ChatGPT

Academic Timetable Generator (Flask + Supabase + OR-Tools)

A **full-stack web app** can be built using Python/Flask for the backend and Jinja2 templating with Bootstrap 5 for the frontend. All user data and scheduling data live in a Supabase-managed PostgreSQL, and authentication is handled by Supabase Auth (email/password only). The **Flask** app (`app.py`) renders Jinja templates (in `templates/`) and serves static assets (in `static/`). The **data tier** is a shared PostgreSQL (via Supabase) with tables for Users, Institutions, Departments, Teachers, Courses, Rooms, TimeSlots, TimetableEntries, FacultyAvailability, ElectiveGroups, and InstitutionRules (as specified). Using Supabase's Row-Level Security (RLS) and foreign-key constraints ensures data integrity and multi-tenant isolation [1]. All configuration values (Supabase URL, API keys, etc.) are stored in environment variables (`.env` or Replit secrets) so no values are hardcoded. Below is an outline of the system design.

Authentication & User Management

- **Supabase Auth (Email/Password)**: Use the `supabase-py` client to handle signup and login. For example, new users sign up with:

```
response = supabase.auth.sign_up({
    "email": "user@example.com",
    "password": "secret123"
})
```

and login with:

```
session = supabase.auth.sign_in_with_password({
    "email": "user@example.com",
    "password": "secret123"
})
```

(This follows the Supabase Python SDK examples [2] [3].) The Flask app verifies the returned JWT or session token on each request to protected pages. User sessions are stored server-side (e.g. in Flask's session or a secure cookie) and tied to Supabase Auth. The **supabase_client.py** module encapsulates these calls. Once authenticated, user info (including custom `role` metadata) is available from Supabase's JWT, which the backend can use to enforce access rules [4] [5].

Frontend UI (Flask + Jinja2 + Bootstrap)

- **Templates & Layout:** Use Jinja2 to create reusable HTML templates. For example, a base layout (`base.html`) includes a Bootstrap navbar, footer, and links to CSS/JS. Static assets (Bootstrap CSS/

JS) are placed under `static/` and referenced via `url_for`, e.g. `{{ url_for('static', filename='css/bootstrap.min.css') }}` [6]. Each page (Login, Signup, Dashboard, Timetable Editor, Profile, Notifications) extends this base template. Templates can use Jinja conditionals to show/hide elements based on user state (logged in vs not, or role) [7] [8].

- **Bootstrap 5:** Leverage Bootstrap 5's grid and components for a clean responsive design [9]. For instance, use Bootstrap form classes for login/signup pages, cards or tables for dashboard overviews, and a responsive grid for the timetable. A sticky navbar (Bootstrap's `.navbar`) provides navigation between pages (Dashboard, Editor, Profile, Notifications, Logout). Minimal CSS transitions (e.g. `transition` on hover or collapse) suffice; no heavy JS animations are needed. Bootstrap's built-in JavaScript (bundled) can handle the navbar toggle and modals (e.g. for edit dialogs) without custom code. The result is a professional, mobile-friendly UI (in line with Bootstrap's responsive templates [9] [6]).

Scheduling Algorithm & Timetable Editor

Figure: Example scheduling pipeline. Course requests and constraints are filtered (top) to produce conflict-free timetable options (bottom).

The core is a **constraint-satisfaction scheduling engine** (`scheduler/timetable_engine.py`). Timetabling is **NP-hard** [10]: we treat each class as a variable whose value is a (timeslot, room, teacher) assignment. Typical *constraints* include: no student or instructor is double-booked, no two classes in the same room at once, room capacity must fit the class size, teachers must be available, mandatory breaks, no classes on holidays, etc. [10] [11]. We first model all *hard* constraints and try to find an assignment that satisfies them all. For example, OR-Tools' CP-SAT solver is well-suited for this purpose: it can encode classes as variables and add constraints like "class A and class B cannot share a timeslot" or "teacher X's assigned slots must respect their availability" [12] [11]. In practice, one might use a combination of heuristics (e.g. greedy backtracking) and OR-Tools optimization to minimize conflicts (teacher overload or idle periods) [12].

