Sillah Phase 5

CS340: Introduction to Databases Systems

Section: 1629

Instructor: Maram Alajlan

Prepared by:

Yara Albugami - 223410190

Rose Alrakan - 223410382

Shoug Alomran - 223410392

Raghad Abdulaziz - 222511358

Table of Contents

1. System Overview

Sillah (صلة) is a full-stack healthcare management system designed to support preventive health monitoring for individuals and families. The system enables users to manage personal health records, track family members, schedule medical appointments, and receive alerts related to health conditions.

A defining characteristic of the system is its **dual-layer architecture**, which separates real-world application functionality from academic database requirements. The deployed system consists of two integrated components:

1. **Main Prototype Application**

   A fully functional healthcare system that represents the real user-facing platform.

2. **CS340 Phase 5 Demo Layer**

   A database-focused module embedded within the application to demonstrate relational database concepts required by the CS340 course.

Although both components are accessible within the same deployed environment, they operate using **different backend systems and databases**. This ensures that the system remains both **technically realistic** and **academically compliant**, without incorrectly treating Supabase and MySQL as a single unified database.

2. Final System Architecture

The system follows a **hybrid multi-backend architecture**, where each component is responsible for a distinct set of responsibilities. This design enables the system to simultaneously support modern application features and strict relational database requirements.

2.1 Main Prototype Application

The Main Prototype Application represents the **core operational system** used by end users. It provides complete healthcare-related functionality, including:

- User registration and authentication
- Dashboard interfaces
- Appointment management
- Health tracking and alerts

- Doctor-patient relationships
- Family member management

Technology Stack:

- **Frontend:** React + Vite
- **Database:** Supabase (PostgreSQL)
- **Authentication:** Supabase Auth

Supabase was selected for this component because it provides:

- Built-in authentication mechanisms
- Scalable cloud database infrastructure
- Real-time data synchronization

This allows the prototype to behave like a **real-world production system**, supporting responsive and dynamic user interactions.

However, Supabase abstracts many low-level database operations. Therefore, it is not used to demonstrate the explicit SQL-based requirements of the CS340 course.

2.2 CS340 Phase 5 Demo Layer

The CS340 Phase 5 Demo Layer is a **dedicated relational database demonstration module** embedded within the application. It is accessible through the login interface via:

“Open Phase 5 Demo”

This layer is designed specifically to demonstrate:

- CRUD operations using SQL
- Query execution within an application
- Relational schema implementation

Features:

- Users Management (CRUD)
- SQL Query Execution Interface
- Family Members Management (CRUD)

Backend:

- **Node.js (Express)**
- **MySQL database**

API Endpoints:

- `GET /api/health`
- `GET /api/users`
- `POST /api/users`
- `PUT /api/users/:id`
- `DELETE /api/users/:id`
- `GET /api/family-members`
- `POST /api/family-members`
- `DELETE/api/family-members/:id`
- `GET /api/queries/:qid`

This layer ensures that:

- SQL queries are explicitly written and executed
- Database behavior is transparent and verifiable
- Course requirements are fully satisfied

2.3 Data Layer

The system implements a **dual-database architecture**, where each database serves a different purpose:

Layer	Database	Purpose
Main Prototype	Supabase (PostgreSQL)	Real application functionality
CS340 Demo	MySQL	Relational database coursework

Supabase Responsibilities:

- Authentication data
- User profiles
- Doctor-patient relationships
- Appointments and alerts

MySQL Responsibilities:

- Relational schema implementation
- CRUD operations using SQL
- Query execution (q01–q40)

This separation ensures:

- Proper enforcement of relational database concepts
- High performance for application features
- Clear distinction between academic and production layers

3. Authentication Implementation

Authentication is implemented using **Supabase Auth**, which manages user identity and session handling.

```JavaScript
import { createClient } from "@supabase/supabase-js";

const supabaseUrl = import.meta.env.VITE_SUPABASE_URL;
const supabaseAnonKey = import.meta.env.VITE_SUPABASE_ANON_KEY;

export const supabase = createClient(supabaseUrl, supabaseAnonKey);
```

Login Implementation:

```JavaScript
async function login(email, password) {
    const { data, error } = await supabase.auth.signInWithPassword({ email, password });
  if (error) throw error;
  return data.user;
}
```

Route Protection:

```None
export default function ProtectedRoute({ children }) {
  const { currentUser, loading } = useAuth();

  if (loading) return null;
  if (!currentUser) return <Navigate to="/login" replace />;
```

```
  return children;
}
```

This ensures that:

- Only authenticated users can access protected features
- Unauthorized access is prevented

4. Database Implementation

4.1 MySQL Core Relational Tables

The MySQL database is designed using **relational modeling principles**, including:

- Primary Keys (unique identifiers)
- Foreign Keys (relationships between tables)
- Constraints (data validation rules)

For example:

- The `FamilyMember` table references the `User` table
- This prevents orphan records and ensures data consistency

Foreign keys play a critical role in maintaining **referential integrity**, ensuring that relationships between tables remain valid .

