# Database Project, Part 2

Emy Marie Bimond ist1116694

Ilia Golub ist1118697

Ovidiu-Gabriel Lasconi ist1116578

**Group**: 20

**Laboratory teacher**: Professor Paulo Carreira, Professor Francisco Regateiro

**Contributions and effort**: All three group members contributed approximately equally to the project, each with about one third (~33%) of the work and roughly 6 hours of effort.

## Database explanation

The database used in this project is based on the relational schema provided by in the file **schema-part2.sql.**

The schema models the core domain of a boat reservation and trip management system. Countries and locations define the geographical context. Sailors are stored in a general table and specialized into juniors and seniors, with only seniors allowed to be responsible for reservations. Boats are registered in countries, belong to a boat class, and have a defined length. Reservations associate boats with date intervals, define a responsible senior sailor, and specify which sailors are authorized. Trips represent the actual execution of reservations, linking a skipper, departure and arrival locations, dates, and insurance information. Sailing certificates and their validity across countries model legal restrictions on who is allowed to skipper certain boats.

The file **populate.sql** was created as part of the project to initialize the database with meaningful test data. The script inserts data in a dependency-safe order that respects all foreign key constraints. It creates a coherent scenario with multiple countries and locations, sailors with different specializations, boat classes, boats, certificates with limited validity, reservations with authorized sailors, and trips reflecting real boat usage. The dataset intentionally includes both standard and edge cases, such as boats that are reserved but never used, sailors whose certificates are valid only in specific countries, and trips where the skipper lacks the appropriate certificate.

## Integrity Constraints (ICs) implementation

Two integrity constraints are enforced using SQL procedural extensions (stored procedures and triggers) and deferred constraint checking.

### IC-1 – Every Sailor is either Senior or Junior

The sailor entity is a *total* and *disjoint* specialization into the subtypes junior and senior: every sailor must belong to *exactly one* of them (not none, not both).

This is a cross-table constraint that cannot be expressed as a single CHECK or FOREIGN KEY, because it requires:

**Totality**: for every row in sailor, there must exist a matching row in junior OR senior;

**Disjointness**: the same email cannot appear in both subtype tables.

**How we enforced it:**

- We created a PL/pgSQL function that checks:
  whether any email exists in both junior and senior (disjointness violation);
  whether any sailor email exists in neither junior nor senior (totality violation).
- We attached it as **CONSTRAINT TRIGGERs** on **all three tables** (sailor, junior, senior) for INSERT/UPDATE/DELETE, because any of these operations may introduce a violation.
- The triggers are **DEFERRABLE INITIALLY DEFERRED**, so the database checks the constraint at transaction end. This matters because creating/removing a sailor is typically a *multi-step* change (e.g., insert into sailor and then insert into junior/senior). With deferred checking, intermediate states inside the same transaction are allowed as long as the final committed state is valid.

**Practical consequence for the web app.** "Create sailor" must run as one transaction that inserts into sailor and exactly one subtype table, and only then commits (otherwise the deferred trigger would raise an exception at commit time).

**IC-2 – Trips for the same reservation must not overlap**

For a given reservation, two trips cannot overlap in time: a trip cannot take off before another trip arrives.

This is a *set-based temporal constraint* across multiple rows of trip (it depends on other trips of the same reservation). You can't express it with a simple key/foreign key, and a plain CHECK cannot query other rows.

**How we enforced it:**

- We created a PL/pgSQL function that, on insert/update of a trip, searches for any *other* trip of the **same reservation** whose interval overlaps the new/updated interval.
- We used the standard overlap logic: overlap exists when existing.takeoff < new.arrival **and** new.takeoff < existing.arrival (this correctly allows such trips where arrival == next_takeoff).
- The trigger is a **CONSTRAINT TRIGGER** on trip for INSERT/UPDATE. Deleting a trip cannot introduce overlaps, so DELETE is unnecessary.
- It is declared **DEFERRABLE INITIALLY IMMEDIATE**: it is checked right away by default, but can still be deferred inside a transaction if needed.

**Practical consequence for the web app.** "Register trip" is safe as a single statement, but if a workflow requires multiple related inserts/updates, it can still be done atomically inside one transaction.

