GPUL <Labs/>

RESTful APIs

License

- This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit:
- http://creativecommons.org/licenses/by-sa/4.0/

That's me! :D

- Alejo Pacín Jul
- Software Engineer at Softtek
- GPUL member
- https://es.linkedin.com/in/alejopj

Index

- API
- REST
- RESTful APIs
- Exercises
- Python IDEs
- Git tools
- REST API tools
- Django REST Framework

<theory>

API

API

- Application Programming Interface
- Set of routines, protocols, and tools for building software and applications.
- An API expresses a software component in terms of its operations, inputs, outputs, and underlying types, defining functionalities that are independent of their respective implementations, which allows definitions and implementations to vary without compromising the interface.

API

- In other words:
  - Contract
  - Providing functionalities grouped in services
  - By specifying a set of available operations by their signatures (names, inputs and outputs)
  - To allow communication between systems

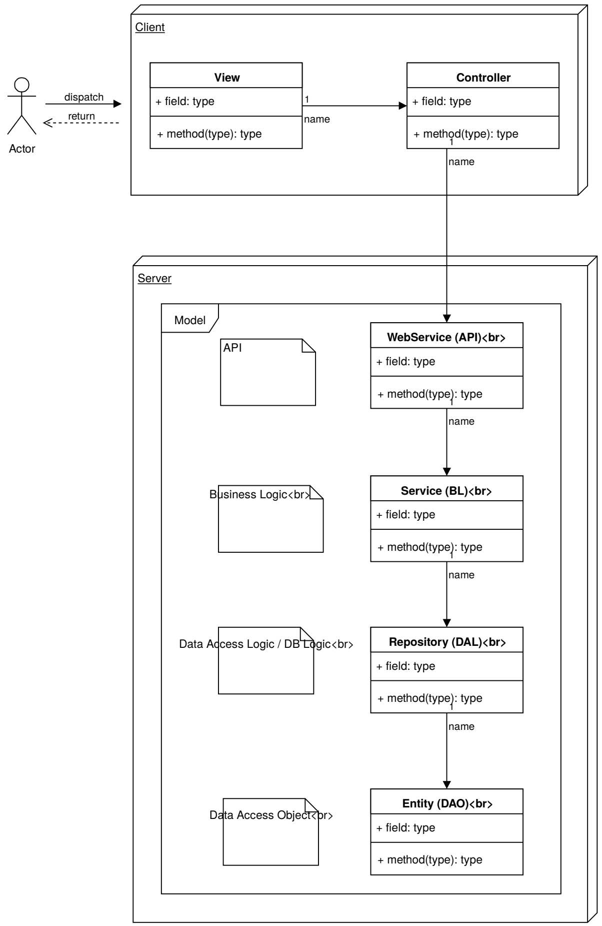
API

- Where to place an API?
  - Assuming we use a software architecture based on the following design patterns:
    - Client-Server
    - MVC
      - Note: it can also be applied to other patterns like MVP or MVVM, but as MVC is the most used one nowadays, we are going to use it as example.
  - Where the View and the Controller are placed on the Client's side.
  - We have 2 very-well-known options for our API architecture:
    - Standard
    - Simplified
      - Note: the designs below can be even simpler but we are focusing only on the API.

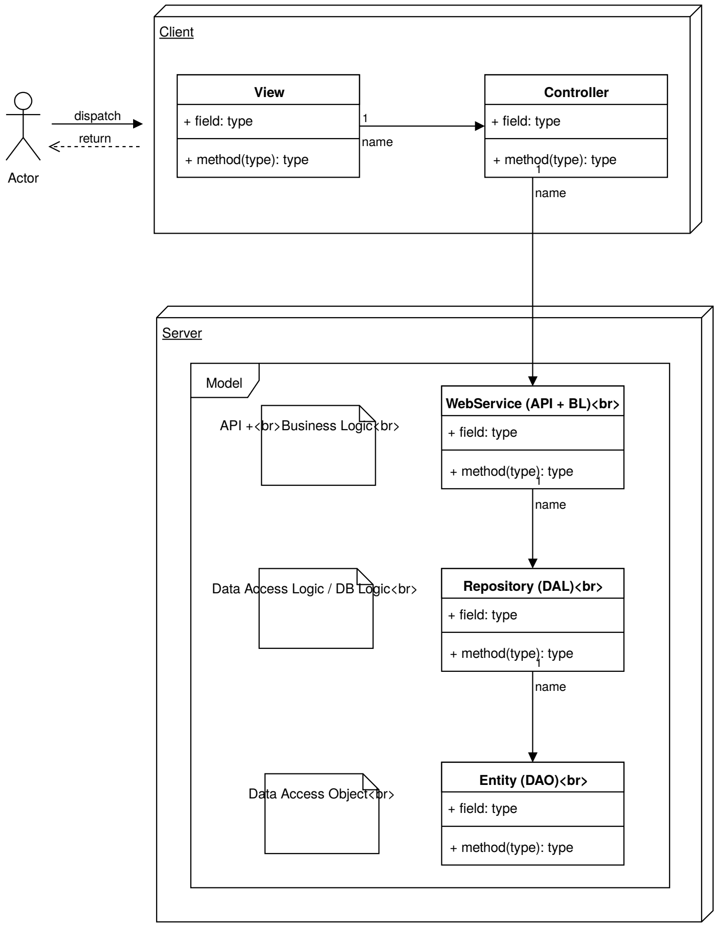
API

- Where to place an API?
  - Assuming we use a software architecture based on the following design patterns:
    - Client-Server
    - MVC
      - Note: it can also be applied to other patterns like MVP or MVVM, but as MVC is the most used one nowadays, we are going to use it as example.
  - Where the View and the Controller are placed on the Client's side.
  - We have 2 very-well-known options for our API architecture:
    - Standard
    - Simplified
      - Note: the designs below can be even more simple but we are focusing only on the API.

