Proyecto de Software

Temario

- Introducción a API REST
- Fundamentos de HTTP
- Verbos HTTP y ejemplos
- Consumo de APIs públicas
- Buenas prácticas y errores comunes
- Ejemplo práctico con Flask
- Cierre y referencias

Servicios web

- Un **servicio web** = tecnología que usa **protocolos** y **estándares** para intercambiar datos.
- Carcterísticas
  - **Interoperabilidad:** Dos sistemas diferentes (ej: Java y Python, Windows y Linux) pueden hablar entre sí.
  - **Protocolos estándar:** Usan tecnologías abiertas como HTTP, XML, JSON, SOAP, WSDL, REST.
  - **Comunicación máquina a máquina:** No están pensados para humanos, sino para que un software consuma datos de otro.
  - **Reutilización:** Se expone la lógica de negocio de una app como “servicio” para que otros puedan usarlo.

Servicios web

- Tipos de Web Services
  - SOAP (Simple Object Access Protocol)
    - Basado en XML.
    - Usa WSDL para describir la API.
    - Muy formal, con mucho protocolo.
    - Ejemplo: sistemas bancarios, gubernamentales.

Servicios web

- Tipos de Web Services
  - REST (Representational State Transfer)
    - Usa HTTP de manera natural (GET, POST, PUT, DELETE).
    - Devuelve JSON, XML o incluso HTML.
    - Más simple, más usado hoy.
    - Ejemplo: APIs de Google, Twitter, Mercado Libre.

