Web APIs

JSON – APIs Restfull

OAuth 2.0 – Open API

Agenda

- JSON
- Servicios REST o Web APIs
- OAuth2
- Open API Specification

JavaScript Object Notation (JSON)

LINS
Laboratorio de Integracion de Sistemas

¿Qué es?

- Es un formato liviano para el intercambio de información
- Está basado en una parte del lenguaje JavaScript
- Es independiente de una plataforma

Sintaxis

- Los datos se representan como elementos de tipo clave-valor
- Los datos se separan por comas
- Los caracteres ‘{‘ y ‘}’ permiten representar objetos
- Los caracteres ‘[‘ y ‘]’ permiten representar arreglos

Ejemplo

- Valores:

```
{"nombre":"Juan"}
```

- Objetos:

```
{"persona": {"id":1, "nombre":"Juan", "apellido":"Perez"}}
```

Ejemplo

- Arreglos

```
{
    "personas": [{
        "persona": {
            "id": 1,
            "nombre": "Juan",
            "apellido": "Perez"
        }
    }, {
        "persona": {
            "id": 2,
            "nombre": "Sara",
            "apellido": "Connor"
        }
    }]
}
```

JSON vs XML

JSON vs XML

- Ambos pueden usarse para el intercambio de información
- Ambos son autodescriptivos
- Ambos son jerárquicos (valores dentro de valores)
- JSON es más corto
- JSON es más rápido de leer/escribir
- JSON permite arreglos

REpresentational State Transfer (REST)

LINS
Laboratorio de Integracion de Sistemas

Servicios REST

Motivación

- WS-* es muy completo pero
  - Es complejo
    - Gran cantidad de estándares
    - Existen muchos más a los vistos en el curso
  - Es pesado
    - Recordar serialización SOAP

- Es necesario algo más simple y liviano
  - REST!

REST

- Representational State Transfer
- Es un estilo arquitectónico para aplicaciones que utilizan hipermedia interconectada
- Es aplicado para la construcción de servicios web, livianos, mantenibles y altamente escalables
- Un servicio basado en REST se denomina RESTfull
- Todo servicio RESTful utiliza HTTP/HTTPS como protocolo de transporte

Características principales

- Representaciones
- URIs
- Interfaz uniforme
- Falta de estado (Stateless)
- Uso de caches

Representaciones

- En REST todo es un recurso
  - Puede verse como un objeto en OOP
- Un recurso puede estar compuesto o relacionados a otros recursos
- Primer paso:
  - Identificar los recursos y sus relaciones
- Segundo paso:
  - Definir sus formas de representación.

Representaciones

- Ejemplos de recursos
  - Información de una persona
  - Las ventas del primer trimestre del año
  - La relación entre dos personas
  - Una entrada en un blog
  - Una lista de bugs en nuestra aplicación
  - La versión de un software

Representaciones

- Recurso: Persona
- Representación:
  - JSON: Deseable para que una página web realice invocaciones vía Javascript.
  - XML: Deseable para algunas comunicaciones B2B

```
{
    "Id": "1",
    "Name": "Pablo",
    "Email": "pablo@gmail.com",
    "Country": "Uruguay"
}
```

```
<Persona>
    <Id>1</Id>
    <Name>Pablo</Name>
    <Email>pablo@gmail.com</Email>
    <Country>Uruguay</Country>
</Persona>
```

Representaciones

- Una representación debe ser capaz de enlazar representación de recursos entre sí.
  - Se pueden usar referencias a otros recursos
  - HATEOAS:
    - Hipertext As The Engine Of Application State

```
<Persona>
    <Id>1</Id>
    <Name>Pablo</Name>
    <Email>Pablo@gmail.com</Email>
    <Link rel="address" href="http://services/address/4324" />
    <Country>Uruguay</Country>
</Persona>
```

Direccionamiento de recursos

- REST requiere que todo recurso tenga una identificación única
  - Una URI identifica un recurso
- La acción sobre el recurso se encuentra especificada por el verbo HTTP en la comunicación con el servicio Rest.
- Una URI NO debe decir nada sobre la acción a realizar
  - Esto permite invocar la misma URI con diferentes verbos HTTP para realizar diferentes acciones
- Por ejemplo, un recurso “persona” puede estar expuesto con esta URI http://myservice/personas/123

URIs

- Generalmente las URIs para acceder a recursos tienen el siguiente formato:
  - Protocol://ServiceName/ResourceType/ResourceID
- Algunas recomendaciones
  - Usar sustantivos en plural para nombrar un recurso
  - Evitar los espacios, usar "_" o "–"
  - Las URIs son sensibles al case.
- Evitar los verbos para describir la URI de un recurso
  - http://myservice/FetchPerson/1, http://myservice/DeletePerson/1
  - http://myservice/pagos/confirmar

URIs y query parameters

- Se utilizan query parameters para:
  - Filtros en búsquedas:
    - http://myservice/personas?name=juan
  - Ordenamiento de resultados
    - http://myservice/personas?sort=age
  - Personalizar la respuesta
    - http://myservice/personas/1?format=xml

- NO se deben utilizar para identificar recursos
  - http://myservice/personas?id=1

