Pratical RESTful API design

Exploration over technologies enabling a Model-First approach in RESTful API development

Giovanni Ciatto
giovanni.ciatto@studio.unibo.it

University of Bologna
Computer Science and Engineering

May 18, 2016

Table of Contents

Outline

Overview I

REST = **RE**presentational **S**tate **T**ransfert

- REST *is* basically a concept, a set of principles (best practices) referred as "constraints" and described through natural language
- REST *is not* a tight specification, and there exists more than a way to produce RESTful web services
  - Yeah, this is about web-services, so cool!
- It's not a dichotomy, nor simply a matter of RESTful and RESTless web-services:
  - there are *more RESTful* ones ...
  - ... and *more RESTless* ones
- RESTful (distributed) systems tend to be scalable, robust and easy to develop, understand and maintain
  - Yet not every system should be RESTful
  - Silver-bullets are like unicorns: they do not exist.

Overview II

REST constraints

RESTful systems *should* satisfy the following 6 constraints:

- Client-server
- Stateless interaction
- Chaceability
- Uniform Interface
- Layered System
- Code on Demand (Optional)

Outline

Client-server I

There are two kind of entities: *servers* and *clients* communicating via a *connector*.

Servers

Reactive entities providing one or more services to multiple clients

Clients

Triggering (proactive) entities making requests that trigger reactions from servers.

Connectors

Mechanism that allows communication between clients and servers (e.g. HTTP protocol, Message Oriented Middleware, RPC, etc.)

Client-server II

Separation of concerns is the key concept here.

Separation of concerns

Once both clients and servers concerns have been fixed and some sort of common interface have been defined, the two kind of components evolve independently.

Client-server constraint in web-based systems

Web-based systems quite often satisfy this constraint.

Stateless interaction I

This is probably the most important constraint as well as the hardest one: each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Application state is therefore kept entirely on the client.

Application state

Data that could vary by client, and per request.

Exceptions are tolerated

Immutability is a utopia. For real world problems, you should just try to minimize mutability. E.g. request rate monitoring requires mutability. However, make every effort to ensure that application state doesn't span multiple requests of your services.

Stateless interaction II

Cookies

Cookies do not necessarily violate this constraint, since the are part of each HTTP interaction.

Sessions identifiers

Assigning any sort of temporary session identifier to the some user *and* storing session data using some data structure within server's memory is inherently a violation of the constraint. Since this is the common usage of cookies, they are discouraged.

Example

A stateless communication litmus test is to turn off session cookies, and determine if the API, web service, or web application still works as designed.

Stateless interaction III

Authentication

This constraint makes authentication critical. Other approaches have been developed which are more secure and RESTful.

Authentication over HTTP - Overview

- HTTP**S** is base-assumption here
- SIDs over Cookies are stateful
- *Basic access authentication* is conceptually what we need - too naive
- *Digest access authentication* employs MD5...
- *OAuth*: is strong & complex
- *JWT* is some good trade-off IMWO

Cacheability

Cacheable responses

- Clients can cache responses
- Responses must therefore, implicitly or explicitly, define themselves as cacheable, or not
  - to prevent clients reusing stale or inappropriate data in response to further requests
- Well-managed caching partially or completely eliminates some client-server interactions
  - further improving scalability and performance

Uniform Interface I

- The uniform interface constraint defines the interface between clients and servers.
- This is probably secret of REST's simplicity and strength.
- RESTful systems expose a standard, unambiguous, clear and human-readable API because of this constraint.
  - Such a principle allows for model-driven approaches.

Uniform Interface Requirements

- Resource-based
- Manipulation of Resources Through Representations
- Self-descriptive Messages
- HATEOAS (dafuq ?!)