The **Timetable Editor** (Jinja/HTML with optional JavaScript) displays the generated schedule as a grid (days vs periods) for each student group or section. After auto-generation, administrators can **manually adjust** it in the UI (e.g. drag-and-drop or select a class cell to reassign time/room). This can be implemented with a dynamic HTML table and minimal JS (or Bootstrap modals to edit entries). All manual edits also trigger backend checks to prevent new conflicts. During scheduling, any *conflicts* detected (e.g. no possible slot found) are reported so they can be reviewed. Analytics like room utilization or teacher workload can be computed from the final schedule and displayed on the dashboard.

Data Management & Supabase Backend

All data is stored in Supabase's Postgres. Key tables (with foreign-key relations) include **Users, Institutions, Departments, Teachers, Courses, Rooms, TimeSlots, ElectiveGroups, FacultyAvailability, InstitutionRules**, and **TimetableEntries** (the generated schedule rows). Each table should have an `institution_id` (and optionally `campus_id`) to enforce tenancy [1]. For example, the data tier uses one Postgres schema for all tenants, with an `institution_id` column on each table, and Supabase's RLS policies (or simple WHERE clauses) enforce that users only see their own data [1] [4]. Every table should have appropriate primary keys and foreign keys (e.g. `Teacher(dept_id)→Department.id`,

`Course(dept_id)→Department.id`, `TimetableEntry(course_id, room_id, timeslot_id, teacher_id)`, etc.) to maintain referential integrity [1].

The **supabase_client.py** module centralizes all database interactions using the Supabase Python SDK. For CRUD operations, it uses calls like:
- **Create:** `supabase.table("Courses").insert({"id": 1, "name": "Calculus", ...}).execute()` [13].
- **Read:** `supabase.table("Departments").select("*").eq("institution_id", current_institution).execute()`.
- **Update:** `supabase.table("Teachers").update({"name": "Dr. Smith"}).eq("id", 5).execute()` [14].
- **Delete:** `supabase.table("Rooms").delete().eq("id", 42).execute()` [15].
These correspond to Admin CRUD actions in the dashboard for each entity. The returned `response` includes status and data. All Supabase calls occur over HTTPS to the hosted DB – **no local database code** is needed.

The admin dashboard UI provides **CRUD pages** (forms and tables) for managing these entities. For example, an "Add Course" form submits to a Flask route that calls `supabase.table("Courses").insert(...)`. Similarly, list views retrieve data via `select().execute()` [16] and render HTML tables. This ensures data is dynamic (not hardcoded) and immediately stored in Supabase.

Profile & Role-Based Access Control

We implement three roles: **Admin, Faculty, Student**. Upon signup, users are assigned a role (stored as a user metadata or in a separate Roles table). Supabase Auth can include a custom `user_role` claim in the JWT, which our RLS policies or backend logic uses to enforce permissions [4] [5]. For instance, only Admins see the Institutions and global settings pages; only Department Heads (admin-faculty hybrid role) can run the timetable generator or edit department data; faculty can view their own schedules and submit change requests; students can only view their enrolled timetable [7] [8]. In Flask routes/templates, we check the user's role and conditionally enable UI elements (e.g. hide "New User" button from non-admins).

The **Profile** page lets any user view and edit their own information (except role). We call Supabase to update the Users table and also use `supabase.auth.update_user()` if changing password. The **Logout** route simply clears the Flask session and calls `supabase.auth.sign_out()` if needed. A small **Notifications** page can list announcements or system messages (pulled from a Supabase table or real-time channel), but heavy messaging could also use Supabase Realtime or email triggers [17].

Export & Reporting

Users can export generated timetables in PDF and Excel formats. On the backend, the current schedule data is fetched from Supabase and formatted into files: for Excel, using Pandas' `DataFrame.to_excel()` (or OpenPyXL/xlsxwriter) to produce an `.xlsx` [18]; for PDF, using a library like ReportLab or PDFKit (server-side) to create a styled table. For example, one might build a Pandas DataFrame of all entries and call `df.to_excel("timetable.xlsx")` [18], or use ReportLab to draw the table into a PDF. The exported file can include a summary of any scheduling conflicts or alerts (e.g. unresolved gaps) and simple analytics

like room utilization or teacher hours. (In-browser, we could also use a Jinja template + wkhtmltopdf or WeasyPrint to render HTML as PDF.)

