SQL for the Web

Databases

Pierre Senellart

PSL UNIVERSITÉ PARIS

03 June 2026

Outline

Where we are

We now know how to:

- write HTML and CSS (lectures 10–11)
- talk to a database from Python with DB-API and SQLAlchemy (lecture 8)
- run a Flask application that turns each URL into a Python function and renders Jinja templates (lecture 12)

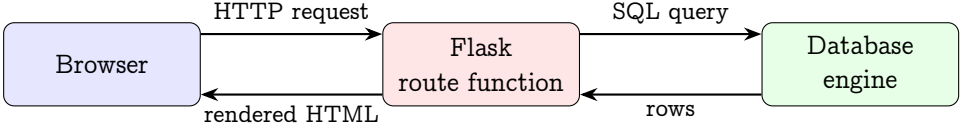
Where we are

We now know how to:

- write HTML and CSS (lectures 10–11)
- talk to a database from Python with DB-API and SQLAlchemy (lecture 8)
- run a Flask application that turns each URL into a Python function and renders Jinja templates (lecture 12)

Today: put the database behind the Flask app. Every page becomes the rendering of an SQL query result; every form becomes an INSERT, UPDATE or DELETE.

The full request cycle



- Each request opens (or reuses) a DB connection
- The route function runs one or more SQL statements
- Rows are passed to a Jinja template
- The template emits HTML that the browser displays

The Model-View-Controller pattern

MVC (Model-View-Controller) divides an interactive application into three responsibilities, so that each can change on its own. Throughout this lecture:

Model the data and the rules that govern it – the SQLAlchemy classes and the tables behind them

View what the user sees – the Jinja templates that render HTML

Controller the glue that receives a request, queries the model and chooses a view – the Flask route functions

The Model-View-Controller pattern

MVC (Model-View-Controller) divides an interactive application into three responsibilities, so that each can change on its own. Throughout this lecture:

Model the data and the rules that govern it – the SQLAlchemy classes and the tables behind them

View what the user sees – the Jinja templates that render HTML

Controller the glue that receives a request, queries the model and chooses a view – the Flask route functions

A new look touches only the views; a new query, only the model; a new page, only a controller.

Running example

A small hotel booking app. Schema:

- `room(id, capacity, price)`
- `guest(id, name, email)`
- `reservation(id, room, guest, arrival, nights)`

Pages we want to build:

- `/` list all rooms
- `/rooms/<id>` details and bookings for one room
- `/book` form to create a reservation
- `/search` find reservations by guest name
- `/reservations/<id>/edit` change a reservation
- `/reservations/<id>/cancel` delete a reservation

Outline

One connection per request

Each HTTP request runs concurrently with others; a database connection is a precious, stateful resource.

One connection per request

Each HTTP request runs concurrently with others; a database connection is a precious, stateful resource. Two pieces of bad advice we will not follow:

a single global connection — one connection serialises every request and breaks under load

a new connection per query — handshakes and TCP cost dominate; the database runs out of slots

One connection per request

Each HTTP request runs concurrently with others; a database connection is a precious, stateful resource. Two pieces of bad advice we will not follow:

a single global connection one connection serialises every request and breaks under load

a new connection per query handshakes and TCP cost dominate; the database runs out of slots

Right balance: one connection per request, obtained when first needed and closed at the end of the request. Behind the scenes, a connection pool reuses real connections across requests.

The Flask application and request contexts

Flask exposes two thread-local proxies for state that varies per request:

`flask.g` a place to stash per-request data (a DB connection, the current user...)

`flask.current_app` the active Flask application

The Flask application and request contexts

Flask exposes two thread-local proxies for state that varies per request:

`flask.g` a place to stash per-request data (a DB connection, the current user...)

`flask.current_app` the active Flask application

(Recall from lecture 12: a route function is the Python function Flask runs to handle a request, e.g. the one decorated with `@app.route("/")`.) Lifecycle hooks:

`@app.before_request` runs before each route function

`@app.teardown_appcontext` runs after each request, even if an exception was raised; the right place to close resources

Pattern with raw DB-API

Python

```python
import mysql.connector
from flask import Flask, g

app = Flask(__name__)

def get_db():
    if "db" not in g:
        g.db = mysql.connector.connect(
            host="localhost", user="hotel",
            password="secret", database="hotel")
    return g.db

@app.teardown_appcontext
def close_db(exception):
    if "db" in g:
        g.db.close()
```

Using get_db in a route

Python

```python
@app.route("/")
def index():
    with get_db().cursor(dictionary=True) as cur:
        cur.execute(
            "SELECT id, capacity, price FROM room ORDER BY id")
        rooms = cur.fetchall()
    return render_template("index.html", rooms=rooms)
```

