ICT INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Python.5-6
UNITÀ: Py.5-6.1

REST & Flask

2024-2026


# Rest & Flask

ICT INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Python.5-6
UNITÀ: Py.5-6.1.1

REST & Flask
Introduzione

2024-2026


Introduzione

# Rest & Flask

# Che cos'è REST

- Architettura *stateless* client-server
- Utilizza HTTP come protocollo di comunicazione
- Principi fondamentali:
  - **Risorse**: identificate da URI
  - **Metodi HTTP**: GET, POST, PUT, DELETE
  - Rappresentazioni: **JSON**, XML
  - **Stateless**: ogni richiesta è indipendente
  - **Cache**: risposte cacheable quando appropriato

# REST: Vantaggi

- Semplicità: utilizza standard HTTP esistenti
- Scalabilità: architettura stateless
- Flessibilità: supporta diversi formati dati
- Interoperabilità: funziona su diverse piattaforme
- Separazione: client e server indipendenti

# Risorse e URI: progettazione

- Una risorsa = un'entità del dominio (oggetti in analisi concettuale!)
- URI dovrebbero essere nomi (sostantivi), non verbi
- Utilizzare plurali per le collezioni

# URI e risorse: esempi corretti

```
GET    /libri          # Ottenere tutti i libri
GET    /libri/5        # Ottenere libro con ID 5
POST   /libri          # Creare nuovo libro
PUT    /libri/5        # Aggiornare libro ID 5
DELETE /libri/5        # Eliminare libro ID 5
```

# URI e risorse: esempi scorretti

```
GET    /ottieniLibri

POST   /creaLibro

DELETE /cancellaLibro/5
```

# REST: Metodi HTTP e operazioni CRUD

| Metodo HTTP | Operazione CRUD | Descrizione | Idempotente? |
|---|---|---|---|
| GET | Read | Recupera risorse | Sì |
| POST | Create | Crea nuove risorse | No |
| PUT | Update | Sostituisce completamente | Sì |
| PATCH | Update | Modifica parziale | No |
| DELETE | Delete | Elimina risorsa | Sì |

# Codici di stato HTTP

**2xx - Successo:**

- `200 OK` - Richiesta completata
- `201 Created` - Risorsa creata
- `204 No Content` - Operazione completata senza contenuto

**4xx - Errori Client:**

- `400 Bad Request` - Richiesta malformata
- `401 Unauthorized` - Autenticazione richiesta
- `404 Not Found` - Risorsa non trovata
- `409 Conflict` - Conflitto con stato attuale

**5xx - Errori Server:**

- `500 Internal Server Error` - Errore generico server

# JSON come Standard per REST

## Vantaggi JSON:

- Leggibile da umani
- Supporto nativo JavaScript
- Parsing efficiente
- Struttura gerarchica

```
{
  "id": 1,
  "titolo": "1984",
  "autore": "George Orwell",
  "isbn": "978-0-452-28423-4",
  "disponibile": true,
  "data_pubblicazione": "1949-06-08",
  "generi": ["dystopian", "political fiction"]
}
```

# Esempio Pratico API REST: Gestione di una libreria

Base URL: https://api.biblioteca.it/v1

```
GET    /libri                    # Lista tutti i libri
GET    /libri?genere=fantasy     # Filtro per genere
GET    /libri/123                # Dettaglio libro
POST   /libri                    # Aggiungi libro
PUT    /libri/123                # Aggiorna libro
DELETE /libri/123                # Elimina libro

GET    /autori                   # Lista autori
GET    /autori/456/libri         # Libri di un autore
```

ICT INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Python.5-6
UNITÀ: Py.5-6.1.2

REST & Flask
Richardson Maturity Model

2024-2026


API REST: Richardson Maturity Model

# REST

# Il Richardson Maturity Model

## Livelli di Maturità REST

Il Richardson Maturity Model definisce **4 livelli** (0-3) per valutare quanto un'API sia "RESTful":

- **Livello 0**: Un singolo *endpoint* per tutto
- **Livello 1**: Risorse multiple
- **Livello 2**: Verbi HTTP + codici di stato
- **Livello 3**: **HATEOAS** (controlli *ipermedia*)

"True REST" = Livello 3, ma la maggior parte delle API si ferma al Livello 2

# Livello 0 - The Swamp of POX

POX = Plain Old XML (o JSON)

Caratteristiche:

- Un singolo endpoint (es. `/api`)
- Sempre `POST`
- Azione specificata nel payload

Problemi: Non sfrutta HTTP, difficile fare cache, poco intuitivo

Esempio:

```
POST /api
Content-Type: application/json
{
  "action": "getBook",
  "bookId": 5
}

POST /api
Content-Type: application/json
{
  "action": "createBook",
  "title": "Nuovo Libro",
  "author": "Autore"
}
```

# Livello 1: risorse multiple

```
POST /libri/5          # Ottenere libro
POST /libri            # Operazioni sulla collezione
POST /autori/3         # Operazioni su autore
```

# Livello 2: Verbi HTTP + Codici di Stato

```
GET    /libri/5        → 200 OK
POST   /libri          → 201 Created
PUT    /libri/5        → 200 OK
DELETE /libri/5        → 204 No Content
GET    /libri/999      → 404 Not Found
```

La maggior parte delle API REST moderne opera a questo livello

# Livello 3 - HATEOAS

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

Risposta con controlli ipermedia:

**Vantaggi**:

- Client dinamico e autodescrittivo
- Evoluzione API senza *breaking changes*
- Navigabilità come il web

**Svantaggi**:

- Maggiore complessità
- Payload più grandi

```
HTTP/1.1 201 Created
Content-Type: application/json
{
  "id": 5,
  "titolo": "1984",
  "autore": "George Orwell",
  "_links": {
    "self": {"href": "/libri/5"},
    "edit": {"href": "/libri/5", "method": "PUT"},
    "delete": {"href": "/libri/5", "method": "DELETE"},
    "author": {"href": "/autori/42"},
    "reviews": {"href": "/libri/5/recensioni"}
  }
}
```

ICT INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Python.5-6
UNITÀ: Py.5-6.2.1

REST & Flask

Introduzione a Flask

2024-2026


Introduzione

# Flask

## Flask - Il *Microframework* Python

- Framework web minimale e flessibile
- "Micro" non significa limitato, ma essenziale
- Filosofia: fornire il *core*, estendibile con plugin
- Ideale per API REST, microservizi, app piccole-medie

**Caratteristiche**:

- *Routing* flessibile
- Template engine Jinja2
- Server di sviluppo integrato
- Facilmente testabile
- Ampio ecosistema di estensioni

# Flask: Installazione e setup

```
# Creare ambiente virtuale
python -m venv venv
source venv/bin/activate    # Linux/Mac
venv\Scripts\activate       # Windows

# Installare Flask
pip install flask flask-restful

# Per il database
pip install flask-sqlalchemy psycopg2-binary

# Per OAuth2
pip install authlib

# Verifica installazione
python -c "import flask; print(flask.__version__)"
```

# Prima applicazione in Flask

```
from flask import Flask
app = Flask(__name__) # Creare istanza Flask

# Definire route
@app.route('/')
def home():
    return 'Ciao da Flask!'

@app.route('/api/status')
def status():
    return {'status': 'OK', 'message': 'API funzionante'}

if __name__ == '__main__':
    app.run(debug=True)
```

**Esecuzione**

```
> python app.py
# Server: http://127.0.0.1:5000
```

# Routing in Flask

```
from flask import Flask, jsonify,
request


app = Flask(__name__)


# Route statica
@app.route('/info')
def info():
    return {'app': 'Libreria API',
'versione': '1.0'}


# Route con parametri
@app.route('/libri/<int:libro_id>')
def get_libro(libro_id):
    return {'id': libro_id,
        'title': f'Libro {libro_id}'}
```

```
# Route con query parameters
@app.route('/ricerca')
def ricerca():
    termine = request.args.get('q', '')
    return {'query': termine,
'risultati': []}


# Metodi HTTP multipli
@app.route('/libri', methods=['GET',
'POST'])
def gestisci_libri():
    if request.method == 'GET':
        return {'libri': []}
    elif request.method == 'POST':
        return {'messaggio': 'Libro
            creato'}, 201
```