Interfaz uniforme

Método	Acción realizada en el servidor
GET	Lee un recurso
PUT	Inserta un nuevo recurso o lo actualiza si este ya existe
POST	Inserta un nuevo recurso
DELETE	Elimina un recurso
OPTIONS	Lista las acciones sobre un recurso
PATCH	Actualiza parcialmente un recurso

Seguridad e idempotencia

- Seguro
  - El método HTTP no modifica el estado del recurso
- Idempotente
  - No importa la cantidad de veces que se ejecute el método, el resultado siempre es el mismo

Interfaz uniforme

Método	Acción realizada en el servidor	¿Seguro o idempotente?
GET	Lee un recurso	
PUT	Inserta un nuevo recurso o lo actualiza si este ya existe	
POST	Inserta un nuevo recurso	
DELETE	Elimina un recurso	
OPTIONS	Lista las acciones sobre un recurso	
PATCH	Actualiza parcialmente un recurso	

¿Cuáles de estos métodos son seguros y cuales idempotentes?

Interfaz uniforme

Método	Acción realizada en el servidor	Característica
GET	Lee un recurso	Segura
PUT	Inserta un nuevo recurso o lo actualiza si este ya existe	Idempotente
POST	Inserta un nuevo recurso	N/A
DELETE	Elimina un recurso	Idempotente
OPTIONS	Lista las acciones sobre un recurso	Segura
PATCH	Actualiza parcialmente un recurso	Puede ser idempotente

PUT vs POST

Solicitud	Acción
PUT http://myservice/personas	No funciona, PUT requiere una URI específica
PUT http://myservice/personas/123	Si no existe, inserta la persona con id=123. Si existe, la sobreescribe. Los datos de la persona van en el body http
POST http://myservcie/personas	Inserta una nueva persona cada vez que se realiza esta solicitud. Cada recurso persona se asocia a un nuevo Id definido por el servicio. La información de la persona va en el body http.

En ambos casos, es deseable retornar en la respuesta el recurso creado.

PATCH vs PUT

- En caso de existir el recurso, PUT hace un remplazo del recurso. El recurso previo se pierde
- PATCH permite dos tipos de funcionamiento:
  - Actualizar los datos presentes en el body http. Realiza actualizaciones parciales del recurso
  - Realizar un conjunto de operaciones como modificar los datos: JSON Patch y JSON Merge
- En ambos casos es deseable retornar el recurso actualizado en la respuesta

PATCH e idempotencia

Idempotente

```
PATCH /users/1

{
    "email": "gllambi@fing.edu.uy"
}
```

Reemplaza el valor del correo del usuario

No Idempotente

```
PATCH /users/1

[
    {
        "op": "add",
        "email": "gllambi@fing.edu.uy"
    }
]
```

Agrega un correo al usuario

Interfaz uniforme

- Los métodos indicados deben ser usados solo para el propósito con el que fueron definidos
  - Ej: Nunca debería usarse un GET para crear un recurso o para eliminarlo
- Si bien HTTP provee una interfaz uniforme, queda en manos del desarrollador del servicio el uso que se les da
  - Ejemplos: PUT, PATCH, POST

Codigos HTTP

Codigo HTTP	
200 OK	Usado por GET, PUT, PATCH, DELETE en respuestas ok o POST en caso que no se haya creado el recurso
201 Created	Recurso creado en POST. Debe ir en conjunto un link (Location header) a la ubicación del nuevo recurso
204 No Content	Respuesta a una solicitud que no retorna un recurso (p.ej: DELETE)
400 Bad Request	Está mal formada la solicitud
401 Unauthorized	Cuando no se pudo autenticar al cliente
403 Forbidden	Cuando se autenticó correctamente al cliente pero no tiene permiso de acceso al recurso
404 Not Found	No se pudo encontrar el servicio
500 Internal Server Error	Error interno al procesar la solicitud

Escenario ejemplo

- Una empresa de venta de entradas desea implementar su servicio de venta mediante servicios REST. Mediante este servicio es posible consultar, pagar, anular e imprimir tickets de entrada.

Escenario ejemplo

Operación	VERBO + URI	Descripción
Consultar disponibilidad	GET http://myservice/tickets?evento=123&sort=precio	Se retornan los tickets disponibles para el evento, ordenados por precio
Comprar un ticket	POST http://myservice/ventas	En el body http se envía el recurso ticket a asociar a la venta
Anular un ticket	PATCH http://myservice/ventas/456	En el body http se envía el nuevo estado de la venta como anulado
Imprimir ticket	GET http://myservice/impresiones?tickets=265	Se retorna el recurso impresión del ticket con id 265

Falta de estado

- Un servicio RESTful no mantiene el estado de aplicación para ningún cliente
  - El servicio trata cada solicitud como una solicitud independiente
  - En ese sentido, las solicitudes que llegan al servicio no pueden estar correlacionadas por él
- Stateless
  - Request1: GET http://MyService/Persons/1 HTTP/1.1
  - Request2: GET http://MyService/Persons/2 HTTP/1.1
- Stateful
  - Request1: GET http://MyService/Persons/1 HTTP/1.1
  - Request2: GET http://MyService/NextPerson HTTP/1.1

Caching

- Caching es el concepto de almacenar los resultados generados para reutilizarlos, en lugar de generarlos en cada oportunidad
- Esto puede ser realizado en el cliente, en el servidor o en cualquier componente intermedio
- Si bien es una herramienta poderosa para mejorar la performance, si no se usa con cuidado, se le pueden enviar al cliente resultados obsoletos
- Podemos controlar el caching de los resultados a través de una serie de cabezales HTTP

Caching

Cabezal HTTP	Aplicación
Date	Fecha y hora en la cual el recurso fue creado
Last modified	Fecha y hora de última modificación del recurso en el servicio
Cache-control	Cabezal HTTP para controlar el uso del caché
Expires	Fecha de expiración del recurso. Para uso de clientes HTTP 1.0
Age	Duración en segundos desde que se tomó el recurso del servicio. Puede ser insertado por un intermediario.