API


Client
Actor
dispatch
return
View
+ field: type
+ method(type): type
1
name
Controller
+ field: type
+ method(type): type
1
name
Server
Model
API
WebService (API)<br>
+ field: type
+ method(type): type
1
name
Business Logic<br>
Service (BL)<br>
+ field: type
+ method(type): type
1
name
Data Access Logic / DB Logic<br>
Repository (DAL)<br>
+ field: type
+ method(type): type
1
name
Data Access Object<br>
Entity (DAO)<br>
+ field: type
+ method(type): type

API


Client
Actor
dispatch
return
View
+ field: type
+ method(type): type
1
name
Controller
+ field: type
+ method(type): type
1
name
Server
Model
API +<br>Business Logic<br>
WebService (API + BL)<br>
+ field: type
+ method(type): type
1
name
Data Access Logic / DB Logic<br>
Repository (DAL)<br>
+ field: type
+ method(type): type
1
name
Data Access Object<br>
Entity (DAO)<br>
+ field: type
+ method(type): type

REST

REST

- Representational State Transfer
- Software architectural style of the World Wide Web
- Architectural style consisting of a coordinated set of architectural constraints applied to components, connectors, and data elements, within a distributed hypermedia system
- Ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements

REST

- Properties
  - Performance
  - Scalability
  - Simplicity
  - Modifiability
  - Visibility
  - Portability
  - Reliability

REST

- Constraints
  - Client-Server
  - Stateless
  - Cacheable
  - Layered system
  - Code on demand (optional)
  - Uniform interface

REST

- Uniform interface
  - Identification of resources
  - Manipulation of resources through these representations
  - Self-descriptive messages
  - Hypermedia as the engine of application state (HATEOAS)

REST

- Richardson Maturity Model
  - Way to grade your API according to the constraints of REST
  - The better your API adheres to these constraints, the higher its score is
  - 4 levels, 0-3, where level 3 designates a truly RESTful API

REST

- Level 0: Swamp of POX
  - Use its implementing protocol (normally HTTP, but it doesn't have to be) like a transport protocol.
  - Tunnel requests and responses through its protocol without using the protocol to indicate application state.
  - It will use only one entry point (URI) and one kind of method (in HTTP, this normally is the POST method).
  - Examples of these are SOAP and XML-RPC.

REST

- Level 1: Resources
  - Can distinguish between different resources.
  - This level uses multiple URIs, where every URI is the entry point to a specific resource.
  - Instead of going through http://example.org/articles, you actually distinguish between http://example.org/article/1 and http://example.org/article/2
  - Still, this level uses only one single method like POST.

REST

- Level 2: HTTP verbs
  - Your API should use the HTTP***** protocol properties in order to deal with scalability and failures
  - Don't use a single POST method for all, but make use of GET when you are requesting resources, and use the DELETE method when you want to delete a resources
  - Also, use the response codes of your application protocol. Don't use 200 (OK) code when something went wrong for instance
  - *****REST is completely protocol agnostic, so if you want to use a different protocol, your API can still be RESTful

REST

- Level 3: Hypermedia controls
  - Uses HATEOAS to deal with discovering the possibilities of your API towards the clients

REST

- REST vs RESTful
  - REST: Level 2
  - RESTful: Level 3

REST

- REST/ful API Description Languages
  - WSDL: Level 0-2
  - WADL: Level 2
  - HATEOAS: Level 3

REST

- HATEOAS
  - Hypermedia as the Engine of Application State
  - Hypertext-driven APIs
  - The client software is not written to a static interface description shared through documentation
  - Instead, the client is given a set of entry points and the API is discovered dynamically through interaction with these endpoints

RESTful APIs

RESTful APIs

- Web service APIs that adhere to the REST architectural constraints are called RESTful APIs
- HTTP-based RESTful APIs are defined with the following aspects:
  - Base URI
    - Such as http://example.com/resources/
  - An Internet media type for the data.
    - This is often JSON but can be any other valid Internet media type (e.g., XML, Atom, microformats, application/vnd.collection+json, etc.)
  - Standard HTTP methods
    - E.g., OPTIONS, GET, PUT, POST, or DELETE
  - Hypertext links to reference state
  - Hypertext links to reference-related resources

RESTful APIs

- How to start building your own RESTful API?
  - It should start right after you domain name
    - Domain: http://www.example.org
    - Following the pattern below:
      - /api/{version}/{access-level}*
        - {version}: API version
        - {access-level}: API access level
        - *Both version and access level are optional
        - **Versioning your API is highly recommended**
        - Access level is not so important and most people don't use it, but it is useful for quick access filtering and pre-securitization

RESTful APIs

- What is the correct way to version my API?
  - The URL way:
    - A commonly used way to version your API is to add a version number in the URL. For instance:
      - /api/**v1**/article/1234
    - To "move" to another API, one could increase the version number:
      - /api/**v2**/article/1234
    - The Hypermedia way:
      - GET /api/article/1234 HTTP/1.1
      - Accept: application/vnd.api.article+xml; **version=1.0**

RESTful APIs

- How to specify the access level?
  - The most common way is to specify two different values:
    - /priv vs /pub
    - /private vs /public
  - And then match those values to you application user roles
  - At this level API resource entry level, it is not worth it to be more granular by adding more values

RESTful APIs

- How do I let users log into my RESTful API?
  - One of the main differences between RESTful and other server-client communications services is that **any session state** in a RESTful setup **is held in the client**, **the server is stateless**
  - This requires the client to provide all information necessary to make the request
  - Different ways:
    - HTTP Basic Authentication
    - HMAC
    - OAuth/OAuth2
    - JWT
  - Notes:
    - Most of them **MUST** be protected through **SSL/TLS**. They should not be used over plain HTTP
    - Using nonces can improve your security, but you MUST store and compare nonces server-side

RESTful APIs

- Tips on how to build a RESTful API
  - Use HTTP Verbs to Make Your Requests Mean Something
  - Provide Sensible Resource Names
  - Use HTTP Response Codes to Indicate Status
  - Offer Both JSON and XML*
    - *Mobile Apps -> JSON, please
  - Create Fine-Grained Resources
  - Consider Connectedness (HATEOAS)