The project statement also says that constraints that can be expressed without procedural extensions should use standard mechanisms instead (and forbids ON DELETE/UPDATE CASCADE). In our solution, all "simple" integrity rules (keys, referential integrity, domains) are handled with PRIMARY KEY, FOREIGN KEY, UNIQUE, and CHECK in the schema, while IC-1 and IC-2 are implemented in ICs.sql using triggers as described above.

**SQL Queries**

The queries required are implemented in the queries.sql file.

**1 - Country with the most boats**

This query identifies the country with the highest number of registered boats.
The boats are grouped by country and counted. The ALL operator is used to compare each country's count with all others, selecting the countries with the maximum number of boats.
This solution uses only standard SQL constructs and avoids vendor-specific features.

**2 - Sailors with at least two certificates**

This query retrieves sailors who hold at least two sailing certificates.
Certificates are grouped by sailor, and the HAVING clause is used to filter only those sailors whose number of certificates is greater than or equal to two.
The query relies solely on aggregation over the sailing_certificate table.

**3 - Sailors who sailed to every location in Portugal**

This query finds sailors who have sailed to every location located in Portugal.
The solution follows a relational division approach implemented with nested NOT EXISTS subqueries.
For each sailor, the query checks that there is no Portuguese location for which the sailor has never performed a trip.
A location is considered visited if the sailor departed from or arrived at that location during a trip.

**4 - Sailors with the most skipped trips**

This query lists the sailors who skipped the highest number of trips.
A skipped trip is defined as a reservation for which the sailor was authorized but did not act as skipper in any associated trip.
The query counts such skipped trips per sailor and uses a subquery with the ALL operator to select the maximum count.

**5 - Sailors with the longest total trip duration for a single reservation**

This query identifies the sailors who accumulated the longest total sailing time for a single reservation.
For each combination of sailor and reservation, the durations of all associated trips are summed.
The query then selects the sailors whose total duration is greater than or equal to all others, also displaying the summed duration.

## View explanation

The view part of the assignment is implemented in the file **view.sql** and is named **TRIP_INFO**. Its purpose is to provide a concise summary of trip-related information by combining data that is otherwise spread across multiple tables.

It is built by joining the trip table with location, country, and boat. The origin and destination locations are determined by matching the latitude and longitude stored in the trip table with the corresponding rows in the location table. From these locations, the associated countries are retrieved, allowing the view to expose both ISO codes and human-readable country names for the origin and destination.

Boat information is obtained by joining the trip table with the boat table using the composite key (country, cni). The country in which the boat is registered is then joined again with the country table to retrieve its ISO code and name. This design clearly distinguishes between the origin country, destination country, and boat registration country, which may differ.

The attribute **TRIP_START_DATE** corresponds to the takeoff date of the trip and provides a clear temporal reference for each entry. Overall, the **TRIP_INFO** view encapsulates a complex set of joins into a single reusable database object, improving readability, reducing duplication of SQL logic, and ensuring a consistent representation of trip information for queries and the web application.

## Web app architecture

### Goal and scope

**A minimal web prototype** was implemented with **Python CGI** and **HTML** that supports the four workflows:
(a) sailors (list/create/remove), (b) reservations (list/create/remove), (c) authorize/de-authorize sailors for reservations, and (d) trips (list/register/remove and list locations). The solution also prevents **SQL injection** and ensures **atomicity** of related DB operations using **transactions**.

**Working version (link):** http://web2.tecnico.ulisboa.pt/ist1118697/header.cgi

### High-level structure

Each feature is a dedicated CGI endpoint that:
renders an HTML page (GET), and
handles form submissions (POST) to perform inserts/deletes and then re-renders the updated view.

### File responsibilities and relations

To keep the code simple and consistent, the app is split into **reusable modules** and **feature CGI pages**. The shared modules encapsulate database access and HTML rendering so that each CGI script stays small and focused.

**Dependency map**:

**Reusable modules:**

- login.py (not included in submission): stores the Postgres credentials string.