4.2 Supabase Application Support Tables

Supabase manages application-level data such as:

- Profiles
- Alerts
- Appointments
- Doctor-patient relationships

These tables are optimized for:

- Real-time interaction
- Scalable application performance
- User-based filtering

4.3 Appointment Functionality Across Both Layers

Appointments are implemented in the Main Prototype Application using Supabase.

```javascript
let query = supabase.from("appointments").select("*");
if (isPatient) {
  query = query.eq("patient_id", currentUser.id);
} else if (isDoctor) {
  query = query.eq("doctor_id", currentUser.id);
}
```

This implementation:

- Filters data based on user roles
- Ensures that users only access relevant records
- Demonstrates application-level access control

4.4 Family Members and Health Features

Family member data is:

- Managed using MySQL in the demo layer
- Integrated into health features in the prototype

This demonstrates how:

- Relational data models support structured storage
- Application layers utilize this data for real functionality

5. Frontend-Backend Integration

The frontend communicates with two backends:

- Supabase for prototype features
- Express API for MySQL operations

Example:

```javascript
const data = await api("/api/users");
setRows(normalizeRows(data));
```

This architecture ensures:

- Clear separation of responsibilities
- Maintainability and scalability
- Modular system design

6. Code Samples

6.1 Supabase Client Configuration

JavaScript

```
import { createClient } from "@supabase/supabase-js";

const supabaseUrl = import.meta.env.VITE_SUPABASE_URL;
const supabaseAnonKey = import.meta.env.VITE_SUPABASE_ANON_KEY;

export const supabase = createClient(supabaseUrl, supabaseAnonKey);
```

6.2 Authentication Login

JavaScript

```
async function login(email, password) {
    const { data, error } = await supabase.auth.signInWithPassword({ email,
password });
  if (error) throw error;
  return data.user;
}
```

6.3 Protected Route

None

```
export default function ProtectedRoute({ children }) {
  const { currentUser, loading } = useAuth();

  if (loading) return null;
  if (!currentUser) return <Navigate to="/login" replace />;
  return children;
}
```

6.4 Fetching Data Through the API

JavaScript

```
const data = await api("/api/users");
setRows(normalizeRows(data));
```

6.5 Inserting Data Through the API

JavaScript

```
await api("/api/family-members", {
  method: "POST",
  body: JSON.stringify({
```

```
    ...form,
    user_id: Number(form.user_id),
  }),
});
```

6.6 Updating Data Through the API

JavaScript

```
app.put("/api/users/:id", async (req, res) => {
  const user_id = Number(req.params.id);
  const { first_name, last_name, email, phone_number } = req.body;

  const [result] = await pool.query(
    `UPDATE \`User\`
     SET first_name = ?, last_name = ?, email = ?, phone_number = ?
     WHERE user_id = ?`,
    [first_name, last_name, email, phone_number, user_id]
  );

  res.json({ affectedRows: result.affectedRows });
});
```

6.7 SQL Query Execution Through the Application

JavaScript

```
app.get("/api/queries/:qid", async (req, res) => {
  const qid = req.params.qid;
  const q = QUERIES[qid];
  if (!q) return res.status(404).json({ error: "Unknown query id" });

  let params = [];

  if (qid === "q03") {
    const term = req.query.term ?? "";
    params = [term];
  }

  return run(res, q, params);
});
```

7. CRUD and SQL Query Execution

The system supports full CRUD operations mapped directly to SQL:

Operation	SQL
Create	INSERT
Read	SELECT
Update	UPDATE
Delete	DELETE

Additionally, the system executes predefined queries (q01–q40), demonstrating:

- Aggregation
- Filtering
- Joins
- Parameterized queries

7.1 Query Usage within the Application

The implemented SQL queries are executed through the application interface and are mapped to different system features:

- q01–q03 → User management dashboard
- q04–q06 → Family member management
- q07–q12 → Health events and medical history views
- q13–q18 → Risk alerts and monitoring system
- q21–q24 → Reporting and joined data views
- q25–q30 → Analytical insights and statistics
- q31–q40 → Advanced filtering and decision-support queries

8. Query Distribution by Team Member

Each team member implemented:

- 5 basic queries
- 5 advanced queries

Total: **40 queries**

9. Deployment

The system is deployed as a unified application combining:

- React frontend
- Supabase backend
- MySQL API server

10. Testing and Validation

Testing included:

- CRUD functionality validation
- API endpoint testing
- Authentication testing
- Query correctness verification

11. Performance Considerations

Performance optimizations include:

- Lazy loading:

```javascript
const Dashboard = lazy(() => import("./Prototype/Pages/Dashboard.jsx"));
```

- Efficient query filtering
- Separation of backend workloads

12. Sustainability Reflection

The system ensures sustainability through modular architecture, separation of concerns, and scalable backend services, allowing independent evolution of the application and database layers without affecting system stability.