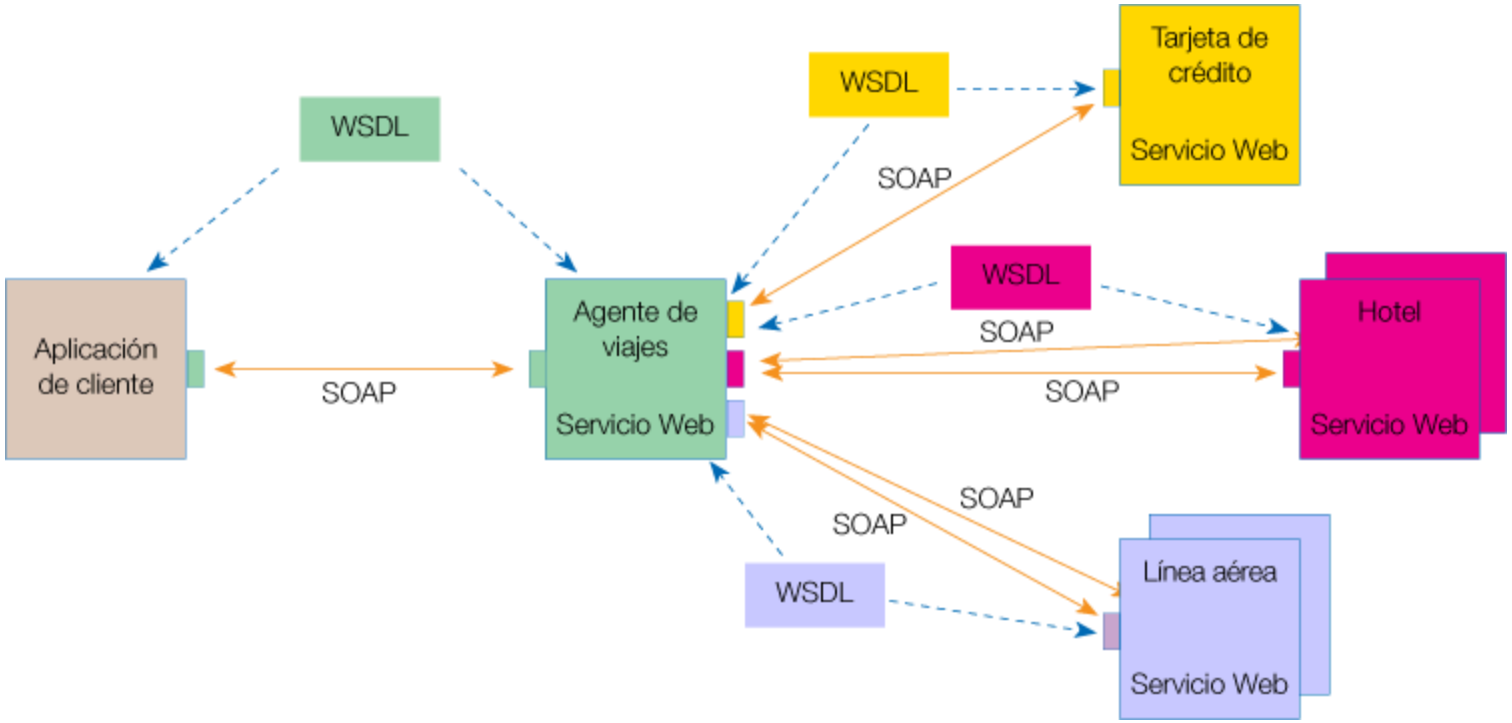
Servicios web

- Tipos de Web Services
  - RPC (Remote Procedure Call)
    - Más antiguo.
    - Invoca funciones remotas como si fueran locales.
    - Versiones modernas: JSON-RPC, XML-RPC.

API REST

- ¿Qué es una API REST?
- REST **no es** librería, es una **arquitectura**.

SOAP vs REST



- SOAP: complejo, verboso, usa XML.
- REST: simple, usa HTTP estándar.
- Pregunta: ¿qué prefieren hoy las APIs de Google, Facebook o Twitter?

SOAP Ejemplo

xml

```xml
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetIssueResponse xmlns="http://example.com/issues">
      <Issue>
        <id>1</id>
        <title>Título modificado</title>
        <description>Nueva descripción AAA</description>
        <created_at>2025-09-21T20:13:17</created_at>
        <updated_at>2025-09-22T18:23:10</updated_at>
        <user>aaaaAAAAAaaa</user>
        <status>
          <id>1</id>
          <name>mmmmmm</name>
          <description>mmmmmmm</description>
        </status>
        <status_id>1</status_id>
        <type>
          <id>1</id>
          <name>fff</name>
          <description>fff</description>
        </type>
        <type_id>1</type_id>
      </Issue>
```

SOAP

WSDL

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:xs="http://www.w3.org/2001/XMLSchema"
             targetNamespace="http://example.com/issues">

  <!-- Tipos (XSD) -->
  <types>
    <xs:schema targetNamespace="http://example.com/issues">
      <xs:complexType name="Status">
        <xs:sequence>
          <xs:element name="id" type="xs:int"/>
          <xs:element name="name" type="xs:string"/>
          <xs:element name="description" type="xs:string"/>
        </xs:sequence>
      </xs:complexType>

      <xs:complexType name="Type">
        <xs:sequence>
          <xs:element name="id" type="xs:int"/>
          <xs:element name="name" type="xs:string"/>
          <xs:element name="description" type="xs:string"/>
        </xs:sequence>
      </xs:complexType>

      <xs:complexType name="Issue">
        <xs:sequence>
          <xs:element name="id" type="xs:int"/>
          <xs:element name="title" type="xs:string"/>
          <xs:element name="description" type="xs:string"/>
          <xs:element name="created_at" type="xs:dateTime"/>
          <xs:element name="updated_at" type="xs:dateTime"/>
          <xs:element name="user" type="xs:string"/>
          <xs:element name="status" type="tns:Status"/>
          <xs:element name="status_id" type="xs:int"/>
          <xs:element name="type" type="tns:Type"/>
          <xs:element name="type_id" type="xs:int"/>
```

SOAP

WSDL - Continuación

```
<!-- Mensajes -->
<message name="GetIssueRequest">
  <part name="id" type="xs:int"/>
</message>

<message name="GetIssueResponse">
  <part name="issue" element="tns:Issue"/>
</message>

<!-- Operaciones -->
<portType name="IssuePortType">
  <operation name="GetIssue">
    <input message="tns:GetIssueRequest"/>
    <output message="tns:GetIssueResponse"/>
  </operation>
</portType>

<!-- Binding -->
<binding name="IssueBinding" type="tns:IssuePortType">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="GetIssue">
    <soap:operation soapAction="http://example.com/issues/GetIssue"/>
    <input><soap:body use="literal"/></input>
    <output><soap:body use="literal"/></output>
  </operation>
</binding>