Errores comunes

- Usar REST como un mecanismo para hacer invocaciones RPC
- Abusar del uso de POST
- Colocar acciones en la URI
- Enmascarar un servicio como un recurso
- Mantener sesiones en el servidor

Ejemplos

- Twitter
  - https://developer.twitter.com/en/docs/api-reference-index
- Facebook
  - https://developers.facebook.com/docs/graph-api/using-graph-api/

REST vs SOAP

Característica	REST	SOAP
Sintáxis	XML, JSON (más popular)	XML
Formato	Ninguno definido	SOAP
Interfaz	WADL, Swagger o RAML (no estándar)	WSDL
Transporte	HTTP	Potencialmente varios HTTP más popular
Orientado a…	Recursos	Operaciones
Propósito	Escalabilidad, performance, sencillez	Interoperabilidad
Operaciones	Limitadas	Ilimitadas
Seguridad	HTTPS, HTTP Basic Authentication, OAuth2	WS-*
Otros requerimientos empresariales	N/A	WS-*

REST vs SOAP

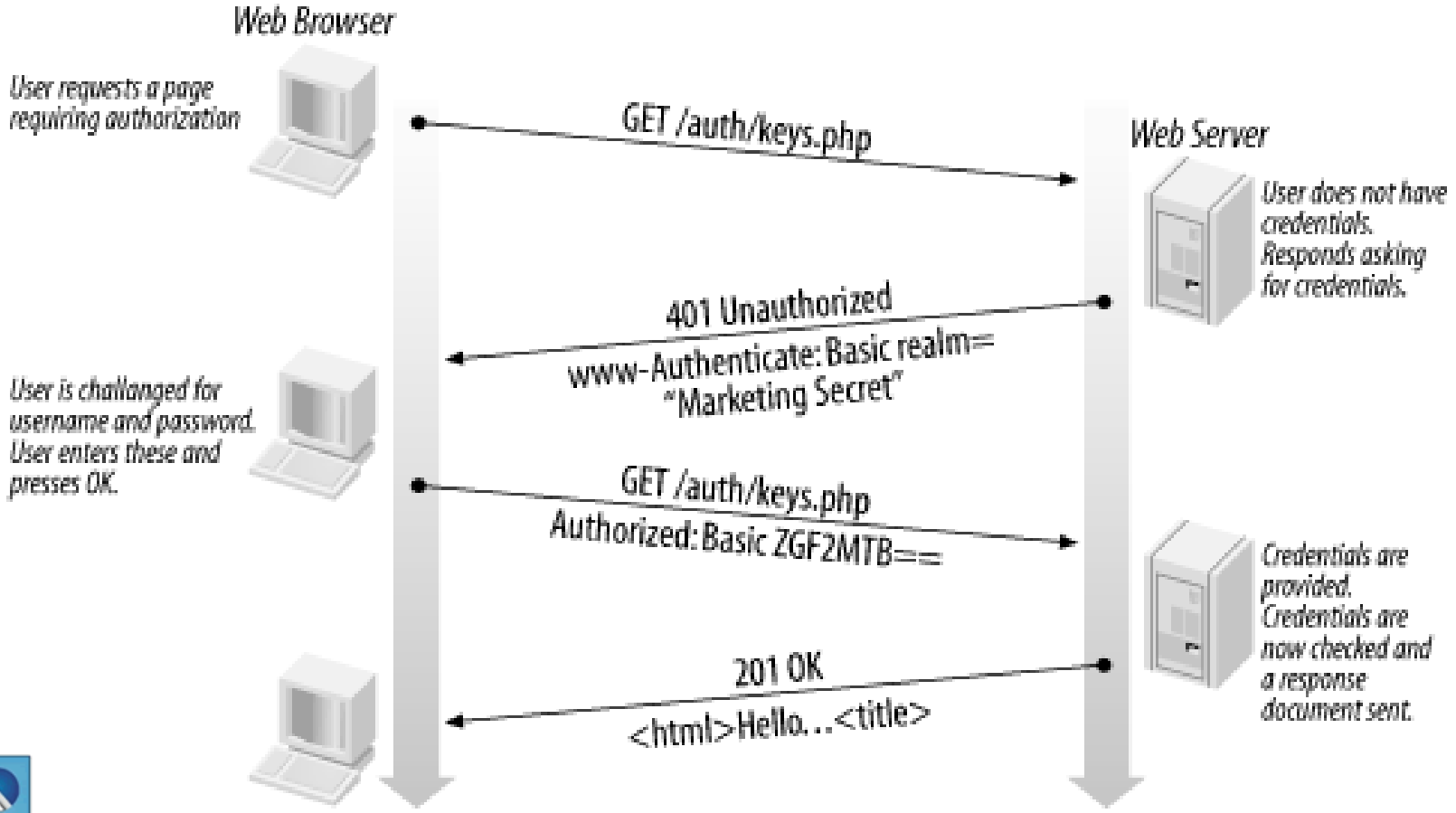
- Ninguno es mejor que el otro a priori
- Ambos tienen mismas capacidades
- SOAP se originó para para integrar sistemas heterogéneos con requerimientos empresariales específicos
- REST está orientado a APIs Web con gran cantidad de clientes y desconocidos
  - Simplicidad
  - Escalar en clientes
  - Ambientes mobile