Using get_db in a route

Python

```python
@app.route("/")
def index():
    with get_db().cursor(dictionary=True) as cur:
        cur.execute(
            "SELECT id, capacity, price FROM room ORDER BY id")
        rooms = cur.fetchall()
    return render_template("index.html", rooms=rooms)
```

In the template:

Jinja

```
<ul>
{% for r in rooms %}
  <li>Room {{ r.id }} -- {{ r.capacity }} guests
      -- {{ r.price }} EUR</li>
{% endfor %}
</ul>
```

Outline

Flask-SQLAlchemy

An official Flask extension that wraps SQLAlchemy with all the per-request plumbing already in place:

Python

```python
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config["SQLALCHEMY_DATABASE_URI"] = \
    "mysql+mysqlconnector://hotel:secret@localhost/hotel"
db = SQLAlchemy(app)
```

Now `db.session` is a scoped session bound to the current request, cleaned up automatically on teardown.

Models on top of Flask-SQLAlchemy

Python

```python
class Room(db.Model):
    id       = db.Column(db.Integer, primary_key=True)
    capacity = db.Column(db.Integer, nullable=False)
    price    = db.Column(db.Numeric(8, 2))
    reservations = db.relationship("Reservation", back_populates="room")

class Guest(db.Model):
    id    = db.Column(db.Integer, primary_key=True)
    name  = db.Column(db.String(80), nullable=False)
    email = db.Column(db.String(120))
    reservations = db.relationship("Reservation", back_populates="guest")

class Reservation(db.Model):
    id       = db.Column(db.Integer, primary_key=True)
    room_id  = db.Column(db.ForeignKey("room.id"), nullable=False)
    guest_id = db.Column(db.ForeignKey("guest.id"), nullable=False)
    arrival  = db.Column(db.Date, nullable=False)
    nights   = db.Column(db.Integer, nullable=False)
    room     = db.relationship("Room", back_populates="reservations")
    guest    = db.relationship("Guest", back_populates="reservations")
```

Sessions in the database, as promised

Lecture 12 recommended keeping the HTTP session server-side rather than in a client cookie. With a database now wired in, that is a two-line configuration: `Flask-Session` can store sessions through the very same SQLAlchemy connection.

Python

```python
from flask_session import Session

app.secret_key = "change-me-in-production"
app.config["SESSION_TYPE"] = "sqlalchemy"
app.config["SESSION_SQLALCHEMY"] = db      # reuse our db object
Session(app)
```

The cookie now carries only a random id; the session data lives in a table, beside the rest of the application's data.

Outline

Listing

Python

```python
@app.route("/")
def index():
    rooms = db.session.scalars(
        db.select(Room).order_by(Room.id)
    ).all()
    return render_template("index.html", rooms=rooms)
```

Jinja

```jinja
<table>
  <thead><tr><th>Room</th><th>Capacity</th><th>Price</th></tr></thead>
  <tbody>
  {% for r in rooms %}
    <tr>
      <td><a href="{{ url_for('show_room', id=r.id) }}">{{ r.id }}</a></td>
      <td>{{ r.capacity }}</td>
      <td>{{ "%.2f"|format(r.price) }} EUR</td>
    </tr>
  {% endfor %}
  </tbody>
</table>
```

Detail page

Python

```python
from flask import abort

@app.route("/rooms/<int:id>")
def show_room(id):
    room = db.session.get(Room, id)
    if room is None:
        abort(404)
    return render_template("room.html", room=room)
```

In `room.html`, the relationship `room.reservations` gives us the list of reservations for free:

Jinja

```jinja
<h2>Reservations for room {{ room.id }}</h2>
<ul>
{% for r in room.reservations %}
  <li>{{ r.guest.name }}: {{ r.arrival }} ({{ r.nights }} nights)</li>
{% endfor %}
</ul>
```

Beware of the N+1 problem

Naive listing of every room with its reservation count:

Python

```python
for room in Room.query.all():
    print(room.id, len(room.reservations))   # one extra query each!
```

Beware of the N+1 problem

Naive listing of every room with its reservation count:

Python

```
for room in Room.query.all():
    print(room.id, len(room.reservations))   # one extra query each!
```

Two cures, already mentioned in lecture 8:

- eager-load the relation:
  `db.select(Room).options(db.selectinload(Room.reservations))`
- or write the join yourself, in a single query:

Python

```
rows = db.session.execute(
    db.select(Room, Reservation).join(Reservation)).all()
```

Watch your dev server log: SQLAlchemy can be told to print every query with `app.config["SQLALCHEMY_ECHO"] = True`.