Uniform Interface II

Resource-Based

- RESTful systems handle *resources*: servers host resources and clients want to CRUD them
  - $\Rightarrow$ **C**reate or **R**ead or **U**pdate or **D**elete
- Resources have a *hierarchical* nature
- Resources are identified and referenced by the mean of URIs
  - $\Rightarrow$ **U**niform **R**esource **I**dentifier

Uniform Interface III

HTTP Verbs

- CRUD in Web-based systems means exploiting HTTP methods, which are often called "verbs" because of their usage within APIs:
  - POST is used for resource Creation
  - GET mean is to Read resources
  - PUT aim is to Update resources
  - DELETE is used to Delete resources
- HTTP Status codes and their general purpose semantics are part of the uniform interface too:
  - e.g. any successful request will result in a `200: Ok` or `204: No Content` status code (depending on weather the response has body or not)
  - e.g. trying to GET or DELETE any non-existent resource will result in a `404: Not Found` status code
  - e.g. POSTing an already-existing user will result in a `409: Conflict` status code

Uniform Interface IV

- Example of resource creation:
  - we want to edit some user's username from `gciatto92` to `gciatto`
  - suppose no authentication is needed

Example of non-RESTful approach

- `GET http://example.com/users?user=gciatto92&-operation=changeName&newName=gciatto`

× RPC style: the request contains the to-be-called operation

× No semantics for `GET` verb

× Which resource am I editing?

Uniform Interface V

Example of RESTful approach

- `PUT http://example.com/users/gciatto92?newName=gciatto`

✓ I'm editing the resource `gciatto92`, which is a user, composing the `users` resource, which represents the collection of registered users

✓ `PUT` verb means Update and that's what I am doing

✓ URI queries are simply a mean for payload transport

Uniform Interface VI

Manipulation of Resources Through Representations

- Resources are not accessed (CRUDed) directly but through their representation(s)
- Representations should expose resources traits to clients enabling them to do what they are allowed to, no more and no less
- Clients cannot make assumptions upon resources implementations, they can only exploit representations

Self-descriptive Messages

- Each message must include enough information to describe how to process the message itself
  - E.g. HTTP's *Content Negotiation* is a powerfull feature: use it! `Accept` and `Content-Type` headers allow different representations for resources exposing some business logic
  - Prefer JSON but try to support XML

Uniform Interface VII

HATEOAS - **H**ypermedia **A**s **T**he **E**ngine **O**f **A**pplication **S**tate

- Difficult constraint to fully accomplish IMHO
- Weak definition is enough for our concerns: *"Services responses should contain 'relational links' when they may be useful"*
- Relationships are standardized
- In completely-RESTful HTTP-based systems, relational links are the only mean used by clients to interact with servers

Uniform Interface VIII

Example

`GET http://example.com/users?limit=3`

supposing the first three users are `gciatto`, `mfrancia` and `mneri`, it should return something like:

```
[
  {
    "username" = "gciatto",
    "links" = [
      { "rel" = "next", "href" = "/users/mfrancia" },
      { "rel" = "prev", "href" = "/users/mneri" },
      { "rel" = "first", "href" = "/users/gciatto" },
      { "rel" = "last", "href" = "/users/mneri" },
      { "rel" = "self", "href" = "/users/gciatto" }
    ]
  }
  ...
]
```

Layered System

Rules

- A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way
- Intermediary servers may improve system scalability by enabling load-balancing and by providing shared caches
- Layers may also enforce security policies

In practice

DO NOT make assumption about network topology

Code on demand (optional constraint)

Rules

- Servers are able to *temporarily* extend or customize the functionality of a client by transferring logic to it that it can execute

In practice

JavaScript

Table of Contents

Outline

Steps of REST Service Creation I

Supposing we already defined some *logic architecture* of the system or we 'simply' need to wrap some pre-existing SW:

1. Define the mean(s) for user/caller authentication
2. Identify *resources* and HTTP *verbs* they support
3. Assign some *route* (URI) to each resource
    - Routes can contain one or more *query parameters*

Steps of REST Service Creation II

4. For each method supported by each route:
   - 4.1 Choose one ore more needed authentication means
   - 4.2 Define supported request/response *content-types*
     - `application/json`, `application/xml`, `text/html`, ...
   - 4.3 Define *where* to put *request's arguments*
     - Headers, Body, URI, Query
   - 4.4 Define request's arguments *structures*
     - *JSON Schema*, *DTD*, *XML Schema*, ...
   - 4.5 Define *status codes* allowed as *responses* and, for each one:
     - *Body*'s structure
     - *Headers*' structure

Steps of REST Service Creation III

Example: Pet Store - API Routes

Paths

/pets

GET /pets

POST /pets

/pets/{id}

GET /pets/{id}

DELETE /pets/{id}

Steps of REST Service Creation IV

Example: Pet Store - Route Detail

/pets/{id}

GET /pets/{id}

Description

Returns a user based on a single ID, if the user does not have access to the pet

Parameters

Name	Located in	Description	Required	Schema
id	path	ID of pet to fetch	Yes	⇄ integer (int64)

Responses

Code	Description	Schema
200	pet response	⇄ ▾pet { id: integer * name: string * tag: string }
default	unexpected error	⇄ ▾errorModel { code: integer * message: string * }

Try this operation

Steps of REST Service Creation V

Example: Pet Store - JSON Schemas (Tiping)

Model-first tools I

What if we had some formal way to model APIs?