13. Individual Contributions

Each team member contributed to:

- Database design
- Frontend development
- Query implementation
- Documentation

14. Conclusion

The Sillah system successfully integrates:

- A real-world healthcare prototype
- A CS340-compliant relational database system

The separation between Supabase and MySQL ensures:

- Technical accuracy
- Practical functionality
- Academic integrity

Appendix A: SQL Queries Implementation (q01-q40)

The following SQL queries are the complete Phase 5 query set, corrected to align with the checked-in MySQL schema in CS340.sql.

A.1 Basic Queries (q01-q20)

q01. Count users

This query returns the total number of users registered in the system.

```
SQL
    SELECT COUNT(*) AS total_users

    FROM User;
```

q02. List latest users

This query retrieves the ten most recently created user records, ordered by creation date.

```
SQL
    SELECT user_id, first_name, last_name, email, phone_number, created_at

    FROM User

    ORDER BY created_at DESC

    LIMIT 10;
```

q03. Search users by email

This query searches for users whose email addresses match a specified keyword or pattern.

```
SQL
    SELECT user_id, first_name, last_name, email

    FROM User
```

```
    WHERE email LIKE CONCAT('%', ?, '%')
    ORDER BY user_id DESC;
```

q04. Count family members

This query calculates the total number of family member records stored in the database.

```
SQL
    SELECT COUNT(*) AS total_family_members
    FROM FamilyMember;
```

q05. List family members

This query retrieves a simple list of the most recently added family members.

```
SQL
    SELECT    member_id,    user_id,    first_name,    last_name,    relationship,
date_of_birth
    FROM FamilyMember
    ORDER BY member_id DESC
    LIMIT 15;
```

q06. Family members for a user

This query displays all family members associated with a specific user account.

```
SQL
    SELECT member_id, first_name, last_name, relationship
    FROM FamilyMember
    WHERE user_id = ?
```

```
    ORDER BY member_id DESC;
```

q07. Count health events

This query returns the total number of health events recorded in the system.

```
SQL
    SELECT COUNT(*) AS total_health_events

    FROM HealthEvent;
```

q08. List health events

This query retrieves recent health events along with their associated member, condition, severity, and date.

```
SQL
    SELECT event_id, member_id, condition_id, severity, event_date

    FROM HealthEvent

    ORDER BY event_date DESC

    LIMIT 15;
```

q09. Events by severity

This query filters health events based on a specified severity level.

```
SQL
    SELECT event_id, member_id, condition_id, severity, event_date

    FROM HealthEvent

    WHERE severity = ?

    ORDER BY event_date DESC;
```

q10. List conditions

This query returns all health conditions stored in the database in alphabetical order.

```SQL
SELECT condition_id, condition_name

FROM HealthCondition

ORDER BY condition_name;
```

q11. List medical history

This query retrieves recent medical history records ordered by event date.

```SQL
SELECT event_id, member_id, condition_id, event_date

FROM MedicalHistory

ORDER BY event_date DESC

LIMIT 15;
```

q12. Medical history for a member

This query displays the medical history details of a specific family member.

```SQL
SELECT event_id, member_id, condition_id, event_date, diagnosis, treatment,
outcome

FROM MedicalHistory

WHERE member_id = ?

ORDER BY event_date DESC;
```

q13. List risk alerts

This query retrieves the latest risk alerts generated in the system.

```sql
SELECT alert_id, member_id, risk_level, status, created_date
FROM RiskAlert
ORDER BY created_date DESC
LIMIT 15;
```

q14. Alerts for a member

This query returns all risk alerts associated with a selected family member.

```sql
SELECT alert_id, member_id, risk_level, status, created_date
FROM RiskAlert
WHERE member_id = ?
ORDER BY created_date DESC;
```

q15. Count alerts by status

This query groups risk alerts by their status and counts the number of alerts in each category.

```sql
SELECT status, COUNT(*) AS total
FROM RiskAlert
GROUP BY status
ORDER BY total DESC;
```

q16. Events for a member

This query retrieves all health events linked to a specific family member.

```sql
SELECT event_id, condition_id, severity, event_date
FROM HealthEvent
WHERE member_id = ?
ORDER BY event_date DESC;
```

q17. High severity events

This query lists only health events marked as high severity.

```sql
SELECT event_id, member_id, condition_id, event_date
FROM HealthEvent
WHERE severity = 'High'
ORDER BY event_date DESC;
```

q18. High risk alerts

This query retrieves alerts classified as high risk.

```sql
SELECT alert_id, member_id, status, created_date
FROM RiskAlert
WHERE risk_level = 'High'
ORDER BY created_date DESC;
```

q19. Users alphabetically

This query lists users ordered alphabetically by last name and first name.

```SQL
SELECT user_id, first_name, last_name, email
FROM User
ORDER BY last_name, first_name
LIMIT 20;
```

q20. Family members alphabetically

This query lists family members in alphabetical order.

```SQL
SELECT member_id, first_name, last_name, relationship
FROM FamilyMember
ORDER BY last_name, first_name
LIMIT 20;
```