OAuth2

Seguridad en servicios RESTfull

Http Basic Authentication

- Permite autenticación de aplicaciones Web



¿Es suficiente?

- Es suficiente si autenticación => autorización
  - Estoy autenticado, entonces estoy autorizado.

- Sin embargo, esto no es suficiente en todos los escenarios.
  - El servicio debe poder tomar decisiones de autorización basadas en información de mayor granularidad sobre el autenticado.
  - Por ejemplo, autorización basada en roles siguiendo el modelo RBAC (Role Based Access Control)

Escenario

- Se quiere permitir acceso al recurso */printer* solo a los usuarios con el rol *admin*

- ¿Cómo implementarían esta solución?

Limitaciones

Limitaciones

- No escala a medida que aumentan los servicios y usuarios (aplicaciones cliente)
  - Nuevos servicios requieren configurarse para identificar qué usuario tiene acceso.
  - Servicios existentes deben configurarse para permitir nuevos usuarios
  - Cambios en roles deben configurarse en cada servicio

Solución

- Se necesita de una gestión de identidades centralizada y de un servicio que pueda proveer y administrar información de una forma segura
- Que no envíe las credenciales en cada invocación.
- OAuth 2.0 provee soluciones a este problema, introduciendo una nueva capa en la arquitectura de nuestras aplicaciones.

¿Qué es OAuth 2.0?

- The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf

  - *RFC 6749, 2012*

Especificaciones Core

- OAuth 2.0 Core Framework (RFC 6749)
- Bearer Token Usage (RFC 6750)
- Threat Model and Security Considerations (RFC 6819)

Especificaciones adicionales

- Token Introspection (RFC 7662)
- Token Binding
- OAuth 2.0 for Native Apps
- PKCE: Proof Key for Code Exchange (RFC 7636)
- Device Flow
- OpenID Connect
- IndieAuth
- Token Revocation (RFC 7009)
- Authorization Server Metadata (RFC 8414)

Atención

- En el curso solo se presenta la parte de la especificación que aplica a servicios Resftull.

- La especificación provee entre otras cosas, soluciones para la autenticación de usuarios Web, aplicaciones móviles y dispositivos (Internet of Things).

Roles

- Client: una aplicación de terceros
- Resource Server: La API
- Authorization Server: Emisor de tokens
- Resource Owner: el usuario físico.

Tipos de permisos

- Authorization code:
  - para aplicaciones que corren en un servidor de aplicaciones, basadas en un navegador o aplicaciones móviles.
  - Se autentican usuarios físicos
- Client credentials:
  - Autenticación de aplicaciones
- Password:
  - Para autenticarse con usuario y contraseña
  - Desacosejado su uso por vulnerabilidades de seguridad
- Implicit:
  - Orientadas a aplicaciones que no permiten el uso de secretos (ej: app web tipo one page). En desuso por Authorization code
    - https://tools.ietf.org/html/draft-ietf-oauth-security-topics-16