Swagger

`http://swagger.io`

RAML - **R**ESTful **A**PI **M**odelling **L**anguage

`http://www.raml.org`

Api Blueprint

`https://apiblueprint.org/`

Model-first tools II

Desired facilities

Specification Language : allows formal (machine-readable) and intellegible (human-readable) API specification

API Console : some sort of GUI allowing for APIs navigation (usually Web-based)

Mock server : proxy server whose code is generated using some formal specification; it validates clients' requests and forwards them to some other server responsible for business-logic implementation

Server stub : code stubs generated using some formal specification for some particular platform (e.g. `express-js`); clients' requests validation is performed by the generated code, programmers only have to implement business-logic

...

Users Example I

In order to compare the three technologies the 'Users' minimal API has been modelled using each language.

Resources and methods informal description pt. 1

`/users` : The set of registered users

- `GET` : Retrieves public informations for a number of users
- `POST` : Creates one user

`/users/{username}` : The specific user whose username is specified within the path

- `GET` : Retrieves informations about the specific user *[requires authentication]*

Users Example II

Resources and methods informal description pt. 2

`/users/{username}/token` : The authentication token for a specific user

`POST` : Creates and returns one signed JWT if the provided username and password (within request's body) are correct

- Authentication: signed JWT within the `Authorization` header identifies the users
- It's an example! Let's suppose HTTP it's secure.

Outline

Swagger features

- ✓ JSON/YAML-based specification language
  - × a little verbose
- ✓ Allows to easily define type schemas
- × Only explicitly supports Basic authentication, Api Key and OAuth2
- ✓ Largest and most active developers community
- ? Industry Backing: Reverb, 3Scale, Apigee (NFI)
- ✓ Largest platform support (Clojure, Go, JS, Java, Node, .Net, PHP, Python, Ruby, Scala)
- ✓ Web-based editor (API console included) available at `http://editor.swagger.io` or as `nodejs` module
- ✓ Generates extremely detailed API which allows for in-app testing
  - × not-so-easy setup: it's a standalone server
- ✓ Lots of tools for editing (`Swagger Editor`), code generation (`Swagger Codegen`) or API presentation/navigation (`Swagger UI`)

Swagger - Users example I

Headers

```yaml
swagger: '2.0'
info:
  description: Some description here
  version: 16.05 acute angle
  title: Swagger Sample App
  contact:
    name: gciatto
    email: giovanni.ciatto@gmail.com
tags:
  - name: Users
  - name: Default
securityDefinitions:
  JWT-User:
    type: apiKey
    in: header
    name: Authorization
    x-jwt-header:
      alg: HS512
    x-jwt-payload:
      iss: localhost
      aud: localhost
      role: user
schemes:
  - http
```

Swagger - Users example II

JSON Schemas definitions (2 columns)

```yaml
definitions:
  Error:
    type: object
    required:
      - message
      - status
    properties:
      message:
        type: string
      status:
        type: integer
        format: int32
  Token:
    type: object
    required:
      - token
    properties:
      token:
        type: string
  User:
    type: object
    required:
      - admin
      - username
    properties:
      admin:
        type: boolean
        default: false
      name:
        type: string
      surname:
        type: string
      username:
        type: string
  UserAuth:
    type: object
    required:
      - password
      - username
    properties:
      password:
        type: string
      username:
        type: string
```

Swagger - Users example III

Paths (columns 1-2 / 4)

```yaml
paths:
  /users:
    get:
      tags:
        - Users
      operationId: getUsers
      consumes: []
      produces:
        - application/json
      parameters:
        - name: limit
          in: query
          required: false
          type: integer
          format: int64
          x-example: 10
        - name: offset
          in: query
          required: false
          type: integer
          format: int64
          x-example: 0
      responses:
        '200':
          description: |
            The request has succeeded
          schema:
            $ref: '#/definitions/User'
    post:
      tags:
        - Users
      operationId: postUsers
      consumes: []
      produces: []
      parameters: []
      responses:
        '200':
          description: OK
  '/users/{username}':
    get:
      tags:
        - Users
      consumes: []
      produces:
        - application/json
      parameters:
```