A.2 Advanced Queries (q21-q40)

q21. User and family members using JOIN

This query joins the `User` and `FamilyMember` tables to show each user together with their related family members.

```SQL
SELECT u.user_id,
       u.email,
       fm.member_id,
       CONCAT(fm.first_name, ' ', fm.last_name) AS member_name,
       fm.relationship
```

```
    FROM User u
    JOIN FamilyMember fm ON fm.user_id = u.user_id
    ORDER BY u.user_id DESC, fm.member_id DESC;
```

q22. Users without family members

This query identifies users who do not have any associated family member records.

```
SQL
    SELECT u.user_id, u.email
    FROM User u
    LEFT JOIN FamilyMember fm ON fm.user_id = u.user_id
    WHERE fm.member_id IS NULL;
```

q23. Family size per user

This query calculates the number of family members linked to each user and ranks users by family size.

```
SQL
    SELECT u.user_id,
           u.email,
           COUNT(fm.member_id) AS family_count
    FROM User u
    LEFT JOIN FamilyMember fm ON fm.user_id = u.user_id
    GROUP BY u.user_id, u.email
    ORDER BY family_count DESC;
```

q24. Health event report with member and condition details

This query joins health events with family members and condition information to produce a more detailed event report.

```SQL
SELECT he.event_id,
       CONCAT(fm.first_name, ' ', fm.last_name) AS family_member,
       hc.condition_name,
       he.severity,
       he.event_date
FROM HealthEvent he
JOIN FamilyMember fm ON he.member_id = fm.member_id
JOIN HealthCondition hc ON he.condition_id = hc.condition_id
ORDER BY he.event_date DESC
LIMIT 20;
```

q25. Events per condition

This query counts how many health events are associated with each condition.

```SQL
SELECT hc.condition_name,
       COUNT(*) AS total_events
FROM HealthEvent he
JOIN HealthCondition hc ON he.condition_id = hc.condition_id
GROUP BY hc.condition_name
ORDER BY total_events DESC;
```

q26. Conditions without events

This query identifies conditions that currently have no related health event records.

```sql
SELECT hc.condition_id, hc.condition_name

FROM HealthCondition hc

LEFT JOIN HealthEvent he ON he.condition_id = hc.condition_id

WHERE he.event_id IS NULL;
```

q27. Members with multiple events

This query finds family members who have more than one recorded health event.

```sql
SELECT he.member_id,

       COUNT(*) AS event_count

FROM HealthEvent he

GROUP BY he.member_id

HAVING COUNT(*) > 1

ORDER BY event_count DESC;
```

q28. Latest event per member

This query retrieves the most recent health event for each family member.

```sql
SELECT he.*

FROM HealthEvent he

JOIN (

  SELECT member_id, MAX(event_date) AS max_date
```

```
    FROM HealthEvent
    GROUP BY member_id
) m ON m.member_id = he.member_id
   AND m.max_date = he.event_date
ORDER BY he.event_date DESC;
```

q29. Medical history with details

This query joins medical history with family members and condition data to provide a detailed medical history report.

```
SQL
SELECT mh.event_id,
       CONCAT(fm.first_name, ' ', fm.last_name) AS member_name,
       hc.condition_name,
       mh.event_date,
       mh.diagnosis,
       mh.treatment
FROM MedicalHistory mh
JOIN FamilyMember fm ON mh.member_id = fm.member_id
JOIN HealthCondition hc ON mh.condition_id = hc.condition_id
ORDER BY mh.event_date DESC;
```

q30. Diagnoses per condition

This query counts how many medical history records exist for each condition.

```sql
SELECT hc.condition_name,
       COUNT(*) AS diagnoses
FROM MedicalHistory mh
JOIN HealthCondition hc ON mh.condition_id = hc.condition_id
GROUP BY hc.condition_name
ORDER BY diagnoses DESC;
```

q31. Members with medical history but no health events

This query identifies family members who have medical history records but no health event records.

```sql
SELECT DISTINCT mh.member_id
FROM MedicalHistory mh
WHERE NOT EXISTS (
  SELECT 1
  FROM HealthEvent he
  WHERE he.member_id = mh.member_id
);
```

q32. Members with health events but no medical history

This query identifies family members who have health event records but no medical history records.

```sql
SELECT DISTINCT he.member_id
FROM HealthEvent he
WHERE NOT EXISTS (
```

```
    SELECT 1
    FROM MedicalHistory mh
    WHERE mh.member_id = he.member_id
);
```

q33. Unresolved alerts per member

This query counts unresolved alerts for each family member.

```
SQL
SELECT member_id,
       COUNT(*) AS unresolved
FROM RiskAlert
WHERE status <> 'Resolved'
GROUP BY member_id
HAVING COUNT(*) >= 1
ORDER BY unresolved DESC;
```