<!-- Endpoint físico -->
<service name="IssueService">
  <port name="IssuePort" binding="tns:IssueBinding">
    <soap:address location="http://example.com/soap/issues"/>
```

¿Qué es REST?

- **REpresentational State Transfer**.
- Define cómo estructurar aplicaciones distribuidas usando HTTP.
- Creado por Roy Fielding (2000).
- Idea clave: **los recursos se identifican con URLs**.

Características REST

- Basado en HTTP.
- **Sin estado** (stateless).
- Orientado a **recursos**.
- Cada recurso tiene su propia URL.
- Soporta diferentes representaciones: HTML, XML, JSON, etc.

Petición HTTP

- Una petición consta de:
  - **URL** + **método** (GET, POST, PUT, DELETE).
  - **Cabeceras**.
  - **Cuerpo** (opcional).

HTTP MODEL

Client

Server

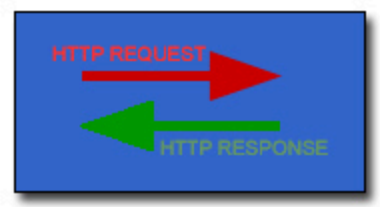
GET / HTTP/1.1

HTTP/1.1 200 OK

POST /login HTTP/1.1

HTTP/1.1 401 Unauthorized

Client



Métodos HTTP

- **GET** → obtener recurso
- **POST** → crear recurso
- **PUT** → modificar recurso
- **DELETE** → eliminar recurso

Idempotencia

- **Seguro**: no modifica datos (ej. GET).
- **Idempotente**: mismo resultado al repetirlo (ej. DELETE).
- **POST no es idempotente**.

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

Accediendo a recursos

- Ejemplo: feriados 2025
- ```
  curl -X GET https://api.mercadolibre.com/categories/MLA5725
  ```
- Otros ejemplos:
  - API REST de Mercado Libre: https://developers.mercadolibre.com.ar/
  - Google Translate (es necesario API_KEY): https://developers.google.com/apis-explorer/#p/translate/v2/
  - El clima en OpenWeatherMap (es necesario API_KEY): clima en La Plata
  - Algunas API rest públicas: en desarrolloweb.com
  ```

Ventajas / Desventajas

- Separación cliente/servidor.
- Simplicidad.
- Seguridad.
- Uso de estándares.
- Escalabilidad.
- Cambio de esquema: usando REST podemos tener varios servidores donde unos no saben que los otros existen.
- +Info: https://desarrolloweb.com/articulos/ventajas-inconvenientes-apirest-desarrollo.html

Generando API REST

- A mano.... o,
- Muchos frameworks que facilitan el desarrollo:
  - Django: https://www.django-rest-framework.org/tutorial/quickstart/
  - Flask: https://flask-restful.readthedocs.io/en/latest/quickstart.html
  - FastAPI: https://fastapi.tiangolo.com/

Algunas librerias

- **Marshmallow**
  - sirve como una biblioteca de serialización y deserialización de objetos
  - convierte datos complejos, como objetos de Python, en formatos más sencillos, como JSON, y viceversa.

Algunas librerias

- **jsonify**
  - Serialización a JSON: Convierte los datos que le pasas (como diccionarios, listas u otros tipos de datos compatibles) en una cadena JSON.
  - Encabezados de respuesta: Establece el encabezado Content-Type de la respuesta a application/json, lo que le indica al cliente (como un navegador o una aplicación) que el contenido es un archivo JSON.
  - Codificación: Asegura que la respuesta esté codificada en UTF-8, el estándar para datos JSON.

Lo que sigue ...

- Definir una API en nuestra aplicación utilizando **marshmallow** y **jsonify**.

Ejemplo

```
# src/schemas.py
from marshmallow import fields
from src.core.db import db
from src.core.issues import Issue, IssueType, IssueStatus
from marshmallow_sqlalchemy import SQLAlchemyAutoSchema, auto_field


class IssueTypeSchema(SQLAlchemyAutoSchema):
    class Meta:
        model = IssueType
        load_instance = True
        include_relationships = True
        sqla_session = db.session

    id = auto_field()
    name = auto_field()
    description = auto_field()
    issues = fields.Nested("IssueSchema", many=True, exclude=("type", "status"))


class IssueStatusSchema(SQLAlchemyAutoSchema):
    class Meta:
        model = IssueStatus
        load_instance = True
        include_relationships = True
        sqla_session = db.session

    id = auto_field()
    name = auto_field()
    description = auto_field()
    issues = fields.Nested("IssueSchema", many=True, exclude=("type", "status"))
```