# Request e response JSON

```
from flask import Flask, request, jsonify

app = Flask(__name__)

# Lista libri (simulazione database)
libri = [
    {'id': 1, 'titolo': '1984', 'autore': 'George Orwell'},
    {'id': 2, 'titolo': 'Il nome della rosa', 'autore': 'Umberto Eco'}
]

@app.route('/libri', methods=['GET'])
def get_libri():
    return jsonify({'libri': libri, 'totale': len(libri)})
```

```
@app.route('/libri', methods=['POST'])
def add_libro():
    # Ottenere dati JSON dalla request
    dati = request.get_json()

    if not dati or 'titolo' not in dati:
        return {'errore': 'Titolo obbligatorio'}, 400

    nuovo_libro = {
        'id': len(libri) + 1,
        'titolo': dati['titolo'],
        'autore': dati.get('autore', 'Sconosciuto')
    }

    libri.append(nuovo_libro)
    return jsonify(nuovo_libro), 201
```

# Gestione degli errori

```
from flask import Flask, jsonify

app = Flask(__name__)

# Error handler personalizzato
@app.errorhandler(404)
def not_found(error):
    return jsonify({
        'errore': 'Risorsa non trovata',
        'codice': 404
    }), 404

@app.errorhandler(400)
def bad_request(error):
    return jsonify({
        'errore': 'Richiesta non valida',
        'codice': 400
    }), 400
```

```
@app.errorhandler(500)
def internal_error(error):
    return jsonify({
        'errore': 'Errore interno del server',
        'codice': 500
    }), 500

# Gestione errori in una route
@app.route('/libri/<int:libro_id>')
def get_libro(libro_id):
    libro = find_libro_by_id(libro_id)

    if not libro:
        return jsonify({'errore': f'Libro {libro_id} non trovato'}), 404

    return jsonify(libro)
```

ICT INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Python.5-6
UNITÀ: Py.5-6.2.2

REST & Flask
Flask-RESTful

2024-2026


# Flask-RESTful

# Introduzione a Flask-RESTful

Flask base è ottimo, ma Flask-RESTful offre:

- Struttura orientata alle risorse (classi ***Resource***)
- Parsing automatico degli argomenti
- Serializzazione avanzata
- Gestione errori integrata
- Decoratori per validazione

Installazione:

```
> pip install flask-restful
```

# Le *Resource*

```
from flask import Flask
from flask_restful import Api, Resource

app = Flask(__name__)
api = Api(app)

class LibroResource(Resource):
    def get(self, libro_id):
        # GET /libri/123
        return {'id': libro_id, 'titolo': 'Esempio'}

    def put(self, libro_id):
        # PUT /libri/123
        return {'messaggio': f'Libro {libro_id} aggiornato'}

    def delete(self, libro_id):
        # DELETE /libri/123
        return '', 204
```

```
class LibriResource(Resource):
    def get(self):
        # GET /libri
        return {'libri': []}

    def post(self):
        # POST /libri
        return {'messaggio': 'Libro creato'}, 201
```

```
# Registrare resource
api.add_resource(LibriResource, '/libri')

api.add_resource(LibroResource, '/libri/<int:libro_id>')
```

# Parsing degli argomenti

```
from flask_restful import Resource, reqparse
class LibriResource(Resource):
    def __init__(self):
        self.parser = reqparse.RequestParser()
        self.parser.add_argument('titolo', type=str, required=True,
                                 help='Titolo obbligatorio')
        self.parser.add_argument('autore', type=str, required=True)
        self.parser.add_argument('anno', type=int, location='args')
# Query param
        self.parser.add_argument('genere', type=str,
choices=['fiction', 'non-fiction'])


    def post(self):
        args = self.parser.parse_args()
        nuovo_libro = {
            'id': generate_id(),
            'titolo': args['titolo'],
            'autore': args['autore'],
            'anno': args.get('anno'),
            'genere': args.get('genere', 'fiction')
        }
        # Salvare in database...
        return nuovo_libro, 201
```

```
    def get(self):
        # Parse solo query parameters
        parser = reqparse.RequestParser()
        parser.add_argument('limit', type=int,
            location='args', default=10)
        parser.add_argument('offset',
type=int,
            location='args', default=0)
        args = parser.parse_args()


        # Paginazione...
        return {'libri': [], 'limit':
args['limit']}
```