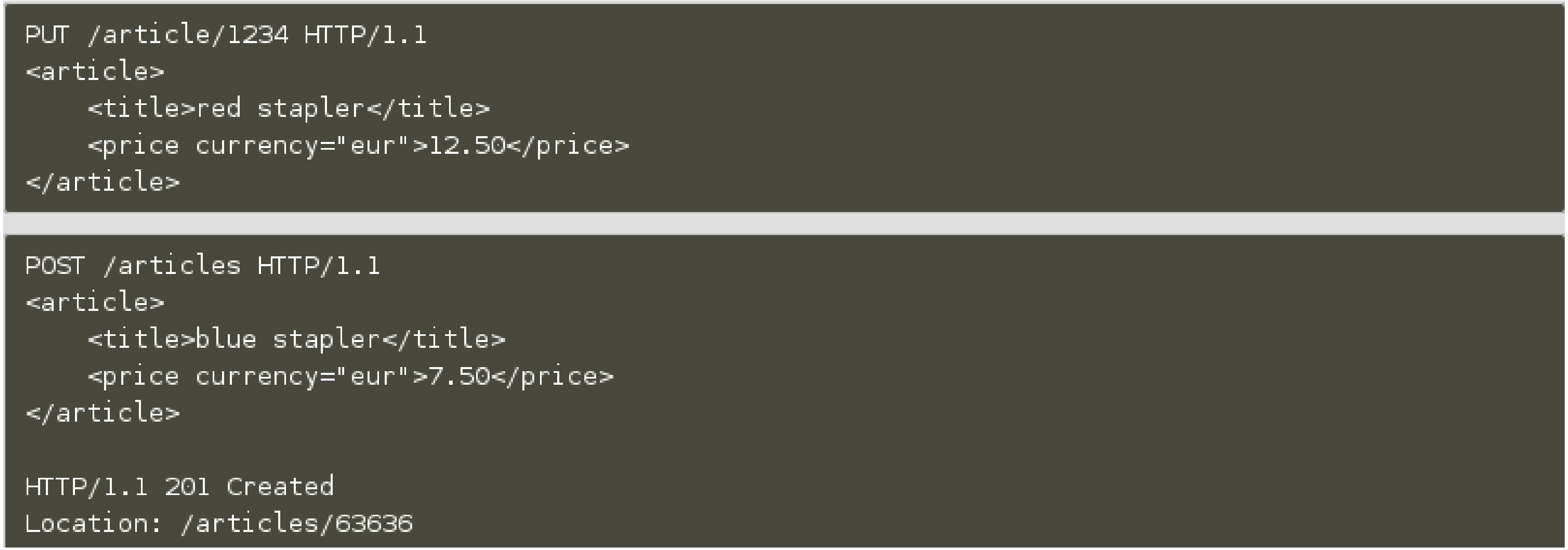
RESTful APIs

- Use HTTP Verbs to Make Your Requests Mean Something
  - The **HTTP verbs** comprise a major portion of our “uniform interface” constraint and **provide** us **the action** counterpart **to the noun-based resource**
  - The primary or most-commonly-used HTTP verbs:
    - GET (Read)
    - POST (Create)
    - PUT (Update)
    - DELETE (Delete)
  - There are a number of other verbs too, but are utilized less frequently:
    - PATCH (Partial update)
    - OPTIONS (Supported methods)
    - HEAD

RESTful APIs

- When to use PUT or POST?
  - Use PUT when you can update a resource completely through a specific resource
  - For instance, if you know that an article resides at http://example.org/article/1234, you can PUT a new resource representation of this article directly through a PUT on this URL
  - PUT and POST are both unsafe methods. However, PUT is idempotent, while POST is not
  - PUTting the same data multiple times to the same resource, should not result in different resources, while POSTing to the same resource can result in the creation of multiple resources

RESTful APIs



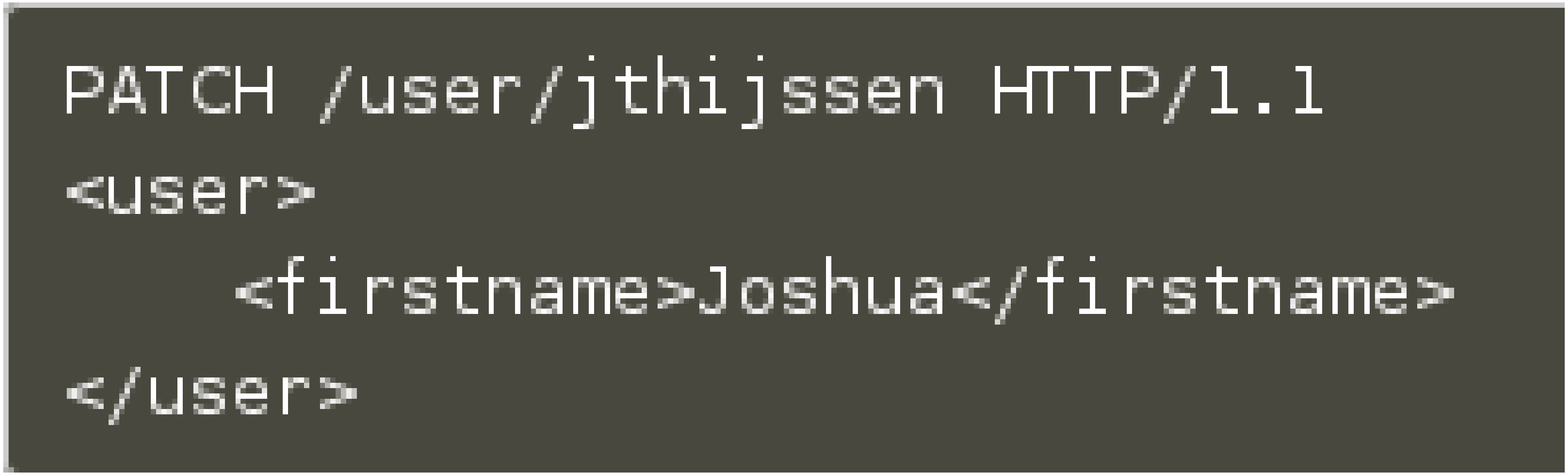
```
PUT /article/1234 HTTP/1.1
<article>
    <title>red stapler</title>
    <price currency="eur">12.50</price>
</article>
```

```
POST /articles HTTP/1.1
<article>
    <title>blue stapler</title>
    <price currency="eur">7.50</price>
</article>

HTTP/1.1 201 Created
Location: /articles/63636
```