Swagger - Users example IV

Paths (columns 3-4 / 4)

```
      - name: username
        in: path
        required: true
        type: string
    responses:
      '200':
        description: Success
        schema:
          $ref: '#/definitions/User'
      '401':
        description: Error 401
        schema:
          $ref: '#/definitions/Error'
      '404':
        description: Error 404
        schema:
          $ref: '#/definitions/Error'
    security:
      - JWT-User: []
'/users/{username}/token':
  post:
    tags:
      - Users
    consumes:
      - application/json
    produces:
      - application/json
    parameters:
      - name: username
        in: path
        required: true
        type: string
      - in: body
        name: body
        required: false
        schema:
          $ref: '#/definitions/UserAuth'
    responses:
      '200':
        description: Success
        schema:
          $ref: '#/definitions/Token'
      '401':
        description: Error 401
        schema:
          $ref: '#/definitions/Error'
```

Swagger - How to 'run' the example

1. Open some web browser and go to `http://editor.swagger.io`
2. Copy & paste the YAML code
   - Pay attention to indentation!
3. As you will notice, the API console on the right allows you to navigate the specification
4. The '`Generate Server`' allows you to generate the server's stub
5. Refer to `http://swagger.io/open-source-integrations/` to see the complete tools gamma

Outline

RAML features

- ✓ YAML-based specification language
  - × for more concise than Swagger
- × RAML v. 0.8 uses raw JSON-Schemas (very verbose)
- ✓ Supports Basic & Digest authentication, OAuth 1 & 2 and explicitly supports custom authentication
- × Poor community support
- ✓ Largest industry Backing: MuleSoft, SOA, AngularJS, Intuit, Box, PayPal & Programmable Web
- ✓ Large platform support (JavaScript, Java, Node, PHP, Python, Ruby)
  - × v. 1.0 is still poorly supported
- ✓ Editor plugin for Atom IDE (API console included)
- ✓ Generates easy-to-read API into a single `.html` or `.md` file through some community tool
- × Too much community (poorly supported) tools