# Marshalling e serializzazione: controllo output con *Fields*

```python
from flask_restful import Resource,fields, marshal_with, abort


# Definire struttura output
libro_fields = {
    'id': fields.Integer,
    'titolo': fields.String,
    'autore': fields.String,
    'isbn': fields.String,
    'anno_pubblicazione': fields.Integer,
    'url': fields.Url('libro_detail')  # Link automatico
}


libri_fields = {
    'libri': fields.List(fields.Nested(libro_fields)),
    'totale': fields.Integer,
    'pagina': fields.Integer
}
```

```python
class LibroResource(Resource):
    @marshal_with(libro_fields)
    def get(self, libro_id):
        libro = get_libro_from_db(libro_id)
        if not libro:
            abort(404)
        return libro


class LibriResource(Resource):
    @marshal_with(libri_fields)
    def get(self):
        parser = reqparse.RequestParser()
        parser.add_argument('page', type=int, default=1)
        args = parser.parse_args()

        libri = get_libri_paginated(args['page'])
        return {
            'libri': libri,
            'totale': count_libri(),
            'pagina': args['page']
        }
```

# Testare l'API con Postman

**1. GET tutti i libri:**

```
GET http://localhost:5000/libri
```

**2. GET libro specifico:**

```
GET http://localhost:5000/libri/1
```

**3. POST nuovo libro:**

```
POST http://localhost:5000/libri
```

Content-Type: application/json

```
{
    "titolo": "Il Signore degli Anelli",
    "autore": "J.R.R. Tolkien",
    "anno": 1954
}
```

**4. PUT aggiorna libro:**

```
PUT http://localhost:5000/libri/1
```

Content-Type: application/json

```
{
    "titolo": "1984 - Edizione Speciale",
    "anno": 1949
}
```

**5. DELETE libro:**

```
DELETE http://localhost:5000/libri/2
```

ICT INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Python.5-6
UNITÀ: Py.5-6.3.1

REST & Flask
Autenticazione

2024-2026


OAuth2

# Autenticazione

# Autenticazione: panoramica

## Metodi di Autenticazione Web

1. **Basic Authentication**:
   - Username/password in header HTTP
   - Base64 encoding (non sicuro senza HTTPS)
2. **Session-based**:
   - Cookie con session ID
   - Server mantiene *stato sessione*
3. **Token-based (JWT)**:
   - Token autocontenuto
   - *Stateless*
4. **OAuth2**:
   - Standard per autorizzazione
   - Delega accesso a terze parti

Per API REST: **Token-based e OAuth2 sono preferibili**

## OAuth2 = Open Authorization 2.0

- Standard per autorizzazione (non autenticazione!)
- Permette **accesso limitato** alle risorse
- Senza condividere credenziali (!)

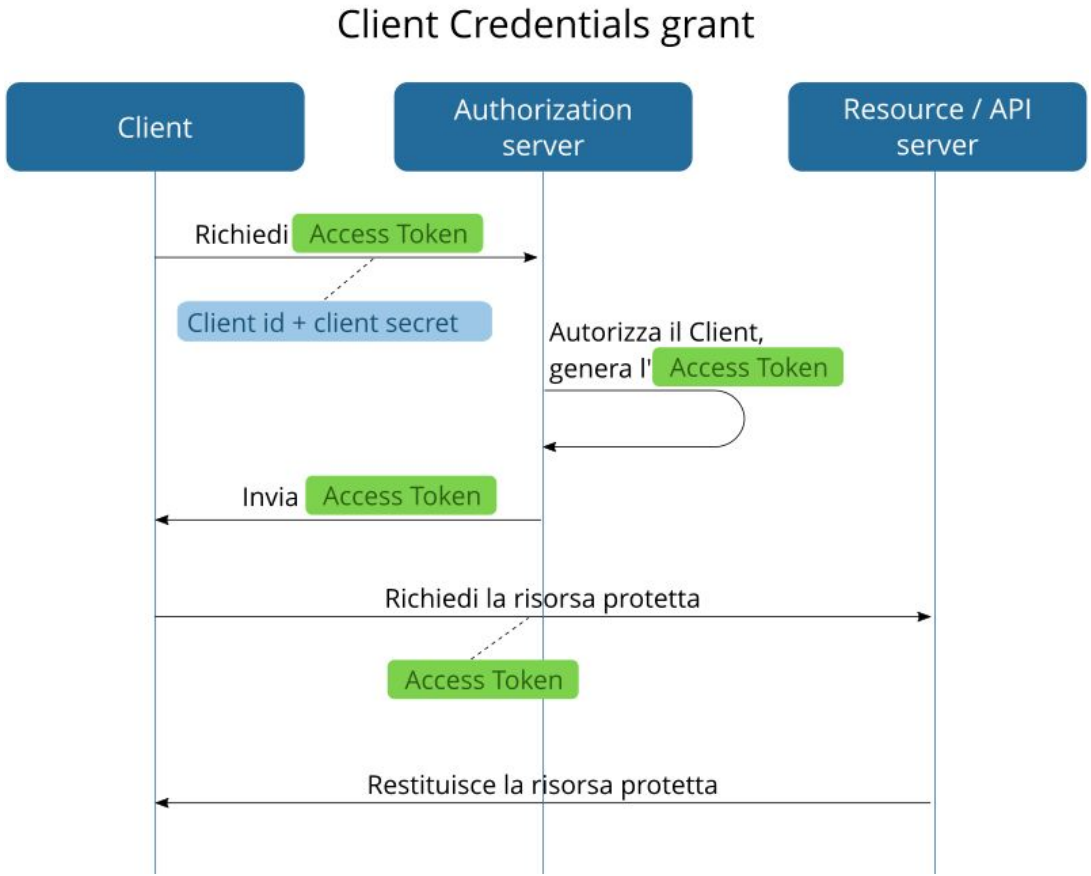
## Attori OAuth2:

- *Resource Owner*: l'utente
- *Client*: l'applicazione che vuole accesso
- *Authorization Server*: rilascia token
- *Resource Server*: API protetta

## Grant Types principali:

- Authorization Code (web app)
- Client Credentials (app-to-app)
- Resource Owner Password (legacy)
- Implicit (deprecato)

# OAuth2: flusso per App-to-App




Vantaggi:

- Semplice per comunicazione server-to-server
- No user interaction
- Sicuro per ambienti controllati

# Implementazione Server OAuth2 con Authlib (1)

```
from flask import Flask, jsonify, request
from authlib.integrations.flask_oauth2 import AuthorizationServer
from authlib.oauth2.rfc6749 import grants
import secrets

app = Flask(__name__)
app.config['SECRET_KEY'] = 'your-secret-key'

# Setup authorization server
authorization = AuthorizationServer(app)

# Client storage (in production: database)
CLIENTS = {
    'client123': {
        'client_secret': 'secret456',
        'client_name': 'My API Client',
        'grant_types': ['client_credentials'],
        'scope': 'read write'
    }
}
```

# Implementazione Server OAuth2 con Authlib (2)

```
# Token storage (in production: database/cache)
TOKENS = {}

class ClientCredentialsGrant(grants.ClientCredentialsGrant):
    def save_token(self, token, request, *args, **kwargs):
        access_token = token['access_token']
        TOKENS[access_token] = {
            'client_id': request.client.client_id,
            'scope': token.get('scope', ''),
            'expires_in': token['expires_in']
        }

def query_client(client_id):
    return CLIENTS.get(client_id)

def save_token(token, request, *args, **kwargs):
    # Implementare salvataggio token
    pass
```

# Implementazione Server OAuth2 con Authlib (3)

```
# Configurare authorization server
authorization.init_app(app, query_client=query_client, save_token=save_token)
authorization.register_grant(ClientCredentialsGrant)

@app.route('/oauth/token', methods=['POST'])
def issue_token():
    return authorization.create_token_response()

# API protetta
def require_oauth(f):
    def decorated_function(*args, **kwargs):
        auth_header = request.headers.get('Authorization')

        if not auth_header or not auth_header.startswith('Bearer '):
            return {'errore': 'Token richiesto'}, 401
        token = auth_header.split(' ')[1]

        if token not in TOKENS:
            return {'errore': 'Token non valido'}, 401
        return f(*args, **kwargs)

    return decorated_function
```

# Implementazione Server OAuth2 con Authlib (4)

```
@app.route('/api/protetta')
@require_oauth
def risorsa_protetta():
    return {'messaggio': 'Accesso autorizzato!', 'dati': 'riservati'}

if __name__ == '__main__':
    app.run(debug=True)
```

# Client OAuth2 (1)

```
import requests
import json

# 1. Ottenere access token
token_url = 'http://localhost:5000/oauth/token'
token_data = {
    'grant_type': 'client_credentials',
    'client_id': 'client123',
    'client_secret': 'secret456'
}

# Richiesta token
response = requests.post(token_url, data=token_data)
token_info = response.json()

print("Token ottenuto:", token_info)
# Output: {'access_token': 'xxx', 'token_type': 'Bearer', 'expires_in': 3600}
```