q34. Alert work queue with priority sorting

This query produces an alert work queue by joining alerts with family member data and sorting the alerts by risk priority and date.

```
SQL
SELECT ra.alert_id,
       CONCAT(fm.first_name, ' ', fm.last_name) AS member_name,
       ra.risk_level,
       ra.status,
```

```
        ra.created_date
FROM RiskAlert ra
JOIN FamilyMember fm ON ra.member_id = fm.member_id
WHERE ra.status <> 'Resolved'
ORDER BY FIELD(ra.risk_level, 'High', 'Medium', 'Low'),
          ra.created_date DESC;
```

q35. Members with no alerts

This query identifies family members who have never received a risk alert.

```
SQL
SELECT fm.member_id, fm.first_name, fm.last_name
FROM FamilyMember fm
WHERE NOT EXISTS (
  SELECT 1
  FROM RiskAlert ra
  WHERE ra.member_id = fm.member_id
);
```

q36. Conditions above average event count

This query finds conditions whose number of events is higher than the average event count across all conditions.

```
SQL
SELECT hc.condition_name,
        COUNT(*) AS total_events
```

```
    FROM HealthEvent he
    JOIN HealthCondition hc ON hc.condition_id = he.condition_id
    GROUP BY hc.condition_name
    HAVING COUNT(*) > (
      SELECT AVG(cnt)
      FROM (
        SELECT COUNT(*) AS cnt
        FROM HealthEvent
        GROUP BY condition_id
      ) t
    )
    ORDER BY total_events DESC;
```

q37. High severity events with details

This query retrieves high severity events together with member and condition details.

```
SQL
    SELECT he.event_id,
           CONCAT(fm.first_name, ' ', fm.last_name) AS member_name,
           hc.condition_name,
           he.event_date
    FROM HealthEvent he
    JOIN FamilyMember fm ON fm.member_id = he.member_id
    JOIN HealthCondition hc ON hc.condition_id = he.condition_id
    WHERE he.severity = 'High'
    ORDER BY he.event_date DESC;
```

q38. Average onset age per condition

This query calculates the average onset age for each condition based on recorded health events.

```sql
SELECT hc.condition_name,
       AVG(he.onset_age) AS avg_onset_age
FROM HealthEvent he
JOIN HealthCondition hc ON hc.condition_id = he.condition_id
WHERE he.onset_age IS NOT NULL
GROUP BY hc.condition_name
ORDER BY avg_onset_age DESC;
```

q39. Members with high-risk alerts

This query identifies family members who have at least one alert marked as high risk.

```sql
SELECT fm.member_id, fm.first_name, fm.last_name
FROM FamilyMember fm
WHERE EXISTS (
  SELECT 1
  FROM RiskAlert ra
  WHERE ra.member_id = fm.member_id
    AND ra.risk_level = 'High'
);
```

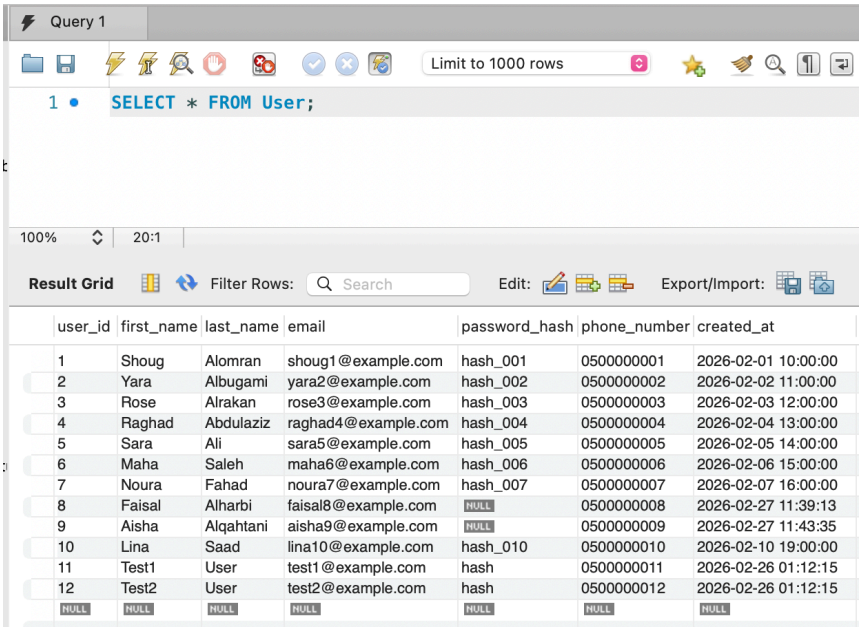
q40. Members with both events and alerts

This query identifies family members who have both recorded health events and risk alerts.

```sql
SELECT fm.member_id, fm.first_name, fm.last_name
FROM FamilyMember fm
WHERE EXISTS (
  SELECT 1
  FROM HealthEvent he
  WHERE he.member_id = fm.member_id
)
AND EXISTS (
  SELECT 1
  FROM RiskAlert ra
  WHERE ra.member_id = fm.member_id
);
```