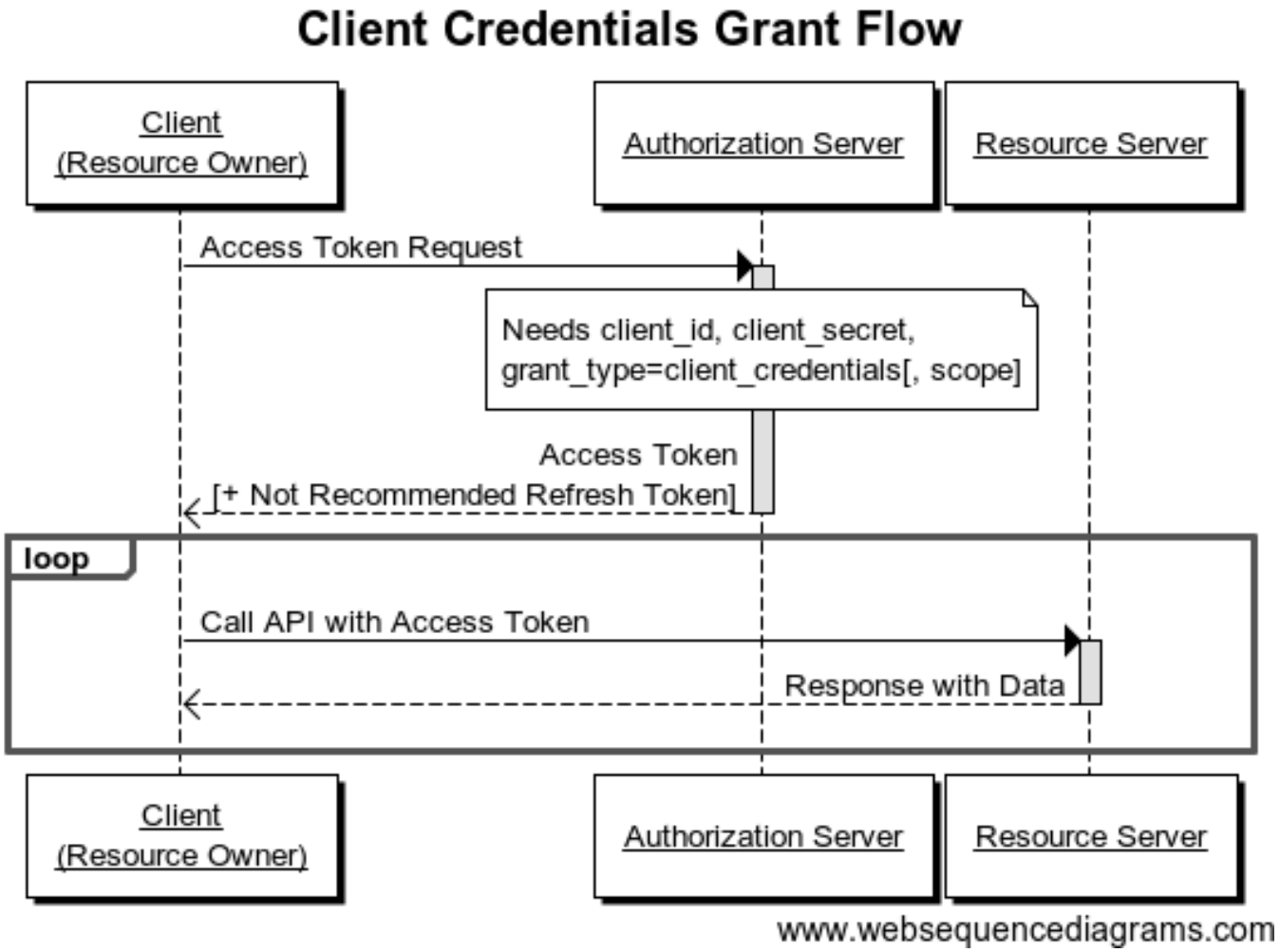
Tipos de permisos

- Authorization code:
  - para aplicaciones que corren en un servidor de aplicaciones, basadas en un navegador o aplicaciones móviles.
  - Se autentican usuarios físicos
- Client credentials:
  - Autenticación de aplicaciones
- ~~Password:~~
  - ~~Para autenticarse con usuario y contraseña~~
  - ~~Desacosejado su uso por vulnerabilidades de seguridad~~
- ~~Implicit:~~
  - ~~Orientadas a aplicaciones que no permiten el uso de secretos (ej: app web tipo one page). En desuso por Authorization code~~
    - https://tools.ietf.org/html/draft-ietf-oauth-security-topics-16

Tipos de permisos

- Authorization code:
  - para aplicaciones que corren en un servidor de aplicaciones, basadas en un navegador o aplicaciones móviles.
  - Se autentican usuarios físicos
- Client credentials:
  - Autenticación de aplicaciones
- ~~Password:~~
  - ~~Para autenticarse con usuario y contraseña~~
  - ~~Desacosejado su uso por vulnerabilidades de seguridad~~
- ~~Implicit:~~
  - ~~Orientadas a aplicaciones que no permiten el uso de secretos (ej: app web tipo one page). En desuso por Authorization code~~
    - https://tools.ietf.org/html/draft-ietf-oauth-security-topics-16

Client Credential Grant Flow



Ejecución OAuth 2.0

1. La aplicación cliente solicita un token de seguridad al Authorization Server
   - Se autentica con algún mecanismo de autenticación. Recomendado usar HTTP Basic Authorization
     - ClientID: nombre de la aplicación cliente
     - Client secret: contraseña de la aplicación cliente
   - Opcionalmente puede solicitar un permiso de uso (scope)
2. El Authorization server autentica a la aplicación cliente y genera token
3. La aplicación cliente invoca al Resource Server, adjuntando token.
4. El Resource Server valida el token y autoriza el acceso al recurso.
   - La validación del token puede implicar consultar al Authorization Server.
5. El Resource Server retorna una respuesta a la aplicación cliente.

Ejemplo de solicitud de token

```
POST /oauth/token HTTP/1.1
Accept: application/json

grant_type=client_credentials
&client_id=YOUR_CLIENT_ID
&client_secret=YOUR_CLIENT_SECRET
```

Ejemplo de solicitud de token

```
POST /oauth/token HTTP/1.1
Accept: application/json

grant_type=client_credentials
&client_id=YOUR_CLIENT_ID
&client_secret=YOUR_CLIENT_SECRET

Response: {
      "access_token":"RsT5OjbzRn430zqMLgV3Ia",
      "refresh_token":" RjY2NjM5NzA2OWJjuE7c",
      "token_type":"bearer",
      "expires_in":3600,
      "scope": "read"
```

Alternativa

- Usar HTTP Basic Authentication

```
POST/oauth/token?grant_type=client_credentials HTTP/1.1
Accept: application/json
Authorization: Basic [clientID:Secret]
```

- Usar TLS con autenticación mutua
  - Cliente presenta un certificado para autenticarse

Invocación al recurso

- Se adjunta el token solicitado en un cabezal HTTP

```
POST /personas HTTP/1.1
Accept: application/json
Authorization: Bearer [TOKEN]
```

Scopes

- Es una representación de los permisos que se tienen sobre un recurso protegido.