# Client OAuth2 (2)

```
# 2. Usare token per accedere all'API
access_token = token_info['access_token']
api_url = 'http://localhost:5000/api/protetta'

headers = {
    'Authorization': f'Bearer {access_token}',
    'Content-Type': 'application/json'
}

# Chiamata API protetta
api_response = requests.get(api_url, headers=headers)
print("Risposta API:", api_response.json())

# Test senza token (deve fallire)
response_no_auth = requests.get(api_url)
print("Senza auth:", response_no_auth.status_code)  # 401
```

# Integrazione con Google OAuth2

```python
from authlib.integrations.flask_client import OAuth
from flask import Flask, redirect, url_for, session
app = Flask(__name__)
app.secret_key = 'your-secret-key'
oauth = OAuth(app)
# Configurare Google OAuth
google = oauth.register(
    name='google',
    client_id='your-google-client-id',
    client_secret='your-google-client-secret',

server_metadata_url='https://accounts.google.com/.well-known/openid-configuration',
    client_kwargs={
        'scope': 'openid email profile'
    }
)
@app.route('/')
def home():
    if 'user' in session:
        user = session['user']
        return f'Ciao {user["name"]}! Email: {user["email"]}'
    return '<a href="/login">Login con Google</a>'
```

```python
@app.route('/login')
def login():
    redirect_uri = url_for('callback', _external=True)
    return google.authorize_redirect(redirect_uri)

@app.route('/callback')
def callback():
    token = google.authorize_access_token()
    user_info = token['userinfo']

    session['user'] = {
        'name': user_info['name'],
        'email': user_info['email'],
        'picture': user_info['picture']
    }

    return redirect('/')

@app.route('/logout')
def logout():
    session.pop('user', None)
    return redirect('/')

# API protetta che richiede login Google
@app.route('/api/profile')
def profile():
    if 'user' not in session:
        return {'errore': 'Login richiesto'}, 401

    return {'profilo': session['user']}
```

ICT INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Python.5-6
UNITÀ: Py.5-6.4.1

REST & Flask

Integrazione con basi di dati

2024-2026


Introduzione

# Flask+PostgreSQL

# Compatibilità

Flask + PostgreSQL:

- Supporto nativo con **psycopg2**
- Integrazione eccellente con **SQLAlchemy**
- Ideale per applicazioni production

```
> pip install psycopg2-binary flask-sqlalchemy
```

# SQLAlchemy con PostgreSQL: creazione database

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from datetime import datetime
app = Flask(__name__)
# Configurazione database
app.config['SQLALCHEMY_DATABASE_URI'] =
'postgresql://libreria_user:password123@localhost/libreria_db
'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)


# Modello Libro (tabella)
class Libro(db.Model):
    __tablename__ = 'libri'
    id = db.Column(db.Integer, primary_key=True)
    titolo = db.Column(db.String(200), nullable=False)
    autore = db.Column(db.String(100), nullable=False)
    isbn = db.Column(db.String(20), unique=True)
    anno_pubblicazione = db.Column(db.Integer)
    genere = db.Column(db.String(50))
    disponibile = db.Column(db.Boolean, default=True)
    data_creazione = db.Column(db.DateTime,
default=datetime.utcnow)
    def to_dict(self):
        return {
            'id': self.id, 'titolo': self.titolo,  'autore': self.autore,
'isbn': self.isbn,
            'anno_pubblicazione': self.anno_pubblicazione',
             'genere': self.genere,  'disponibile': self.disponibile,
              'data_creazione': self.data_creazione.isoformat()    }


# Modello Autore (tabella)
class Autore(db.Model):
    __tablename__ = 'autori'
    id = db.Column(db.Integer, primary_key=True)
    nome = db.Column(db.String(100), nullable=False)
    cognome = db.Column(db.String(100), nullable=False)
    biografia = db.Column(db.Text)

    # Relazione 1..1 - 0..*
    libri = db.relationship('Libro', backref='autore_obj', lazy=True)


# Creare tabelle
with app.app_context():
    db.create_all()
```

# CRUD con SQLAlchemy (1)

```
from flask import Flask, request, jsonify
from flask_restful import Api, Resource
from sqlalchemy.exc import IntegrityError
app = Flask(__name__)
# ... configurazione db come sopra ...

class LibriResource(Resource):
    def get(self):
        # Query con filtri opzionali
        autore_filter = request.args.get('autore')
        genere_filter = request.args.get('genere')

        query = Libro.query
        if autore_filter:
            query = query.filter(Libro.autore.ilike(f'%{autore_filter}%'))
        if genere_filter:
            query = query.filter(Libro.genere == genere_filter)

        libri = query.all()
        return {
            'libri': [libro.to_dict() for libro in libri],
            'totale': len(libri)
        }
```