RESTful APIs

- When to use PATCH?
  - The HTTP methods PATCH can be used to update partial resources
  - For instance, when you only need to update one field of the resource, PUTting a complete resource representation might be cumbersome and utilizes more bandwidth



```
PATCH /user/jthijssen HTTP/1.1
<user>
    <firstname>Joshua</firstname>
</user>
```

RESTful APIs

- What are idempotent and/or safe methods?
  - Safe methods are HTTP methods that do not modify resources
    - For instance, using GET or HEAD on a resource URL, should NEVER change the resource
    - However, this is not completely true
    - It means: it won't change the resource representation
    - It is still possible, that safe methods do change things on a server or resource, but this should not reflect in a different representation

RESTful APIs

- What are idempotent and/or safe methods?
  - Be careful when dealing with safe methods:
    - If a seemingly safe method like GET will change a resource, it might be possible that any middleware client proxy systems between you and the server, will cache this response
    - Another client who wants to change this resource through the same URL (like: http://example.org/api/article/1234/delete), will not call the server, but return the information directly from the cache
    - Non-safe (and non-idempotent) methods will never be cached by any middleware proxies

RESTful APIs

- What are idempotent and/or safe methods?
  - An idempotent HTTP method is a HTTP method that can be called many times without different outcomes
  - It would not matter if the method is called only once, or ten times over. The result should be the same
  - Again, this only applies to the result, not the resource itself. This still can be manipulated

RESTful APIs

- Idempotency is important in building a fault-tolerant API
  - Suppose a client wants to update a resource through POST
  - Since POST is not a idempotent method, calling it multiple times can result in wrong updates
  - What would happen if you sent out the POST request to the server, but you get a timeout. Is the resource actually updated?
  - Does the timeout happened during sending the request to the server, or the response to the client?
  - Can we safely retry again, or do we need to figure out first what has happened with the resource?
  - By using idempotent methods, we do not have to answer this question, but we can safely resend the request until we actually get a response back from the server.

RESTful APIs

HTTP Method	**Idempotent**	**Safe**
OPTIONS	Yes	Yes
GET	Yes	Yes
HEAD	Yes	Yes
PUT	Yes	No
POST	No	No
DELETE	Yes	No
PATCH	No	No

RESTful APIs

- Caching your REST API
  - The goal of caching is never having to generate the same response twice
  - The benefit of doing this is that we gain speed and reduce server load
  - The best way to cache your API is to put a gateway cache (or reverse proxy) in front of it
  - Some frameworks provide their own reverse proxies, but a very powerful, open-source one is **Varnish**

RESTful APIs

- Caching your REST API
  - When a safe method is used on a resource URL, the reverse proxy should cache the response that is returned from your API
  - It will then use this cached response to answers all subsequent requests for the same resource before they hit your API
  - When an unsafe method is used on a resource URL, the cache ignores it and passes it to the API
  - The API is responsible for making sure that the cached resource is invalidated

RESTful APIs

- Caching your REST API
  - HTTP has an unofficial PURGE method that is used for purging caches
  - When an API receives a call with an unsafe method on a resource, it should fire a PURGE request on that resource so that the reverse proxy knows that the cached resource should be expired
  - Note that you will still have to configure your reverse proxy to actually remove a resource when it receives a request with the PURGE method

RESTful APIs

- Provide Sensible Resource Names
  - Producing a great API is 80% art and 20% science
  - Creating a URL hierarchy representing sensible resources is the art part
  - Having sensible resource names (which are just URL paths, such as /customers/12345/orders) improves the clarity of what a given request does
  - Appropriate resource names provide context for a service request, increasing understandability of the API
  - Resources are viewed hierarchically via their URI names, offering consumers a friendly, easily-understood hierarchy of resources to leverage in their applications

RESTful APIs

- Provide Sensible Resource Names
  - Here are some quick-hit rules for URL path (resource name) design:
    - Use identifiers in your URLs instead of in the query-string. Using URL query-string parameters is fantastic for filtering, but not for resource names
      - Good: /users/12345
      - Poor: /api?type=user&id=23
    - Leverage the hierarchical nature of the URL to imply structure.
    - **Design for your clients, not for your data**
    - **Resource names should be nouns. Avoid verbs as resource names**, to improve clarity. **Use the HTTP methods to specify the verb** portion of the request
    - **Use plurals in URL segments to keep your API URIs consistent across all HTTP methods**, using the collection metaphor.
      - Recommended: /customers/33245/orders/8769/lineitems/1
      - Not: /customer/33245/order/8769/lineitem/1
    - Avoid using collection verbiage in URLs. For example 'customer_list' as a resource. Use pluralization to indicate the collection metaphor (e.g. customers vs. customer_list)
    - Use lower-case in URL segments, separating words with underscores ('_') or hyphens ('-'). Some servers ignore case so it's best to be clear
    - Keep URLs as short as possible, with as few segments as makes sense

RESTful APIs

- Use HTTP Response Codes to Indicate Status
  - Response status codes are part of the HTTP specification
  - There are quite a number of them to address the most common situations
  - In the spirit of having our RESTful services embrace the HTTP specification, our Web APIs should return relevant HTTP status codes
  - For example, when a resource is successfully created (e.g. from a POST request), the API should return HTTP status code 201.

