification file “hello world” example

```yaml
swagger: "2.0"
info:
  version: "1.0"
  title: "Hello World API"
paths:
  /hello/{user}:
    get:
      description: Returns a greeting to the user!
      parameters:
        - name: user
          in: path
          type: string
          required: true
          description: The name of the user to greet.
      responses:
        200:
          description: Returns the greeting.
          schema:
            type: string
        400:
          description: Invalid characters in "user" were provided.
```

# swagger.yaml specification file request parameters position

The field `in` defines where the server will get the parameter.

Available values are ***query***, ***header***, ***path***, ***formData***, ***body***

- ***query*:** from the url query (e.g. localhost/?field=value)
- ***header*:** from the http request headers
- ***path*:** from the URI (the field name must be marked in the URI string - see the hello world example)
- ***formData*:** from the http body, when data are passed as key-values. Use the ***consumes*** option to define the accepted data MIMEs.
- ***body*:** from the http body, when data are passed with custom structures (e.g., JSON). Use the ***schema*** option to define the data structure.

*See http://swagger.io/specification/#parameterObject*

# swagger.yaml specification file complex data types

The field **`type`** defines the type of data accepted by the server. It is used in many parameters options, such as requests' parameters and responses.

The **type** field defines the primive data types (e.g., string, int, number). An additional **format** option can be given to detail the actual nature of the data (e.g., int32, int64, float, datetime, password). See “Data Types” at http://swagger.io/specification/.

Complex data types (e.g., custom JSON data) can be defined by using the special type **object**, followed by an additional option **properties** that defines the structure of the data.

Complex structures can be defined either *inline* or by custom data **definitions** referenced by using the option **$ref**

# swagger.yaml specification file
# complex data types - example (inline)

```
...
      responses:
        200:
          description: ...
          schema:
            type: object
            properties:
              status:
                type: string
                format: string
              status_code:
                type: integer
                format: int32
              message:
                type: string
                format: string
              detail:
                type: string
          format: string
```

```
{
    'status': <status-string>,
    'status_code': <status-code>,
    'message': <standard-message>
    'detail': <content>
}
```

# swagger.yaml specification file
# complex data types - example (definition)

```
...
      responses:
        200:
          description: ...
          schema:
            $ref: '#/definitions/MessageResponse'
definitions:
  MessageResponse:
    type: object
    properties:
      status:
        type: string
        format: string
      status_code:
        type: integer
        format: int32
      message:
        type: string
        format: string
      detail:
        type: string
        format: string
```

# Utilities examples

- Tools for editing specification files
  - e.g., Swagger Editor https://swagger.io/swagger-editor/
    - both online and offline version (e.g., for confidential documents)

- Tools for creating software based on the specification files
  - create stub and skeleton libraries (e.g., see client and server generation of the swagger editor utility)

- Tools for testing
  - e.g., Swagger UI https://swagger.io/swagger-ui/

# Exercise 1

- Write a RESTful Web API with flask restful that implements the Color application for storing and retrieving custom colors
  - Extra: Require that new colors can be created only by registered users that own a correct API key

- Write the swagger specification file for the Color REST API
  - Chose your design choices if some details were not specified

# Exercise 2

- Write a Web service for a Quiz
  - the application uses the *random fomus quotes Web service* to get data about famus quotes and their authors
    - https://market.mashape.com/andruxnet/random-famous-quotes
  - the application allows users to get a random quote, for which they must guess an author

- Note: take all the due design choices trying to comply the RESTful paradigm and produce your swagger specification
  - (I will propose my design choice at the end of the month)