A Python PDF library comparison notes options like FPDF2, ReportLab, PDFKit, WeasyPrint, etc. [19]. Our code would pick one (e.g. `reportlab` for tables), install it via `requirements.txt`, and produce a download when the user clicks "Export PDF." Similarly, Pandas is in `requirements.txt` for Excel. Any charts (e.g. bar chart of daily workload) could be generated with Matplotlib/Seaborn or done client-side with Chart.js/D3.js (as suggested in design guides [20]). The user can then download or receive via email the Excel or PDF file.

Deployment on Replit

The app runs entirely on Replit with no special setup. All secrets (SUPABASE_URL, SUPABASE_KEY) are set in Replit's environment. The `requirements.txt` includes `Flask`, `supabase-py`, `python-dotenv`, `pandas`, `openpyxl`, `reportlab`, `ortools`, etc. The entry point is `app.py` (Flask) as given. No local database server is needed since we use Supabase's hosted Postgres. By using this folder layout and `app.run()` in `app.py`, Replit will detect the Flask server. All environment-specific config (keys, instance URL) comes from `.env` or Replit secrets.

In summary, this design meets all requirements: a Flask/Jinja + Bootstrap UI; Supabase Auth for login; Python/OR-Tools scheduling; Supabase PostgreSQL for data; role-based access; PDF/Excel export; and a Replit-friendly structure. Citations above reference best practices for templating [6] [9], Supabase integration [21] [2], and constraint-scheduling methods [22] [10] that guide this implementation.

Folder structure example (per spec):

```
/ (root)
├── app.py
├── templates/
│   ├── login.html
│   ├── signup.html
│   ├── dashboard.html
│   ├── timetable.html
│   ├── profile.html
│   └── base.html
├── static/
│   ├── css/
│   └── js/
├── scheduler/
│   └── timetable_engine.py
├── supabase_client.py
├── utils/
│   └── validators.py
```

```
├── requirements.txt
└── .env        # Supabase keys and other secrets
```

This clean, modular layout (inspired by Flask best practices [23]) is ready for open-source publishing and easy maintenance. All input fields, entity data, and constraints are dynamic (no hard-coded values).

Sources: Authoritative guides and docs on Flask/Jinja templating and Bootstrap [6] [9]; Supabase Python API (auth and CRUD) [2] [21]; and academic timetabling methods (NP-hard scheduling, CSP/OR-Tools) [10] [22] informed this design.

---

[1] [5] [7] [8] [12] [17] [20] [22] Academic Timetable Generator – System Design & Implementation Guide.pdf
file://file-TvuVsyCRwMPns3teASwBay

[2] Python: Create a new user | Supabase Docs
https://supabase.com/docs/reference/python/auth-signup

[3] Python: Sign in a user | Supabase Docs
https://supabase.com/docs/reference/python/auth-signinwithpassword

[4] Custom Claims & Role-based Access Control (RBAC) | Supabase Docs
https://supabase.com/docs/guides/database/postgres/custom-claims-and-role-based-access-control-rbac

[6] Python Programming Tutorials
https://pythonprogramming.net/bootstrap-jinja-templates-flask/

[9] [23] Flask + Bootstrap 5 starter web sites | Variance Digital
https://medium.com/variance-digital/flask-bootstrap-5-starter-web-sites-1f1237a85e83

[10] Academic Timetable Scheduling_ Challenges and Comprehensive Solutions.pdf
file://file-NcNvC6kXFJusrHhsEci6kQ

[11] Class Scheduling with AI: Using Google OR-Tools for Constraint Satisfaction Problem
https://blog.ademartutor.com/p/class-scheduling-with-ai-using-google

[13] Python: Insert data | Supabase Docs
https://supabase.com/docs/reference/python/insert

[14] Python: Update data | Supabase Docs
https://supabase.com/docs/reference/python/update

[15] Python: Delete data | Supabase Docs
https://supabase.com/docs/reference/python/delete

[16] [21] How to use Supabase with Flask? | Bootstrapped Supabase Guides
https://bootstrapped.app/guide/how-to-use-supabase-with-flask

[18] pandas.DataFrame.to_excel — pandas 2.3.0 documentation
https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_excel.html

[19] How to Generate PDFs in Python: 8 Tools Compared (Updated for 2025)
https://templated.io/blog/generate-pdfs-in-python-with-libraries/