RESTful APIs

- Use HTTP Response Codes to Indicate Status
  - Suggested usages for the "Top 10" HTTP Response Status Codes are as follows:
    - **200 OK**
      - General success status code. This is the most common code. Used to indicate success
    - **201 CREATED**
      - Successful creation occurred (via either POST or PUT). Set the Location header to contain a link to the newly-created resource (on POST). Response body content may or may not be present
    - **204 NO CONTENT**
      - Indicates success but nothing is in the response body, often used for DELETE and PUT operations
    - **400 BAD REQUEST**
      - General error for when fulfilling the request would cause an invalid state. Domain validation errors, missing data, etc. are some examples
    - **401 UNAUTHORIZED**
      - Error code response for missing or invalid authentication token
    - **403 FORBIDDEN**
      - Error code for when the user is not authorized to perform the operation or the resource is unavailable for some reason (e.g. time constraints, etc.)
    - **404 NOT FOUND**
      - Used when the requested resource is not found, whether it doesn't exist or if there was a 401 or 403 that, for security reasons, the service wants to mask
    - **405 METHOD NOT ALLOWED**
      - Used to indicate that the requested URL exists, but the requested HTTP method is not applicable
      - For example, POST /users/12345 where the API doesn't support creation of resources this way (with a provided ID). The Allow HTTP header must be set when returning a 405 to indicate the HTTP methods that are supported. In the previous case, the header would look like "Allow: GET, PUT, DELETE"
    - **409 CONFLICT**
      - Whenever a resource conflict would be caused by fulfilling the request. Duplicate entries, such as trying to create two customers with the same information, and deleting root objects when cascade-delete is not supported are a couple of examples
    - **500 INTERNAL SERVER ERROR**
      - Never return this intentionally. The general catch-all error when the server-side throws an exception. Use this only for errors that the consumer cannot address from their end

RESTful APIs

- When to return 4xx or 5xx codes to the client?
  - 4xx codes are used to tell the client that a fault has taken place on THEIR side
    - They should not retransmit the same request again, but fix the error first
  - 5xx codes tell the client something happened on the server and their request by itself was perfectly valid
    - The client can continue and try again with the request without modification

RESTful APIs

HTTP Verb	**CRUD**	**Entire Collection** (e.g. /customers)	**Specific Item** (e.g. /customers/{id})
POST	Create	201 (Created), 'Location' header with link to /customers/{id} containing new ID.	404 (Not Found), 409 (Conflict) if resource already exists..
GET	Read	200 (OK), list of customers. Use pagination, sorting and filtering to navigate big lists.	200 (OK), single customer. 404 (Not Found), if ID not found or invalid.
PUT	Update / Replace	404 (Not Found), unless you want to update/replace every resource in the entire collection.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
PATCH	Update / Modify	404 (Not Found), unless you want to modify the collection itself.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
DELETE	Delete	404 (Not Found), unless you want to delete the whole collection—not often desirable.	200 (OK). 404 (Not Found), if ID not found or invalid.

RESTful APIs

- Are REST and HTTP the same thing?
  - Nope!
  - HTTP stands for HyperText Transfer Protocol and is a way to transfer files. This protocol is used to link pages of hypertext in what we call the World Wide Web. However, there are other transfer protocols available, like FTP
  - REST, or REpresentational State Transfer, is a set of constraints that ensure a scalable, fault-tolerant and easily extendible system. The world-wide-web is an example of such system

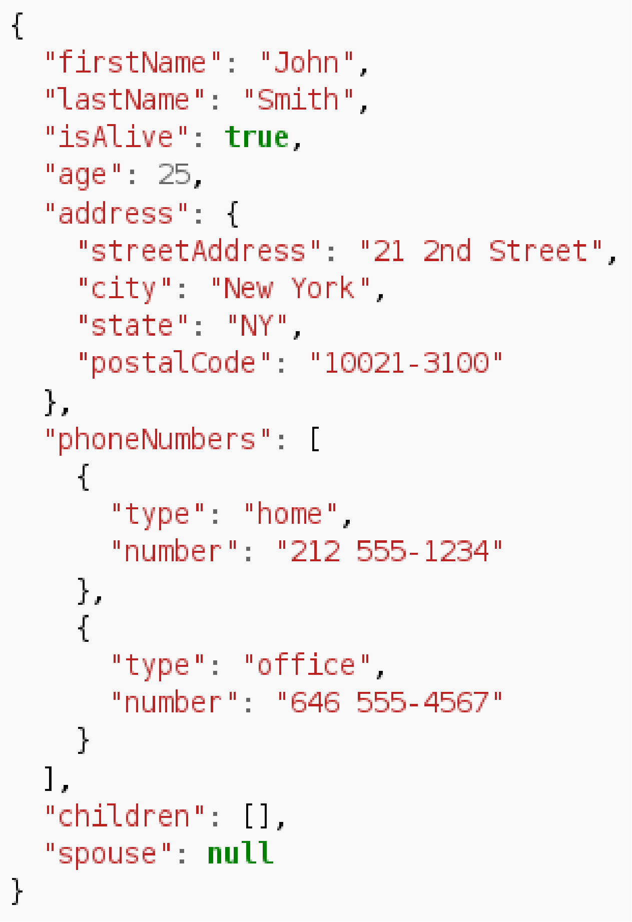
RESTful APIs

- Offer Both JSON and XML
  - Favor JSON support unless you're in a highly-standardized and regulated industry that requires XML, schema validation and namespaces, and offer both JSON and XML unless the costs are staggering
  - Ideally, let consumers switch between formats using the HTTP Accept header, or by just changing an extension from .xml to .json on the URL

RESTful APIs

- JSON
  - JavaScript Object Notation
  - The official Internet media type for JSON is “application/json”
  - Data types:
    - Number
    - String
    - Boolean
    - Array
    - Object
    - null