- Los scopes se representan como strings
  - Ejemplo: ‘read’
- Un conjunto de scopes se representa como un conjunto de strings separados por un espacio
  - Ejemplo: ‘read write’

Definición de scopes

- Oauth no especifica cómo definir scopes
- Recomendación:
  - Comenzar con read/write y luego evolucionar a scopes más sofisticados
- Github posee un scope independiente para:
  - Acceder a los repositorios privados
  - Borrar repositorios

Scopes Github

- user
  - Read/write access to profile info only.
- public_repo
  - Read/write access to public repos and organizations.
- repo
  - Read/write access to public and private repos and organizations.
- delete_repo
  - Delete access to adminable repositories.
- gist
  - write access to gists

Scopes Facebook

Laboratorio de Integracion de Sistemas

User permission	Friends permission	Description
user_about_me	friends_about_me	Provides access to the "About Me" section of the profile in the about property
user_activities	friends_activities	Provides access to the user's list of activities as the activities connection
user_birthday	friends_birthday	Provides access to the birthday with year as the birthday property
user_checkins	friends_checkins	Provides read access to the authorized user's check-ins or a friend's check-ins that the user can see. This permission is superseded by user_status for new applications as of March, 2012.
user_education_history	friends_education_history	Provides access to education history as the education property
user_events	friends_events	Provides access to the list of events the user is attending as the events connection
user_groups	friends_groups	Provides access to the list of groups the user is a member of as the groups connection

JSON Web Token (JWT)

- Estándar (RFC 7519) que define de forma compacta y autocontenida una forma de transmitir información entre entidades utilizando el formato JSON.

- La información se puede verificar mediante el uso de firmas digitales con claves simétricas y asimétricas (clave pública/privada).

¿Cuándo utilizar JWT?

- Autorización
  - Necesito incluir información de acceso en el token (Recursos, servicios, etc)

- Intercambio de información:
  - Es una forma segura de transmitir información entre dos partes.
  - Firma permite identificar el origen y que no se modificó

Estructura

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

Header

- Indica tipo de token y algoritmo de hash para la firma.
  - Un valor *none* en *alg* significa que el token no está firmado

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Payload

- Contiene afirmaciones sobre el usuario (claims)
- Tres tipos:
  - Registradas: predefinidas, no obligatorias pero recomendadas
    - Issuer, expiration, subject, audience
  - Públicas:
    - Personalizadas y registradas en IONA JWT Registry
  - Privadas:
    - Especificas a la organización

Payload

- Ejemplo

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

Signature

- Contiene la firma del token
- Permite asegurar que el token no fue modificado

JWT + OAuth 2.0

- El token se adjunta en el cabezal HTTP Authorization codificado en base64
  - Formato: [Header].[Payload].[Signature]

```
POST /personas HTTP/1.1
Authorization: Bearer eyJhbGciOiJIU…ssw5c
```

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ
zdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJ
SMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Escenario ejemplo

- Una empresa de venta de entradas desea implementar su servicio de venta mediante servicios REST. Mediante este servicio es posible consultar, pagar, anular e imprimir tickets de entrada.
- Las consultas son públicas, sin embargo para pagar e imprimir entradas se requiere autenticar al usuario. La anulación solo se pueden hacer desde los sistemas de la empresa en sus puntos de venta.

Comentarios sobre Oauth 2.0

- Es un protocolo de delegación
- Se conoce como protocolo de autorización, debido al nombre en el RFC pero no lo es.
- El protocolo no indica cómo autorizar
- Sí brinda los mecanismos para que una aplicación pueda actuar en nombre de un usuario.
- Los recursos son los encargados de llevar a cabo la autorización en base a la información del token.

Qué no es

- No es para cualquier protocol (HTTP)
- No es un protocol de autenticación
- No define mecanismos de delegación usuario-usuario
- No define mecanismos de autorización
- No define un formato de token
- No define mecanismos de encriptación
- No es un único protocolo

Open API Specification

LINS

Laboratorio de Integracion de Sistemas

Definición

- The OpenAPI Specification (OAS) defines a standard, language-agnostic interface to RESTful APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection. When properly defined, a consumer can understand and interact with the remote service with a minimal amount of implementation logic.

Open API Specification, version 3.0.3

¿Qué es?