Outline

Search through the query string

URL: /search?q=alice

Python

```python
@app.route("/search")
def search():
    q = request.args.get("q", "").strip()
    if q:
        rows = db.session.scalars(
            db.select(Reservation)
              .join(Guest)
              .where(Guest.name.icontains(q))
              .order_by(Reservation.arrival)
        ).all()
    else:
        rows = []
    return render_template("search.html", q=q, rows=rows)
```

Pagination

Long lists need to be paged: `/rooms?page=3`

Python

```python
PER_PAGE = 20

@app.route("/rooms")
def list_rooms():
    page = request.args.get("page", 1, type=int)
    q    = db.select(Room).order_by(Room.id)
    rooms = db.paginate(q, page=page, per_page=PER_PAGE)
    return render_template("rooms.html", rooms=rooms)
```

Template:

Jinja

```jinja
{% for r in rooms.items %} ... {% endfor %}

{% if rooms.has_prev %}
  <a href="{{ url_for('list_rooms', page=rooms.prev_num) }}">Prev</a>
{% endif %}
```

Outline

A booking form

templates/book.html:

Jinja

```
<form action="{{ url_for('book') }}" method="post">
  <label>Guest:
    <input name="guest" required></label>
  <label>Email:
    <input type="email" name="email"></label>
  <label>Room:
    <select name="room">
      {% for r in rooms %}
        <option value="{{ r.id }}">{{ r.id }}</option>
      {% endfor %}
    </select></label>
  <label>Arrival:
    <input type="date" name="arrival" required></label>
  <label>Nights:
    <input type="number" name="nights" min="1" required></label>
  <button>Book</button>
</form>
```

The matching route

Python

```python
from datetime import date

@app.route("/book", methods=["GET", "POST"])
def book():
    if request.method == "POST":
        guest = Guest(name=request.form["guest"],
                      email=request.form.get("email") or None)
        r = Reservation(
            guest   = guest,
            room_id = int(request.form["room"]),
            arrival = date.fromisoformat(request.form["arrival"]),
            nights  = int(request.form["nights"]),
        )
        db.session.add(r)        # cascades to the new guest
        db.session.commit()
        flash(f"Booked room {r.room_id} for {guest.name}.", "success")
        return redirect(url_for("show_room", id=r.room_id))
    rooms = db.session.scalars(db.select(Room)).all()
    return render_template("book.html", rooms=rooms)
```

Post / Redirect / Get, once more

After `commit`, we redirect: a `303 See Other` to a GET URL.

- Reloading the resulting page no longer resubmits the form
- The Back button does not warn "Form will be resent"
- The URL is bookmarkable and shareable

Post / Redirect / Get, once more

After `commit`, we redirect: a `303 See Other` to a GET URL.

- Reloading the resulting page no longer resubmits the form
- The Back button does not warn "Form will be resent"
- The URL is bookmarkable and shareable

Rule: every state-changing endpoint ends with a redirect, not with an HTML response.

Outline

Updating a record

Python

```python
@app.route("/reservations/<int:id>/edit", methods=["GET", "POST"])
def edit_reservation(id):
    r = db.session.get(Reservation, id) or abort(404)
    if request.method == "POST":
        r.nights = int(request.form["nights"])
        db.session.commit()      # no add(): r is already tracked
        return redirect(url_for("show_room", id=r.room_id))
    return render_template("edit_reservation.html", r=r)
```

Updating a record

Python

```python
@app.route("/reservations/<int:id>/edit", methods=["GET", "POST"])
def edit_reservation(id):
    r = db.session.get(Reservation, id) or abort(404)
    if request.method == "POST":
        r.nights = int(request.form["nights"])
        db.session.commit()      # no add(): r is already tracked
        return redirect(url_for("show_room", id=r.room_id))
    return render_template("edit_reservation.html", r=r)
```

An object obtained through the session is tracked: assigning to its attributes is enough; the next `commit` issues the `UPDATE`.

Deletion: never via GET

Wrong: a plain link `<a href="/reservations/42/delete">Cancel</a>`.

- Any link preview, web crawler or browser prefetch will delete the reservation
- Violates HTTP: `GET` must be safe

Deletion: never via GET

Wrong: a plain link `<a href="/reservations/42/delete">Cancel</a>`.

- Any link preview, web crawler or browser prefetch will delete the reservation
- Violates HTTP: `GET` must be safe

Right: a one-button form, with `POST`:

Jinja

```
<form action="{{ url_for('cancel_reservation', id=r.id) }}"
      method="post">
  <button>Cancel reservation</button>
</form>
```

Python

```
@app.route("/reservations/<int:id>/cancel", methods=["POST"])
def cancel_reservation(id):
    db.session.delete(db.session.get(Reservation, id) or abort(404))
    db.session.commit()
    return redirect(url_for("index"))
```

Validation

User input cannot be trusted: even a well-meaning form may send garbage if the user changed the HTML in the dev tools.