- db.py: a small connection helper exposing a connect() context manager. CGI scripts open connections via with db.connect() as conn: and can wrap multi-step operations in with conn: to commit/rollback atomically.
- ui.py: shared HTML helpers (page header/footer, navigation bar, message boxes), plus **HTML escaping** via h() to avoid injecting untrusted values into HTML output. All pages share the same layout/navigation because ui.py prints a common header with links to each page.

**CGI feature pages -> (db.py, ui.py):**

- header.cgi: "home" page with navigation.
- sailors.cgi: list/create/remove sailors (and insert into junior/senior subtype tables).
- reservations.cgi: list/create/remove reservations (including inserting the matching date_interval, and ensuring the responsible sailor starts as authorized).
- authorised.cgi: manage authorized sailors per reservation.
- trips.cgi: list/register/remove trips for a selected reservation and show available locations.

Each CGI endpoint imports db and ui and does **not** import other CGI scripts; "coordination" between pages happens through standard HTTP navigation links in the shared header. This makes each feature page independently executable and easier to test.

**Request/response flow**

For every page, the runtime flow is the same:

**Browser -> Apache CGI -> <page>.cgi -> db.connect() -> PostgreSQL -> HTML response**

A **GET** request renders the page with current DB state (tables and forms).
A **POST** request performs an action (insert/delete) using parameterized SQL, typically inside a transaction, and then re-renders the updated state.

**Security (SQL injection and output safety)**

- **SQL injection prevention:** all DB writes use **parameterized SQL** (cursor.execute(sql, data)).
- **HTML output safety:** dynamic values printed into HTML are escaped using ui.h() so user-controlled strings (e.g., email) don't break the page or inject markup.

**Transactions and atomic operations**

In our design, every multi-step write is executed inside a single transaction boundary (with conn:). This ensures that if any step fails (FK, trigger exception, etc.), the whole change is rolled back automatically. The intended usage pattern is documented directly in db.py.

Examples of multi-step atomic operations in the prototype include:

- creating a reservation + inserting its date_interval + inserting the initial "authorized" row for the responsible sailor;
- deleting a reservation + deleting dependent trips and authorizations first (without relying on ON DELETE CASCADE);
- registering a trip (single insert, but still done inside a transaction to cleanly handle constraint failures).

## Indexes

In our database, we **did not create extra manual indexes**. Instead, we rely on the **indexes automatically created by PostgreSQL** for PRIMARY KEY and UNIQUE constraints defined in schema-part2.sql. These indexes both **enforce uniqueness** and **speed up joins/lookups** over key attributes that are used by the web application and the SQL queries.

**Automatically created indexes (from schema constraints)**

**Country**

- country(name) – **PK index** (used when other tables reference the country by name)
- country(iso_code) – **UNIQUE index** (enforces ISO uniqueness required by the specification)

**Location**

- location(latitude, longitude) – **PK index** (uniquely identifies a location)

**Sailors and subtypes**

- sailor(email) – **PK index**
- junior(email) – **PK index**
- senior(email) – **PK index**

**Boats and certificates**

- boat_class(name) – **PK index**
- boat(country, cni) – **PK index** (identifies a boat by its registration country + CNI)
- sailing_certificate(sailor, issue_date) – **PK index**
- valid_for(country_name, sailor, issue_date) – **PK index**

**Reservations and trip-related tables**

- date_interval(start_date, end_date) – **PK index**
- reservation(start_date, end_date, country, cni) – **PK index**
- authorised(start_date, end_date, boat_country, cni, sailor) – **PK index**
- trip(takeoff, reservation_start_date, reservation_end_date, boat_country, cni) – **PK index**

The CGI application accesses and modifies records primarily by their **natural identifiers**, i.e., the same attributes used in the primary keys. Because the most common operations filter by these full keys (or by key prefixes), the automatically created PK indexes already align well with the application's access patterns.

PostgreSQL does not automatically create indexes for every foreign key column. For large datasets, adding indexes on frequently-joined FK columns can improve performance. However, for this project's minimal prototype and moderate data volume, the PK/UNIQUE indexes above were sufficient, and we kept the schema intentionally simple (consistent with the project's emphasis on clarity and explainability).