RESTful APIs



```json
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

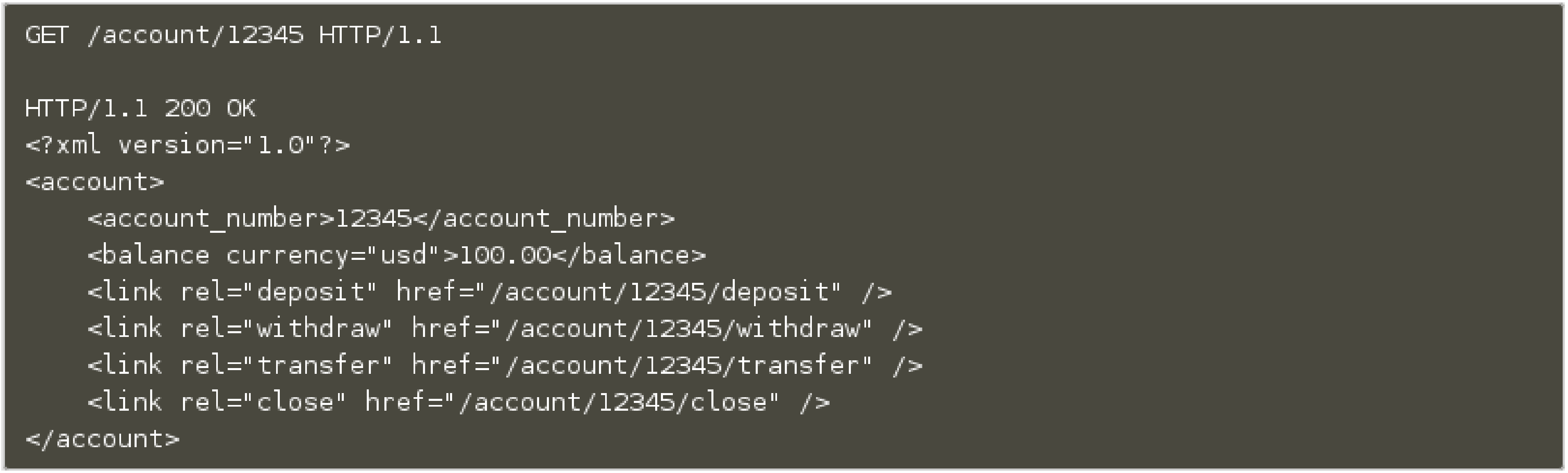
RESTful APIs

- Consider Connectedness
  - One of the principles of REST is connectedness—via hypermedia links (HATEOAS)
  - While services are still useful without them, APIs become more self-descriptive and discoverable when links are returned in the response
  - At the very least, a 'self' link reference informs clients how the data was or can be retrieved
  - Additionally, utilize the HTTP Location header to contain a link on resource creation via POST (or PUT)
  - For collections returned in a response that support pagination, 'first', 'last', 'next' and 'prev' links at a minimum are very helpful

RESTful APIs

- What is HATEOAS and why is it important for my REST API?
  - HATEOAS stands for Hypertext As The Engine Of Application State
  - It means that hypertext should be used to find your way through the API
  - Let's see an example

RESTful APIs



```
GET /account/12345 HTTP/1.1

HTTP/1.1 200 OK
<?xml version="1.0"?>
<account>
    <account_number>12345</account_number>
    <balance currency="usd">100.00</balance>
    <link rel="deposit" href="/account/12345/deposit" />
    <link rel="withdraw" href="/account/12345/withdraw" />
    <link rel="transfer" href="/account/12345/transfer" />
    <link rel="close" href="/account/12345/close" />
</account>
```

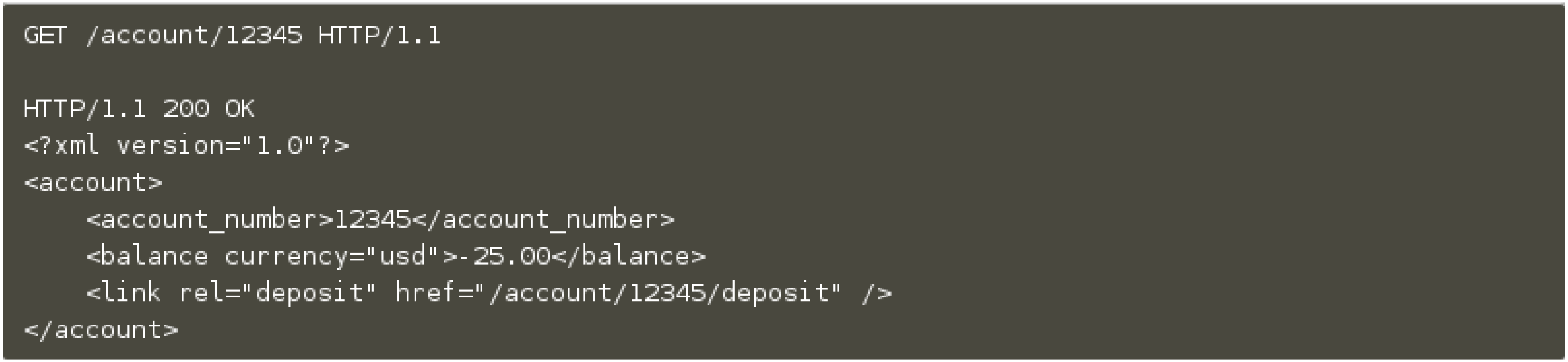
RESTful APIs

- Apart from the fact that we have 100$ in our account, we can see 4 options: deposit more money, withdraw money, transfer money to another account, or close our account.
- The "link"-tags allows us to find out the URLs that are needed for the specified actions

RESTful APIs

- Now, let's suppose we didn't have 100$ in the bank, but we actually are in the red

RESTful APIs