- Cast and clamp values: `int(request.form["nights"])` inside a `try`/`except`, or `request.form.get("nights", type=int)`
- Re-render the form with an error if anything is wrong
- Let the database have the last word: constraints (`NOT NULL`, `CHECK`, `FOREIGN KEY`) protect you when the application code does not
- For larger forms, use a library: Flask-WTF, pydantic or marshmallow

Outline

The two ends of the same string

Two classic vulnerabilities, both due to mixing trusted syntax with untrusted data:

SQL injection user input ends up inside an SQL query and changes its structure
→ attacker reads or destroys data

Cross-Site Scripting (XSS) user input ends up inside an HTML page and changes its structure
→ attacker runs JavaScript in other users' browsers

The two ends of the same string

Two classic vulnerabilities, both due to mixing trusted syntax with untrusted data:

SQL injection user input ends up inside an SQL query and changes its structure
→ attacker reads or destroys data

Cross-Site Scripting (XSS) user input ends up inside an HTML page and changes its structure
→ attacker runs JavaScript in other users' browsers

Both have the same cure: never concatenate strings across a language boundary. Use the tool that does proper escaping for that boundary.

SQL injection through a form

Tempting and wrong:

Python

```python
@app.route("/login", methods=["POST"])
def login():
    name = request.form["name"]
    pwd  = request.form["password"]
    cur  = get_db().cursor()
    cur.execute(
        "SELECT * FROM user WHERE name='" + name +
        "' AND password='" + pwd + "'")
    row = cur.fetchone()
    return "OK" if row else "Bad credentials"
```

SQL injection through a form

Tempting and wrong:

Python

```python
@app.route("/login", methods=["POST"])
def login():
    name = request.form["name"]
    pwd  = request.form["password"]
    cur  = get_db().cursor()
    cur.execute(
        "SELECT * FROM user WHERE name='" + name +
        "' AND password='" + pwd + "'")
    row = cur.fetchone()
    return "OK" if row else "Bad credentials"
```

Submit `name = "admin' --"` and any password; the query becomes `... WHERE name='admin' --' AND password=...`, the `--` comments out the rest, and the attacker is logged in as `admin`.

The parameterised cure

Right (DB-API):

Python

```
cur = get_db().cursor()
cur.execute(
    "SELECT * FROM user WHERE name=%s AND password=%s",
    (name, pwd))
row = cur.fetchone()
```

Right (SQLAlchemy Core / ORM):

Python

```
db.session.execute(
    db.select(User).where(User.name == name, User.password == pwd)
)
```

The parameterised cure

Right (DB-API):

Python

```
cur = get_db().cursor()
cur.execute(
    "SELECT * FROM user WHERE name=%s AND password=%s",
    (name, pwd))
row = cur.fetchone()
```

Right (SQLAlchemy Core / ORM):

Python

```
db.session.execute(
    db.select(User).where(User.name == name, User.password == pwd)
)
```

Two rules to live by:

- Placeholders (%s / named parameters) for values
- String concatenation only for schema names that you hard-code (table names, column names); never from user input

When the column itself comes from the user

Sorting by a user-chosen column: `/rooms?sort=price`

- Parameters do not work for identifiers
- Do not interpolate either: `?sort=price;DROP TABLE room` would be catastrophic

When the column itself comes from the user

Sorting by a user-chosen column: `/rooms?sort=price`

- Parameters do not work for identifiers
- Do not interpolate either: `?sort=price;DROP TABLE room` would be catastrophic

Whitelist:

Python

```
ALLOWED_SORTS = {"id": Room.id, "price": Room.price,
                 "capacity": Room.capacity}
col = ALLOWED_SORTS.get(request.args.get("sort", "id"), Room.id)
rooms = db.session.scalars(db.select(Room).order_by(col)).all()
```

Any value not in the whitelist falls back to a safe default.