Ejemplo - Continuación

```
# src/schemas.py

class IssueSchema(SQLAlchemyAutoSchema):
    class Meta:
        model = Issue
        load_instance = True
        include_relationships = True
        sqla_session = db.session

    id = auto_field()
    user = auto_field()
    title = auto_field()
    description = auto_field()
    type_id = auto_field()
    type = fields.Nested(IssueTypeSchema, exclude=("issues",))
    status_id = auto_field()
    status = fields.Nested(IssueStatusSchema, exclude=("issues",))
    updated_at = auto_field()
    created_at = auto_field()
```

Ejemplo - Continuación

```
# src/web/controllers/api_v1.py

from flask import Blueprint, jsonify, request
from src.schemas import IssueSchema, IssueTypeSchema, IssueStatusSchema
from src.core.issues import Issue
from marshmallow import ValidationError
from sqlalchemy.orm import joinedload
from src.core.db import db

# Crea un Blueprint específico para la API
api_v1_blueprint = Blueprint("api_v1", __name__, url_prefix="/apiv1")

# Inicializa los esquemas
issue_schema = IssueSchema()
issues_schema = IssueSchema(many=True)
issue_type_schema = IssueTypeSchema()
issue_types_schema = IssueTypeSchema(many=True)
issue_status_schema = IssueStatusSchema()
issue_statuses_schema = IssueStatusSchema(many=True)
```

Ejemplo - Continuación

```
# src/web/controllers/api_v1.py

.....


@api_v1_blueprint.route("/issues", methods=["GET", "POST"])
def issues_api():
    if request.method == "POST":
        try:
            new_issue = issue_schema.load(request.form)
            db.session.add(new_issue)
            db.session.commit()
            return jsonify(issue_schema.dump(new_issue)), 201
        except ValidationError as err:
            return jsonify(err.messages), 400

    elif request.method == "GET":
        issues = Issue.query.options(
            joinedload(Issue.type), joinedload(Issue.status)
        ).all()
        return jsonify(issues_schema.dump(issues))

@api_v1_blueprint.route("/issues/<int:issue_id>", methods=["GET"])
def get_issue(issue_id):
    issue = Issue.query.options(
        joinedload(Issue.type), joinedload(Issue.status)
    ).get_or_404(issue_id)
    return jsonify(issue_schema.dump(issue))
```

Ejemplo - Continuación

```
# src/web/controllers/api_v1.py

.....


@api_v1_blueprint.route("/issues/<int:issue_id>", methods=["PUT"])
def update_issue(issue_id):
    try:
        # 1. Busca el Issue a actualizar.
        issue = Issue.query.get_or_404(issue_id)

        # 2. Deserializa y valida los datos recibidos.
        #    'partial=True' permite actualizar solo algunos campos, sin requerir todos.
        updated_issue_data = issue_schema.load(
            request.form, instance=issue, partial=True
        )

        # 3. La instancia 'issue' ya fue actualizada por Marshmallow,
        #    solo necesitas confirmar los cambios en la base de datos.
        db.session.commit()

        # 4. Devuelve el objeto actualizado serializado.
        return jsonify(issue_schema.dump(updated_issue_data)), 200

    except ValidationError as err:
        return jsonify(err.messages), 400


@api_v1_blueprint.route("/issues/<int:issue_id>", methods=["DELETE"])
def delete_issue(issue_id):
    # 1. Busca el Issue a eliminar.
    issue = Issue.query.get_or_404(issue_id)

    # 2. Elimina el objeto de la sesión.
    db.session.delete(issue)

    # 3. Confirma la eliminación en la base de datos.
    db.session.commit()

    # 4. Devuelve una respuesta vacía o un mensaje de éxito con un código 204.
    return "", 204
```

Ejemplo - Continuación

```
# src/web/__init__.py

....
from src.web.controllers.api_v1 import api_v1_blueprint

....
def create_app():
    ....
    app.register_blueprint(api_v1_blueprint)
    ....
```

Referencias Rest

- http://martinfowler.com/articles/richardsonMaturityModel.html
- http://restcookbook.com/Miscellaneous/richardsonmaturitymodel/
- http://www.restapitutorial.com/lessons/whatisrest.html
- http://asiermarques.com/2013/conceptos-sobre-apis-rest/
- http://rest.elkstein.org/
- http://restfulwebapis.org/rws.html
- https://restfulapi.net/
- https://www.paradigmadigital.com/dev/introduccion-django-rest-framework/
- https://flask-restful.readthedocs.io/en/latest/

Fin