```
GET /account/12345 HTTP/1.1

HTTP/1.1 200 OK
<?xml version="1.0"?>
<account>
    <account_number>12345</account_number>
    <balance currency="usd">-25.00</balance>
    <link rel="deposit" href="/account/12345/deposit" />
</account>
```

RESTful APIs

- Now we are 25$ in the red
- Do you see that right now we have lost many of our options, and only depositing money is valid?
- As long as we are in the red, we cannot close our account, nor transfer or withdraw any money from the account
- The hypertext is actually telling us what is allowed and what not: HATEOAS

RESTful APIs

- Is my API RESTful when I use (only) JSON?
  - Nope!
  - One of the key constraints on REST is that a RESTful API must use hypermedia formats (HATEOAS)
  - JSON is not a hypermedia format
  - Although JSON does't have inherent hypermedia support, some standardisation is on its way to change that:
    - **JSON-LD**, which is already an official W3C standard
    - And **HAL**, which is a personal project
  - Both formalize the expression of links in JSON so that clients can instantly follow and discover these links without having to rely on out-of-band additional knowledge

</theory>

</br>

<practice>

Exercises

Exercises

- **Exercise 1**: provide an API that fits the requirements for the following user story
  - **Story/Feature**: user resets his/her password
    - **In order to** log in
    - **As a** user
    - **I want to** reset my password
- Note: BDD, Gherkin language

Exercises

- **Scenario 1**: providing a valid user's email while resetting the password should send a unique one-shot time-limited link by email to the user
  - **Given** the user is not logged in
  - **And** is on the login page
  - **And** clicks on the reset password link
  - **When** filling the email field with a valid email
  - **And** clicking on the reset password button
  - **Then** a reset password email should be sent to the provided email address

Exercises

- **Scenario 2**: clicking on the email's reset password link should display the reset password page
  - **Given** the user is not logged in
  - **When** clicking on the email's reset password link
  - **Then** the reset password page should be displayed

Exercises

- **Scenario 3**: resetting the password should get the user logged in
  - **Given** the user is not logged in
  - **And** is on the reset password page
  - **When** introducing a valid password twice
  - **And** both passwords match
  - **Then** the password is reset
  - **And** a confirmation email is sent to the user
  - **And** the user is automatically logged in

Exercises

- **Exercise 2**: the following ideas were proposed as possible applications to be implemented during the next GPUL IoT Hackatons:
  - Drowning prevention in swimming pools (surveillance system)
  - Home automation (Domotic)
  - Twitter/Identi.ca bot
- Write down a few user cases for each idea using BDD, including possible scenarios
- Create an API design for each idea, user case and scenario
- Deliver them all to the GPUL <Labs/> team, so they can be shared with all the participants:
  - https://labs.gpul.org/#contact

Exercises

- **Exercise 2**: the following ideas were proposed as possible applications to be implemented during the next GPUL IoT Hackatons:
  - Drowning prevention in swimming pools (surveillance system)
  - Home automation (Domotic)
  - Twitter/Identi.ca bot
- Write down a few user cases for each idea using BDD, including possible scenarios
- Create an API design for each idea, user case and scenario
- Deliver them all to the GPUL <Labs/> team, so they can be shared with all the participants:
  - https://labs.gpul.org/#contact

Python IDEs

Python IDEs

- Ninja IDE
- PyCharm
- **PyDev**

Python IDEs

- PyDev
  - Python IDE for Eclipse, which may be used in Python, Jython and IronPython development
    - Django integration
    - Code completion
    - Code completion with auto import
    - Type hinting
    - Code analysis
    - Go to definition
    - Refactoring
    - Debugger
    - Remote debugger
    - Find Referrers in Debugger
    - Tokens browser
    - Interactive console
    - Unittest integration
    - Code coverage
    - Find References (Ctrl+Shift+G)

Python IDEs

- PyDev
  - Download “Eclipse IDE for Java EE Developers” zipped version if possible
  - It is highly recommended to setup a different workspace for this instance if you already have another one installed
  - Follow the PyDev installation manual paying attention to the details, please
  - Add an alias to your .bashrc config:
    - alias pydev=/eclipse-installation-path/eclipse

Git tools

Git tools

- GitEye
  - Intuitive graphical Git client with integration to your favorite planning, tracking, code reviewing, and build tools
  - Support for Windows, GNU/Linux and Mac OSX

REST API tools

REST API tools

- Curl
- **SoapUI**
- Firefox + plugins
  - REST
    - **REST Easy**
    - RESTED
    - RESTClient
  - JSON
    - **JSON-handle**
    - JSONView
- Chrome + plugins
  - Postman

REST API tools

- Curl
  - Is a tool to transfer data from or to a server, using one of the supported protocols (DICT, FILE, FTP, FTPS, GOPHER, HTTP, HTTPS, IMAP, IMAPS, LDAP, LDAPS, POP3, POP3S, RTMP, RTSP, SCP, SFTP, SMTP, SMTPS, TELNET and TFTP)
  - The command is designed to work without user interaction

REST API tools

- Curl
  - # man curl
  - # curl http://maps.googleapis.com/maps/api/geocode/json?address=%22a%20coru%C3%B1a%22&sensor=false
  - # curl http://www.openstreetmap.org/api/0.6/relation/346810

REST API tools

- SoapUI
  - Is a free and open source cross-platform Functional Testing solution
  - With an easy-to-use graphical interface, and enterprise-class features, SoapUI allows you to easily and rapidly create and execute automated functional, regression, compliance, and load tests
  - In a single test environment, SoapUI provides complete test coverage and supports all the standard protocols and technologies

REST API tools

- SoapUI
  - New REST project

REST API tools

- Firefox
  - REST Easy
  - JSON-handle

Django REST Framework

Django REST Framewok

- Django
  - If you have not attended the previous GPUL <Labs/> about Django, please, refer to the related repository at:
    - https://github.com/gpul-labs/roadmap#semana4

Django REST Framewok

- Is a powerful and flexible toolkit for building Web APIs
  - The Web browsable API is a huge usability win for your developers
  - Authentication policies including packages for OAuth1a and OAuth2
  - Serialization that supports both ORM and non-ORM data sources
  - Customizable all the way down - just use regular function-based views if you don't need the more powerful features
  - Extensive documentation, and great community support

Django REST Framewok

- Requeriments
  - Python
  - Django
- Optional (recommended)