- Es una especificación que permite describir APIs Restfull
- Estándar de facto
- Construida sobre el lenguaje YAML y JSON

Beneficios

- Generación de código automático
  - Clientes y APIs
- Generación de documentación automática a partir del código
- Desarrollo de herramientas para facilitar el testing
  - Ejemplos: Swagger Inspector, SoapUI y LoadUI

APIs a documentar: Blog API

- Obtener todas las entradas del blog (posts)
  - GET /posts
- Crear una entrada de blog
  - POST /posts
- Obtener una entrada de blog
  - GET /posts/{id}
- Obtener los comentarios de un post
  - GET /comments?postId={id}

APIs a documentar: Blog API

Laboratorio de Integracion de Sistemas

Estructura de un post

```
{
    "userId": 1,
    "id": 1,
    "title": "The First Post",
    "body": "we are building a blog post API using OpenAPI
Specification."
}
```

Estructura de un comentario

```
{
    "postId": 1,
    "id": 1,
    "name": "A comment",
    "email": "person@example.com",
    "body": "like it!"
}
```

https://medium.com/@amirm.lavasani/restful-apis-tutorial-of-openapi-specification-eeada0e3901d

Objetos Open API

- Metadata
- Info
- Server
- Path
- Parameter
- Request
- Response
- Schema

Metadata Object

- Una especificación OAS comienza con el tag *openapi* indicando la versión de la especificación utilizada

```
openapi: 3.0.2
```

Info Object

- Permite definir metadatos de la API
  - Título
  - Descripción
  - Términos y condiciones de uso
  - Contacto
  - Licenciamiento
  - Versión
  - …

Info Object

```yaml
openapi: 3.0.2

info:
  title: Blog Posts API
  description: >
    This is an example API for blog posts using OpenApi Specification.

    ---

    Using this API you can retreive blog posts, comments on each blog
    post and delete or update the posts.
  termsOfService: "http://swagger.io/terms/"
  contact:
    name: Amir Lavasani
    url: "https://amir.lavasani.dev"
    email: amirm.lavasani@gmail.com
  license:
    name: "Apache 2.0"
    url: "http://www.apache.org/licenses/LICENSE-2.0.html"
  version: 1.0.0
```

Server Object

- Permite definir los servidores asociados a la API
  - Ej: servidor de producción, testing, etc

- Es posible parametrizar las URLs de los servidores

Server Object

```yaml
servers:
  - url: '{protocol}://{environment}.example.com/v1'
    variables:
      environment:
        default: api    # Production server
        enum:
          - api         # Production server
          - api.dev     # Development server
          - api.staging # Staging server
      protocol:
        default: https
        enum:
          - http
          - https
```

Path Object

- Permite definir los recursos (paths) de nuestra API y las operaciones que podemos realizar sobre ellos.
- Para armar la url completa del recurso, se hace un append de la url del server y la especificada en esta sección
- Open API soporta varias operaciones http
  - GET, POST, PUT, DELETE, PATCH, HEAD

Path Object

- Cada operación puede tener los siguientes elementos:
  - Summary:
    - Resumen de la API
  - Description:
    - Descripción detallada de la operación
  - Tags:
    - una lista de etiquetas para clasificar la API
  - OperationId:
    - Nombre del método qué implementa en el código la API

Path Object

- Cada operación puede tener los siguientes elementos:
  - Parameters:
    - Son parámetros que acepta el endpoint de la API
      - Ej: cabezales HTTP
  - RequestBody
    - Estructura del cuerpo del mensaje request HTTP
  - Response
    - Estructura del cuerpo del mensaje response HTTP
  - …

Path Object

```
paths:
  /posts:
    get:
      tags:
        - Posts
      summary: Return all the posts
      description: Return all the posts that are in our blog.
      responses:
```

Parameters Object

- Son parámetros que acepta la API

- Parámetros en la URL
  - /posts/{id}
- Parámetros como query parameters
  - /comments?postId={Id}
- Cabezales http
  - X-Custom-Header: Value
- Cookies
  - Cookie: debug=0

Parameters Object

```
/comments:
  get:
    tags:
      - Comments
    summary: Return comments
    description: Return comments on a post with postId={id}.
    parameters:
      - in: query
        name: postId
        schema:
          type: string
        description: The postId which we want the comments
```

Request Object

- Permite describir la estructura de los mensajes de entrada
  - Permite indicar estructura del objeto
    - Schema
  - Tipo de mensaje
    - Ej: Application/json
- Se utiliza el elemento *requestBody*

Request Object

```yaml
paths:
  /posts:
      post:
        tags:
          - Posts
        requestBody:
          required: true
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Post'
```

Response Object

- Permite describir cómo es la respuesta
- Cada operación puede tener múltiples códigos HTTP de respuesta
  - Ej: 200, 404, 403, 500
- Puede tener la estructura de la respuesta y ejemplos de mensajes

Response Object

```yaml
paths:
  /posts:
      post:
        tags:
          - Posts
        requestBody:
          required: true
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Post'
        responses:
          '201':
            description: The request has succeeded and a new resource has been created as a result.
            content:
              application/json:
                schema:
                  type: string
```

Schema Object

- Permite definir estructuras de datos para definir documentar los mensajes de entrada y salida de las APIs

- Se utiliza el elemento component seguido de schemas

Schema Object

```yaml
Post:
  type: object
  properties:
    id:
      type: string
      description: "The post id."
    userId:
      type: string
      description: "The post userId."
    title:
      type: string
      description: "The title of the post"
    body:
      type: string
      description: "The body content of the post"
  required:
    - id
    - userId
    - title
    - body
```

```yaml
Posts:
  description: "An array of post objects."
  type: array
  items:
    $ref: '#/components/schemas/Post'
  example: [
        {
          "userId": 1,
          "id": 1,
          "title": "First post",
          "body": "This is the first post body."
        },
        {
          "userId": 1,
          "id": 2,
          "title": "Second post",
          "body": "The second post content."
        },
        {
          "userId": 1,
          "id": 3,
          "title": "Another post",
          "body": "Other post content"
        }
  ]
```