RAML - Users example

TODO

Insert example here

Table of Contents

Outline

Overview I

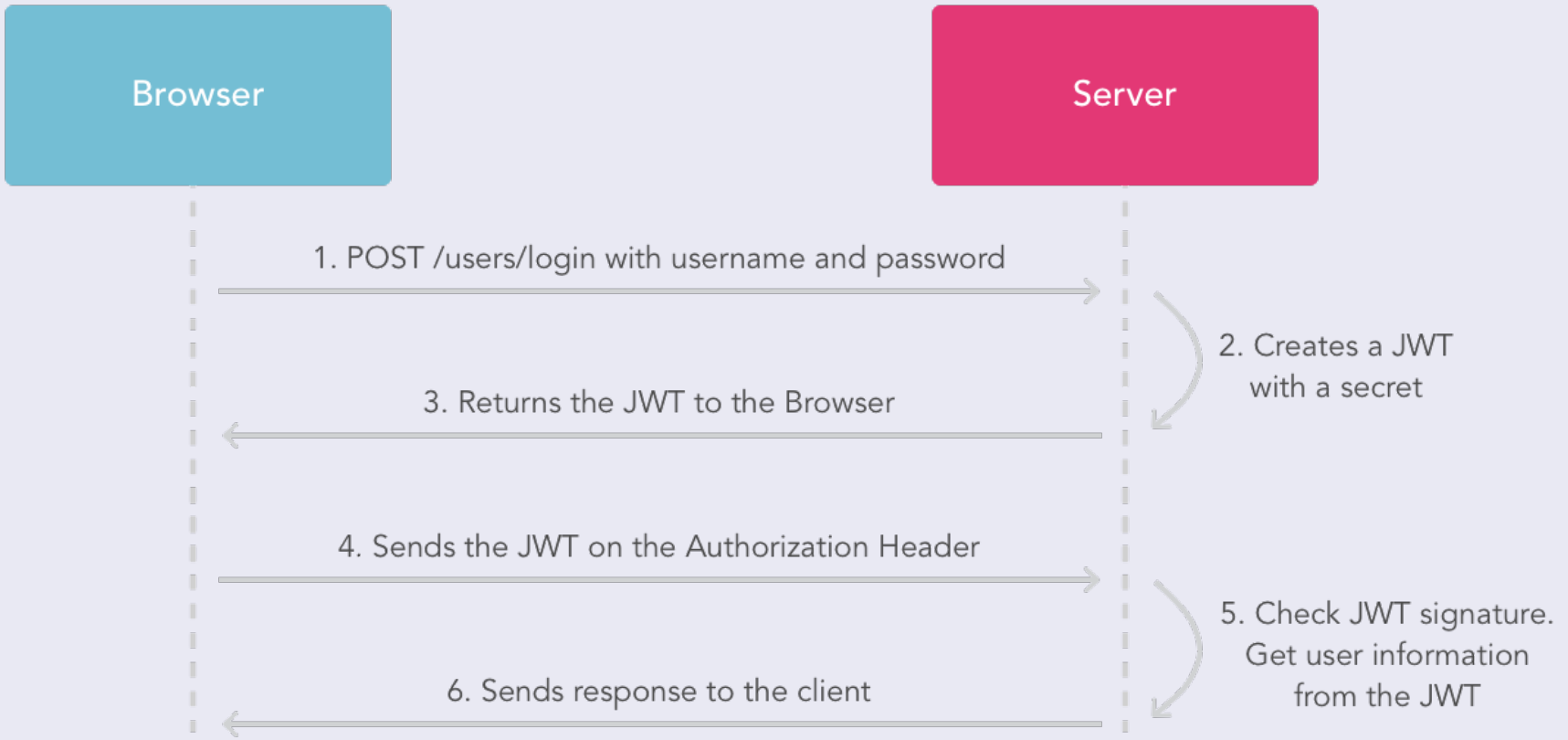
JWT*s* = **J**SON **W**eb **T**oken*s*

- Simple and stateless mean to provide authentication
  - the server doesn't need to store session data
- Use case:
  1. Some client POSTs username & password to some predefined route, e.g. `/users/{username}/login`
  2. On valid requests, the server responds with one *token*

     N.B. the token is *signed* using some secret that never leaves the server, so the token **cannot be compromised or forged**
  3. The client stores the token, e.g. within its *local storage*
  4. Any subsequent request from the client should contain the token, e.g. within the `Authorization` header, the query or the body
  5. The server should verify the signature of every request containing one token: once validated, any *claim* contained within the token can be entrusted