```
    def post(self):
        dati = request.get_json()
        try:
            nuovo_libro = Libro(
                titolo=dati['titolo'],
                autore=dati['autore'],
                isbn=dati.get('isbn'),
anno_pubblicazione=dati.get('anno_pubblicazione'),
                genere=dati.get('genere')
            )
            db.session.add(nuovo_libro)
            db.session.commit()
            return nuovo_libro.to_dict(), 201
        except IntegrityError:
            db.session.rollback()
            return {'errore': 'ISBN già esistente'},
409
        except Exception as e:
            db.session.rollback()
            return {'errore': str(e)}, 500
```

# CRUD con SQLAlchemy (2)

```
class LibroResource(Resource):


    def put(self, libro_id):
        libro = Libro.query.get_or_404(libro_id)
        dati = request.get_json()
        try:
            libro.titolo = dati.get('titolo', libro.titolo)
            libro.autore = dati.get('autore', libro.autore)
            libro.isbn = dati.get('isbn', libro.isbn)
            libro.anno_pubblicazione =
dati.get('anno_pubblicazione', libro.anno_pubblicazione)
            libro.genere = dati.get('genere', libro.genere)
            libro.disponibile = dati.get('disponibile',
libro.disponibile)
            db.session.commit()
            return libro.to_dict()
        except IntegrityError:
            db.session.rollback()
            return {'errore': 'ISBN già esistente'}, 409
```

```
def get(self, libro_id):
        libro = Libro.query.get_or_404(libro_id)
        return libro.to_dict()

def delete(self, libro_id):
        libro = Libro.query.get_or_404(libro_id)
        db.session.delete(libro)
        db.session.commit()
        return '', 204



# Creazione app e aggiunta risorse
api = Api(app)
api.add_resource(LibriResource, '/libri')
api.add_resource(LibroResource,
      '/libri/<int:libro_id>')
```

# Psycopg2: Accesso diretto a PostgreSQL (senza ORM)

```
import psycopg2
from psycopg2.extras import RealDictCursor
from flask import Flask, jsonify, request

app = Flask(__name__)

# Configurazione connessione
DB_CONFIG = {
    'host': 'localhost',
    'database': 'libreria_db',
    'user': 'libreria_user',
    'password': 'password123'
}

def get_db_connection():
    return psycopg2.connect(**DB_CONFIG)
```

# Psycopg2: Accesso diretto a PostgreSQL (2)

```
@app.route('/libri', methods=['GET'])
def get_libri():
    conn = get_db_connection()
    try:
        with conn.cursor(cursor_factory=RealDictCursor) as cur:
            # Query con parametri sicuri
            autore = request.args.get('autore')
            if autore:
                cur.execute(
                    "SELECT * FROM libri WHERE autore ILIKE %s ORDER BY titolo", (f'%{autore}%',) )
            else:
                cur.execute("SELECT * FROM libri ORDER BY titolo")

            libri = cur.fetchall()
            # Convertire Row objects in dict
            result = [dict(libro) for libro in libri]

        return jsonify({'libri': result, 'totale': len(result)})
    except psycopg2.Error as e:
        return jsonify({'errore': str(e)}), 500
    finally:
        conn.close()
```

# Psycopg2: Accesso diretto a PostgreSQL (3)

```
@app.route('/libri', methods=['POST'])
def add_libro():
    dati = request.get_json()
    conn = get_db_connection()

    try:
        with conn.cursor() as cur:
            cur.execute("""
                INSERT INTO libri (titolo, autore, isbn,
anno_pubblicazione, genere)
                VALUES (%s, %s, %s, %s, %s)
                RETURNING id, titolo, autore, isbn,
anno_pubblicazione, genere, disponibile, data_creazione
            """, (
                dati['titolo'],
                dati['autore'],
                dati.get('isbn'),
                dati.get('anno_pubblicazione'),
                dati.get('genere')
            ))
        nuovo_libro =  cur.fetchone()
        conn.commit()
            return jsonify({
                'id': nuovo_libro[0],
                'titolo': nuovo_libro[1],
                'autore': nuovo_libro[2],
                'isbn': nuovo_libro[3],
                'anno_pubblicazione': nuovo_libro[4],
                'genere': nuovo_libro[5],
                'disponibile': nuovo_libro[6],
                'data_creazione':
nuovo_libro[7].isoformat()
            }), 201

    except psycopg2.IntegrityError:
        conn.rollback()
        return jsonify({'errore': 'ISBN già esistente'}),
409
    except psycopg2.Error as e:
        conn.rollback()
        return jsonify({'errore': str(e)}), 500
    finally:
        conn.close()
```