Dashboard de APIs

Laboratorio de Integracion de Sistemas

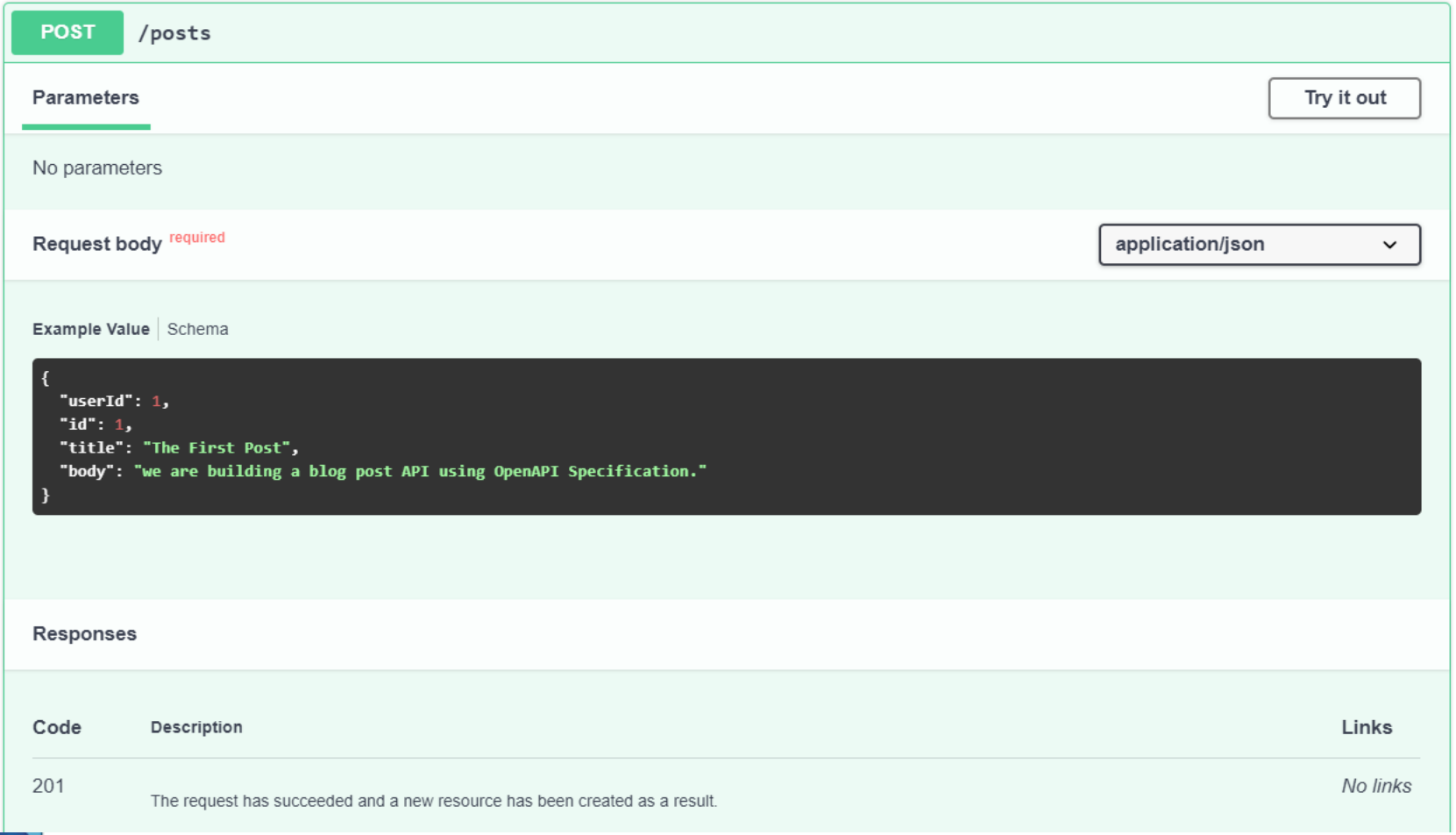
Posts

GET	`/posts`	Return all the posts
POST	`/posts`	
GET	`/posts/{id}`	Return a post
PUT	`/posts/{id}`	Update a post
PATCH	`/posts/{id}`	Update a post
DELETE	`/posts/{id}`	Deletes a post

Comments

GET	`/comments`	Return comments

Dashboard de APIs



Algunas referencias

- La especificación:
  - https://swagger.io/docs/specification/about/
- Swagger Editor
  - https://editor.swagger.io/
- Ejemplo Blog API
  - https://medium.com/@amirm.lavasani/restful-apis-tutorial-of-openapi-specification-eeada0e3901d

Resumen

- Introducción a las Web APIs o servicios RESTfull
- Mecanismos para proteger nuestras APIs
  - OAuth 2.0 + JWT
- Mecanismos para documentar nuestras APIs
  - Open API Specification

Fin

LINS

Laboratorio de Integracion de Sistemas