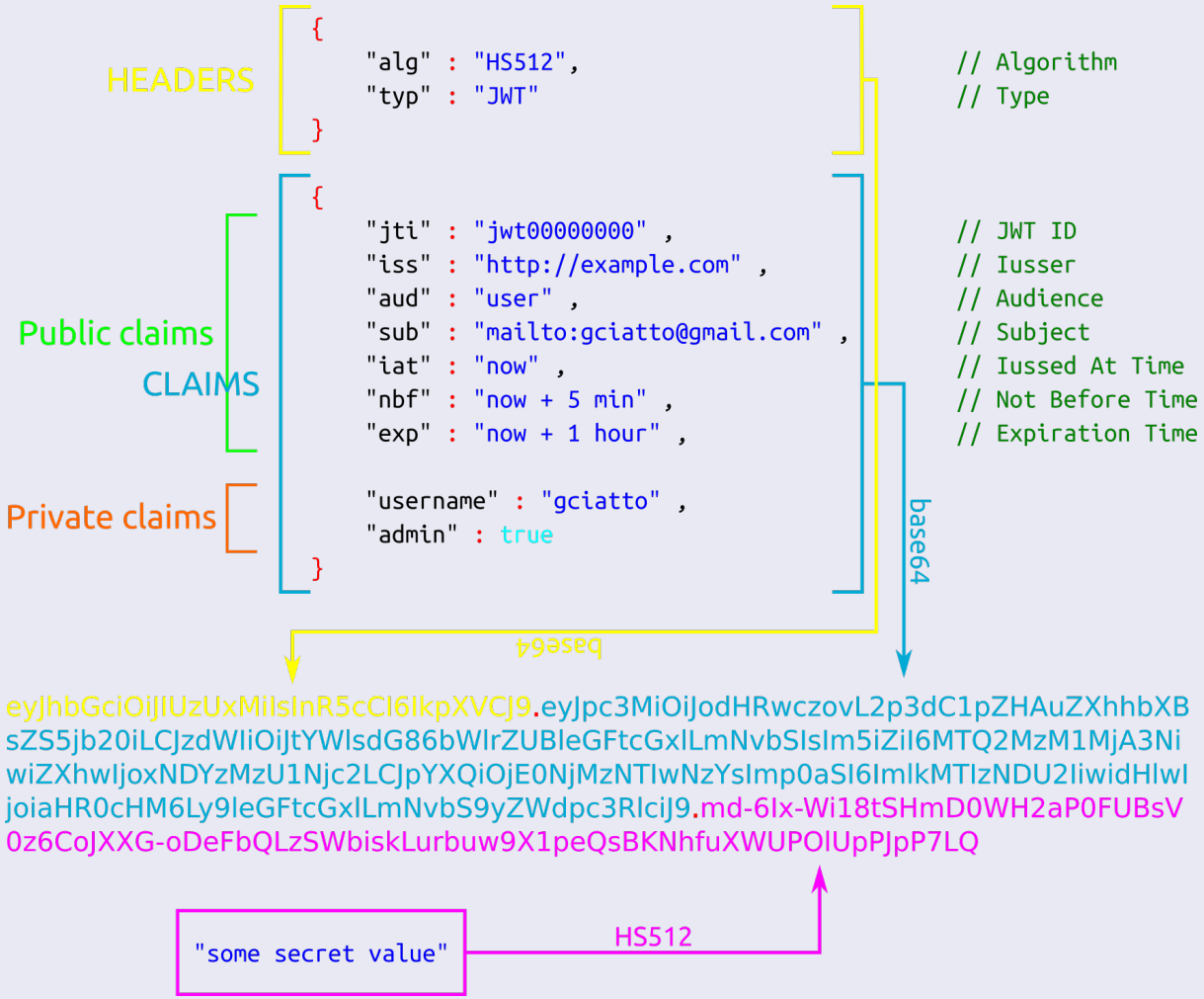
Overview II

JWT - Sequence Diagram



JWT Structure I

JWT - Anathomy Diagram



JWT Structure II

JWT - Sections

- Defined in `RFC 7519`
- String containing 3 dot-separated section
  1. The first section is a JSON object, called *Headers*, `base64` encoded
  2. The second section is a JSON obect, called *Claims*, `base64` encoded
  3. The third section is the *Signature*, obtained using `HMAC` and some *secret* string

JWT - Headers

`typ` (Type) : usually `"JWT"`

`alg` (Algorithm) : the hash algorithm used (`HS512` = `HMAC SHA 512`)

JWT Structure III

JWT - Public claims

`jti` (JWT Identifier) : identifies the entity that issued the JWT

`iss` (Issuer) : URI of the entity issuing the token

`aud` (Audience) : identifies the recipients that the JWT is intended for

`sub` (Subject) : identifies the entity that is the subject of the JWT

`iat` (Issued At) : identifies the time at which the JWT was issued

`nbf` (Not Before) : identifies the time before which the JWT *musth not* be accepted for processing

`exp` (Expiration) : identifies the expiration time on or after which the JWT *musth not* be accepted for processing

JWT Structure IV

JWT - Private claims

- They are user-defined claims
- REMEMBER: they are public (since the token is not encrypted) but trusted (since they are signed)
- E.g., within the image both username and privilege level are claimed

Further Reading I

Fielding, Roy Thomas
Architectural Styles and the Design of Network-based Software Architectures.
Doctoral dissertation, University of California, Irvine, 200.
http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

RestApiTutorial.com
Learn REST: A RESTful Tutorial
http://www.restapitutorial.com/

Swagger Specification
http://swagger.io/specification/

RAML 1.0 Specification
https://github.com/raml-org/raml-spec/blob/raml-10/versions/raml-10/raml-10.md/

Further Reading II


API Blueprint Specification
https://github.com/apiaryio/api-blueprint/blob/master/API%20Blueprint%20Specification.md


Stowe, Mike
RAML vs. Swagger vs. API Blueprint
http://www.mikestowe.com/blog/2014/07/raml-vs-swagger-vs-api-blueprint.php


RFC 7519
https://tools.ietf.org/html/rfc7519