Outline

XSS from the database

An attacker may put <script>...</script> in the database, not just in the form. The danger surfaces when the page is rendered:

Jinja

```
<!-- safe: Jinja escapes by default -->
<p>Guest: {{ guest.name }}</p>

<!-- DANGER -->
<p>Guest: {{ guest.name|safe }}</p>
```

XSS from the database

An attacker may put `<script>...</script>` in the database, not just in the form. The danger surfaces when the page is rendered:

Jinja

```
<!-- safe: Jinja escapes by default -->
<p>Guest: {{ guest.name }}</p>

<!-- DANGER -->
<p>Guest: {{ guest.name|safe }}</p>
```

Rules:

- Never apply `|safe` to a string that originally came from a user
- Escape on output, not on input. Store data raw in the DB; let the template engine handle the HTML escaping
- Set the `Content-Security-Policy` HTTP header to forbid inline scripts (defense in depth)

The OWASP Top 10

Industry-standard list of the most common Web vulnerabilities, updated every few years:

- A03 Injection (SQL, NoSQL, command, LDAP...)
- A04 Insecure Design
- A05 Security Misconfiguration
- A07 Identification & Authentication Failures
- A08 Software & Data Integrity Failures
- ...

Read once and revisit every project: `https://owasp.org/Top10/`.

Outline

The natural unit

- Each HTTP request is, conceptually, a unit of work: either it succeeds entirely or it should have no effect
- One transaction per request is the standard pattern
- Flask-SQLAlchemy starts a transaction lazily on first query, and we `db.session.commit()` (or `db.session.rollback()`) before returning the response

The natural unit

- Each HTTP request is, conceptually, a unit of work: either it succeeds entirely or it should have no effect
- One transaction per request is the standard pattern
- Flask-SQLAlchemy starts a transaction lazily on first query, and we `db.session.commit()` (or `db.session.rollback()`) before returning the response

When an exception escapes the route function, the teardown handler issues a rollback automatically: atomicity is preserved without manual `try`/`except` everywhere.

Avoiding the lost-update race

Two clerks open the same reservation, both change the nights, and click “Save”. Last writer wins, silently – a plain `commit` cannot tell, since the overwriting write succeeds.

Avoiding the lost-update race

Two clerks open the same reservation, both change the nights, and click “Save”. Last writer wins, silently – a plain `commit` cannot tell, since the overwriting write succeeds. Fix: write only if the row still holds the value the clerk first saw (carried in a hidden field), and check how many rows matched:

Python

```
updated = db.session.execute(
    db.update(Reservation)
      .where(Reservation.id == id)
      .where(Reservation.nights == old_nights)  # only if unchanged
      .values(nights=new_nights)).rowcount
db.session.commit()
if updated == 0:                  # someone got there first
    flash("Changed meanwhile; reload and retry.")
```

A real app guards on a dedicated `version` column rather than the field itself.

Long requests are bad transactions

- A request that takes 10 s holds locks for 10 s and blocks other writers
- Avoid network calls (HTTP to a third party, slow file uploads) inside a transaction
- For long jobs, save the request to an asynchronous queue of jobs to process and return a redirect to a status page
- Keep the routes short; let the database get on with its work

Outline

What we just built

A complete hotel booking app – every route from the start of the lecture – in a few dozen lines:

- a database (MySQL) with three tables and foreign keys
- pages to list rooms, view a room, book, search, edit and cancel reservations
- HTML-escaped output, parameterised SQL, redirects after writes, flash messages, a shared layout
- a clear seam between the model (SQLAlchemy classes), the request handling (Flask routes) and the presentation (Jinja templates)

What we just built

A complete hotel booking app – every route from the start of the lecture – in a few dozen lines:

- a database (MySQL) with three tables and foreign keys
- pages to list rooms, view a room, book, search, edit and cancel reservations
- HTML-escaped output, parameterised SQL, redirects after writes, flash messages, a shared layout
- a clear seam between the model (SQLAlchemy classes), the request handling (Flask routes) and the presentation (Jinja templates)

The full source is on Moodle; this is the MVC shape we previewed at the start of the lecture.

Outline

What we learnt

- How to attach a database to Flask cleanly: one connection (or session) per request, opened on demand, closed on teardown
- How to read rows and feed them to Jinja templates
- How to handle forms safely: Post / Redirect / Get, parameterised queries, server-side validation, no destructive `GET`
- How the two great Web vulnerabilities – SQL injection and XSS – both come from the same bad habit, and how Flask and Jinja help us avoid them
- How transactions map onto requests, and where this breaks down

To explore on your own

- Flask tutorial, "Flaskr": `https://flask.palletsprojects.com/en/stable/tutorial/`
- Flask-SQLAlchemy documentation: `https://flask-sqlalchemy.readthedocs.io/`
- OWASP Top 10: `https://owasp.org/Top10/`
- Read the SQL queries SQLAlchemy emits, with `SQLALCHEMY_ECHO=True`; many surprises (and N+1 bugs) hide there
- Look at the Network tab of your browser's dev tools while interacting with your app; every HTTP exchange is visible

License