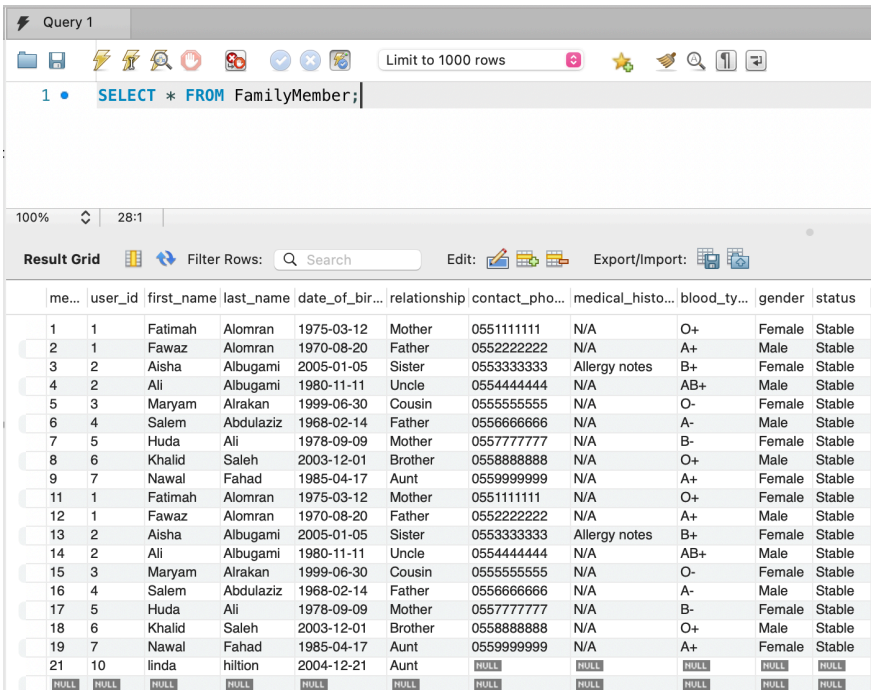
Appendix B:

Database records

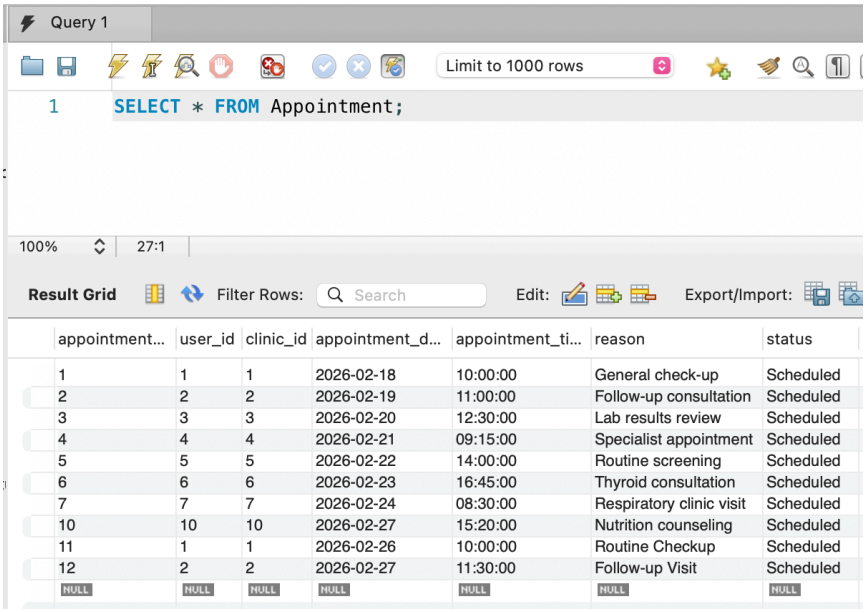


user_id	first_name	last_name	email	password_hash	phone_number	created_at
1	Shoug	Alomran	shoug1@example.com	hash_001	0500000001	2026-02-01 10:00:00
2	Yara	Albugami	yara2@example.com	hash_002	0500000002	2026-02-02 11:00:00
3	Rose	Alrakan	rose3@example.com	hash_003	0500000003	2026-02-03 12:00:00
4	Raghad	Abdulaziz	raghad4@example.com	hash_004	0500000004	2026-02-04 13:00:00
5	Sara	Ali	sara5@example.com	hash_005	0500000005	2026-02-05 14:00:00
6	Maha	Saleh	maha6@example.com	hash_006	0500000006	2026-02-06 15:00:00
7	Noura	Fahad	noura7@example.com	hash_007	0500000007	2026-02-07 16:00:00
8	Faisal	Alharbi	faisal8@example.com	NULL	0500000008	2026-02-27 11:39:13
9	Aisha	Alqahtani	aisha9@example.com	NULL	0500000009	2026-02-27 11:43:35
10	Lina	Saad	lina10@example.com	hash_010	0500000010	2026-02-10 19:00:00
11	Test1	User	test1@example.com	hash	0500000011	2026-02-26 01:12:15
12	Test2	User	test2@example.com	hash	0500000012	2026-02-26 01:12:15
NULL	NULL	NULL	NULL	NULL	NULL	NULL