  - markdown - Support for the browsable API.
  - django-filter - Filtering support
  - django-crispy-forms - Improved HTML display for filtering
  - django-guardian - Object level permissions support

Django REST Framewok

- Installation
  - #apt-get install python3
  - #pip3 install django
  - #pip3 install **djangorestframework**
  - #pip3 install markdown
  - #pip3 install django-filter
  - #pip3 install django-crispy-forms
  - #pip3 install django-guardian

Django REST Framewok

- PyDev
  - File → New → PyDev Django Project
  - Grammar Version → 3.0
  - Interpreter
    - Click here to configure an interpreter not listed
      - New
        - Name → python3
        - Executable → /usr/bin/python3
  - Django version → 1.4 or later (1.9)
  - DB → SQLite3

Django REST Framewok

- settings.py
  - Add 'rest_framework' to your INSTALLED_APPS settings
    - INSTALLED_APPS = (
    -     ...
    -     'rest_framework',
    -)

Django REST Framewok

- urls.py
  - If you're intending to use the browsable API you'll probably also want to add REST framework's login and logout views
  - Add the following to your root urls.py file
    - urlpatterns = [
    -     ...
    -     url(r'^api-auth/', include('rest_framework.urls', namespace='rest_framework'))
    -]

Django REST Framewok

- Example
  - http://www.django-rest-framework.org/#example
  - DB
    - https://docs.djangoproject.com/en/1.9/intro/tutorial02/
    - Secondary button over example folder
    - Django → Migrate/Make Migrations
  - Run
    - http://www.pydev.org/manual_adv_django.html
    - Secondary button over example folder
    - Run as → PyDev:Django
    - http://127.0.0.1:8000/

Django REST Framewok

- Quickstart
  - http://www.django-rest-framework.org/tutorial/quickstart/

Django REST Framework

- Tutorial
  - Serialization
  - Requests & Responses
  - Class based views
  - **Authentication & permissions**
  - Relationships & **hyperlinked APIs**
  - **Viewsets & routers**

Django REST Framework

- API Guide
  - Requests
  - Responses
  - Views
  - Generic views
  - Viewsets
  - **Routers**
  - Parsers
  - Renderers
  - Serializers
  - Serializer fields
  - Serializer relations
  - Validators
  - **Authentication**
  - **Permissions**
  - **Throttling**
  - Filtering
  - Pagination
  - **Versioning**
  - Content negotiation
  - Metadata
  - Format suffixes
  - Returning URLs
  - Exceptions
  - Status codes
  - **Testing**
  - Settings

Django REST Framework

- Authentication
  - http://www.django-rest-framework.org/api-guide/authentication/
  - BasicAuthentication
  - TokenAuthentication
  - SessionAuthentication
  - Custom authentication
  - Third party packages
    - **Django OAuth Toolkit**
    - **JSON Web Token Authentication**

Django REST Framework

- Versioning
  - http://www.django-rest-framework.org/api-guide/versioning/
  - Roy Fielding: *“Versioning an interface is just a “polite” way to kill deployed clients”*.

Django REST Framework

- Testing
  - http://www.django-rest-framework.org/api-guide/testing/
  - Jacob Kaplan-Moss: *“Code without tests is broken as designed”*.

Django REST Framework

- Topics
  - **Documenting your API**
  - Internationalization
  - **AJAX, CSRF and CORS**
  - HTML & Forms
  - Browser enhancements
  - The Browsable API
  - **REST, Hypermedia & HATEOAS**

Django REST Framework

- REST, Hypermedia & HATEOAS
  - http://www.django-rest-framework.org/topics/rest-hypermedia-hateoas/
  - A **MUST** read
  - What REST framework doesn't do is give you machine readable hypermedia formats such as:
    - HAL,
    - Collection+JSON,
    - JSON API
    - or HTML microformats by default,
  - Or the ability to auto-magically create fully HATEOAS style APIs that include hypermedia-based form descriptions and semantically labelled hyperlinks
  - Doing so would involve making opinionated choices about API design that should really remain outside of the framework's scope

Django REST Framework

- REST, Hypermedia & HATEOAS
  - ProTip:
    - Django Ripozo
    - https://www.reddit.com/r/Python/comments/3bntj3/ripozo_a_tool_for_quickly_creating_hateoasrest/
    - https://github.com/vertical-knowledge/django-ripozo

Django REST Framework

- **Exercise 3**: based on the RasPi home automation project developed on the last GPUL <Labs/> about Django:
  - https://github.com/gpul-labs/roadmap#semana4
- Write down a few user cases using BDD, including possible scenarios
- Create an API design for each user case and scenario
- Deliver them all to the GPUL <Labs/> team, so they can be shared with all the participants:
  - https://labs.gpul.org/#contact
- Note: model evolutions are allowed

</practice>

</end>

<bibliography>

Bibliography

- API
  - Wikipedia
    - https://en.wikipedia.org/wiki/Application_programming_interface

Bibliography

- REST
  - How I Explained REST to Non-Tech People
    - http://web.archive.org/web/20130116005443/http://tomayko.com/writings/rest-to-my-wife
  - Wikipedia - REST
    - https://en.wikipedia.org/wiki/Representational_state_transfer
  - REST Cook Book
    - http://restcookbook.com/

Bibliography

- RESTful APIs
  - Wikipedia - REST
    - https://en.wikipedia.org/wiki/Representational_state_transfer
  - REST Cook Book
    - http://restcookbook.com/
  - REST API Tutorial
    - http://www.restapitutorial.com/
  - Wikipedia - RESTful API Description Languages
    - https://en.wikipedia.org/wiki/Overview_of_RESTful_API_Description_Languages
  - Wikipedia – JSON
    - https://en.wikipedia.org/wiki/JSON

Bibliography

- Exercises
  - BDD
    - https://en.wikipedia.org/wiki/Behavior-driven_development
  - Gherkin language
    - https://en.wikipedia.org/wiki/Cucumber_%28software%29
  - GPUL IoT Hackaton Ideas:
    - https://twitter.com/gpul_/status/705140161907138569
  - GPUL </Lab> about Django Framework;
  - https://github.com/gpul-labs/roadmap#semana4

Bibliography

- Python IDEs
  - Ninja IDE
    - http://ninja-ide.org/
  - PyCharm
    - https://www.jetbrains.com/pycharm/
  - PyDev
    - http://www.pydev.org/
    - https://www.eclipse.org/downloads/
    - http://www.pydev.org/manual_101_install.html

Bibliography

- Git tools
  - GitEye
    - http://www.collab.net/products/giteye

Bibliography

- REST API tools
  - Curl
  - SoapUI
    - https://www.soapui.org/
  - Firefox + plugins
    - REST
      - REST Easy
      - RESTED
      - RESTClient
    - JSON
      - JSON-handle
      - JSONView
  - Chrome + plugins
    - Postman
    - https://chrome.google.com/webstore/search/rest

Bibliography

- Django REST Framework
  - Django Project
    - https://www.djangoproject.com/
  - Django REST Framework
    - http://www.django-rest-framework.org/
  - HATEOAS
    - http://www.django-rest-framework.org/topics/rest-hypermedia-hateoas/
    - https://www.reddit.com/r/Python/comments/3bntj3/ripozo_a_tool_for_quickly_creating_hateoasrest/