```
# Utility per gestione connessioni
class DatabaseManager:
    def __init__(self, config):
        self.config = config

    def execute_query(self, query, params=None):
        conn = psycopg2.connect(**self.config)
        try:
            with conn.cursor(cursor_factory=RealDictCursor) as cur:
                cur.execute(query, params)
                if query.strip().upper().startswith('SELECT'):
                    return cur.fetchall()
                conn.commit()
                return cur.rowcount
        finally:
            conn.close()

db_manager = DatabaseManager(DB_CONFIG)
```

ICT INFORMATION AND COMMUNICATIONS TECHNOLOGY ACADEMY

MODULO: Python.5-6
UNITÀ: Py.5-6.4.1

REST & Flask

Flask: best practices

2024-2026


Conclusione

# Flask

# Organizzazione del codice professionale

```
├── app/
│   ├── __init__.py          # Factory pattern
│   ├── config.py            # Configurazioni
│   ├── models/
│   │   ├── __init__.py
│   │   ├── libro.py
│   │   └── utente.py
│   ├── resources/           # REST resources
│   │   ├── __init__.py
│   │   ├── libri.py
│   │   └── auth.py
│   ├── services/            # Business logic
│   │   ├── __init__.py
│   │   └── libro_service.py
│   └── utils/
│       ├── __init__.py
│       └── decorators.py
├── tests/
│   ├── __init__.py
│   ├── test_libri.py
│   └── test_auth.py
├── requirements.txt
├── run.py                   # Entry point
└── docker-compose.yml
```

# app/__init__.py : Application Factory

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_restful import Api

db = SQLAlchemy()
api = Api()

def create_app(config_name='development'):
    app = Flask(__name__)

    # Caricare configurazione
    app.config.from_object(f'app.config.{config_name.title()}Config')

    # Inizializzare estensioni
    db.init_app(app)
    api.init_app(app)

    # Registrare blueprints/resources
    from app.resources.libri import LibriResource, LibroResource
    api.add_resource(LibriResource, '/libri')
    api.add_resource(LibroResource, '/libri/<int:libro_id>')

    return app
```

# Validazione dell'input

```python
from marshmallow import Schema, fields, ValidationError

class LibroSchema(Schema):
    titolo = fields.Str(required=True,
     validate=lambda x: len(x) > 0 and len(x) <= 200)

    autore = fields.Str(required=True,
     validate=lambda x: len(x) <= 100)

    isbn = fields.Str(
     validate=lambda x: len(x) == 13 if x else True)

    anno_pubblicazione = fields.Int(
     validate=lambda x: 1000 <= x <= 2030 if x else True)

    genere = fields.Str(
     validate=lambda x: x in ['fiction', 'non-fiction',
     'science', 'history']
                    if x else True)
```

```python
def validate_libro_data(f):
  def decorated_function(*args, **kwargs):
      schema = LibroSchema()
      try:
        # Validare dati input
        validated_data =
            schema.load(request.get_json())
        request.validated_data =validated_data
            return f(*args, **kwargs)
      except ValidationError as err:
            return {'errori': err.messages},
400

    return decorated_function

class LibriResource(Resource):
    @validate_libro_data
    def post(self):
      # Dati già validati
        dati = request.validated_data
        # ... resto della logica
```

# Prevenzione SQL Injection

```
# ❌ MAI FARE COSÌ - Vulnerabile a SQL injection
def search_libri_unsafe(termine):
    query = f"SELECT * FROM libri WHERE titolo LIKE  '%{termine}%'"
    cursor.execute(query)  # PERICOLOSO!

# ✅ SEMPRE usare parameterized queries
def search_libri_safe(termine):
    query = "SELECT * FROM libri WHERE titolo  ILIKE %s"
    cursor.execute(query, (f'%{termine}%',))

# ✅ Con SQLAlchemy è automaticamente sicuro
def search_libri_sqlalchemy(termine):
    return Libro.query.filter(Libro.titolo.ilike(f'%{termine}%')).all()
```