Database records

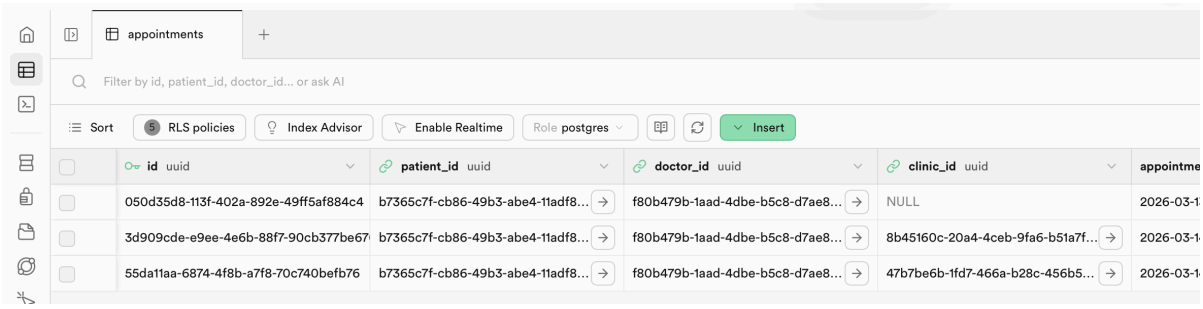


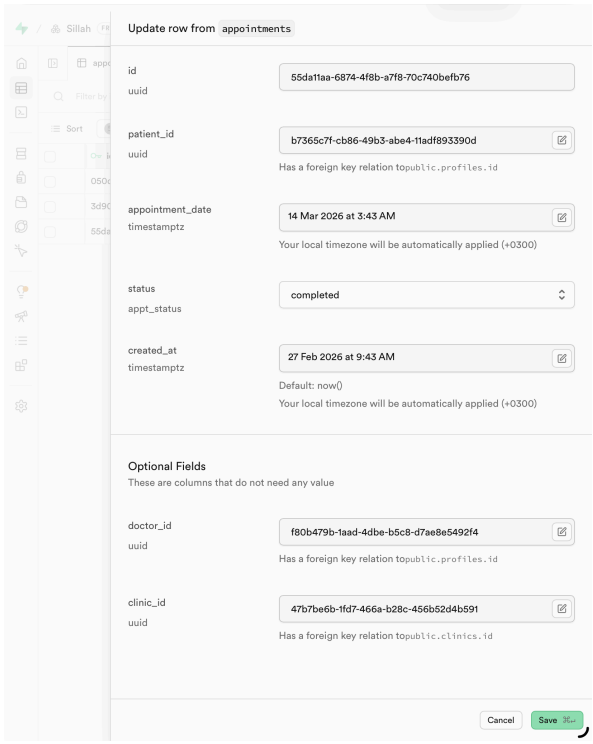
me...	user_id	first_name	last_name	date_of_bir...	relationship	contact_pho...	medical_histo...	blood_ty...	gender	status
1	1	Fatimah	Alomran	1975-03-12	Mother	0551111111	N/A	O+	Female	Stable
2	1	Fawaz	Alomran	1970-08-20	Father	0552222222	N/A	A+	Male	Stable
3	2	Aisha	Albugami	2005-01-05	Sister	0553333333	Allergy notes	B+	Female	Stable
4	2	Ali	Albugami	1980-11-11	Uncle	0554444444	N/A	AB+	Male	Stable
5	3	Maryam	Alrakan	1999-06-30	Cousin	0555555555	N/A	O-	Female	Stable
6	4	Salem	Abdulaziz	1968-02-14	Father	0556666666	N/A	A-	Male	Stable
7	5	Huda	Ali	1978-09-09	Mother	0557777777	N/A	B-	Female	Stable
8	6	Khalid	Saleh	2003-12-01	Brother	0558888888	N/A	O+	Male	Stable
9	7	Nawal	Fahad	1985-04-17	Aunt	0559999999	N/A	A+	Female	Stable
11	1	Fatimah	Alomran	1975-03-12	Mother	0551111111	N/A	O+	Female	Stable
12	1	Fawaz	Alomran	1970-08-20	Father	0552222222	N/A	A+	Male	Stable
13	2	Aisha	Albugami	2005-01-05	Sister	0553333333	Allergy notes	B+	Female	Stable
14	2	Ali	Albugami	1980-11-11	Uncle	0554444444	N/A	AB+	Male	Stable
15	3	Maryam	Alrakan	1999-06-30	Cousin	0555555555	N/A	O-	Female	Stable
16	4	Salem	Abdulaziz	1968-02-14	Father	0556666666	N/A	A-	Male	Stable
17	5	Huda	Ali	1978-09-09	Mother	0557777777	N/A	B-	Female	Stable
18	6	Khalid	Saleh	2003-12-01	Brother	0558888888	N/A	O+	Male	Stable
19	7	Nawal	Fahad	1985-04-17	Aunt	0559999999	N/A	A+	Female	Stable
21	10	linda	hiltion	2004-12-21	Aunt	NULL	NULL	NULL	NULL	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL



appointment...	user_id	clinic_id	appointment_d...	appointment_ti...	reason	status
1	1	1	2026-02-18	10:00:00	General check-up	Scheduled
2	2	2	2026-02-19	11:00:00	Follow-up consultation	Scheduled
3	3	3	2026-02-20	12:30:00	Lab results review	Scheduled
4	4	4	2026-02-21	09:15:00	Specialist appointment	Scheduled
5	5	5	2026-02-22	14:00:00	Routine screening	Scheduled
6	6	6	2026-02-23	16:45:00	Thyroid consultation	Scheduled
7	7	7	2026-02-24	08:30:00	Respiratory clinic visit	Scheduled
10	10	10	2026-02-27	15:20:00	Nutrition counseling	Scheduled
11	1	1	2026-02-26	10:00:00	Routine Checkup	Scheduled
12	2	2	2026-02-27	11:30:00	Follow-up Visit	Scheduled
NULL	NULL	NULL	NULL	NULL	NULL	NULL

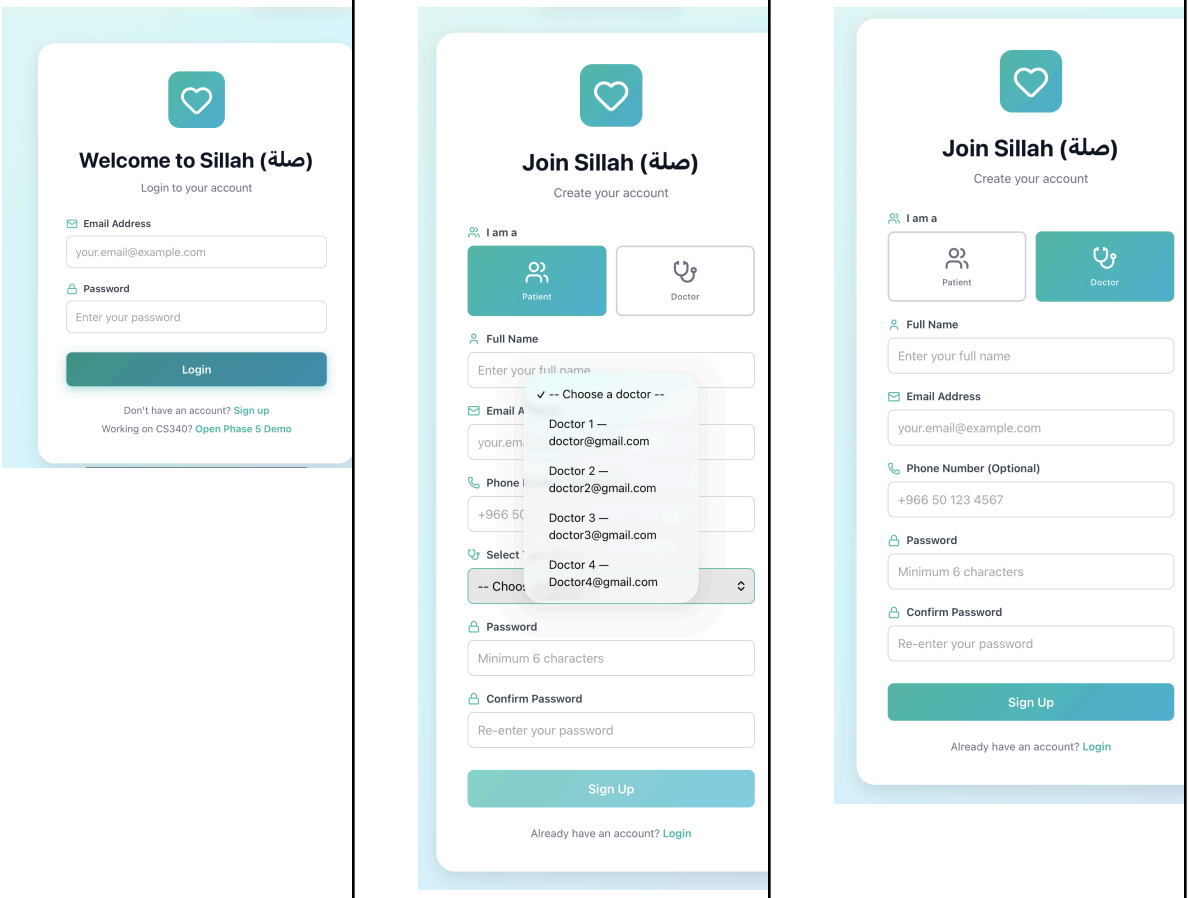
Appointments Table With Data





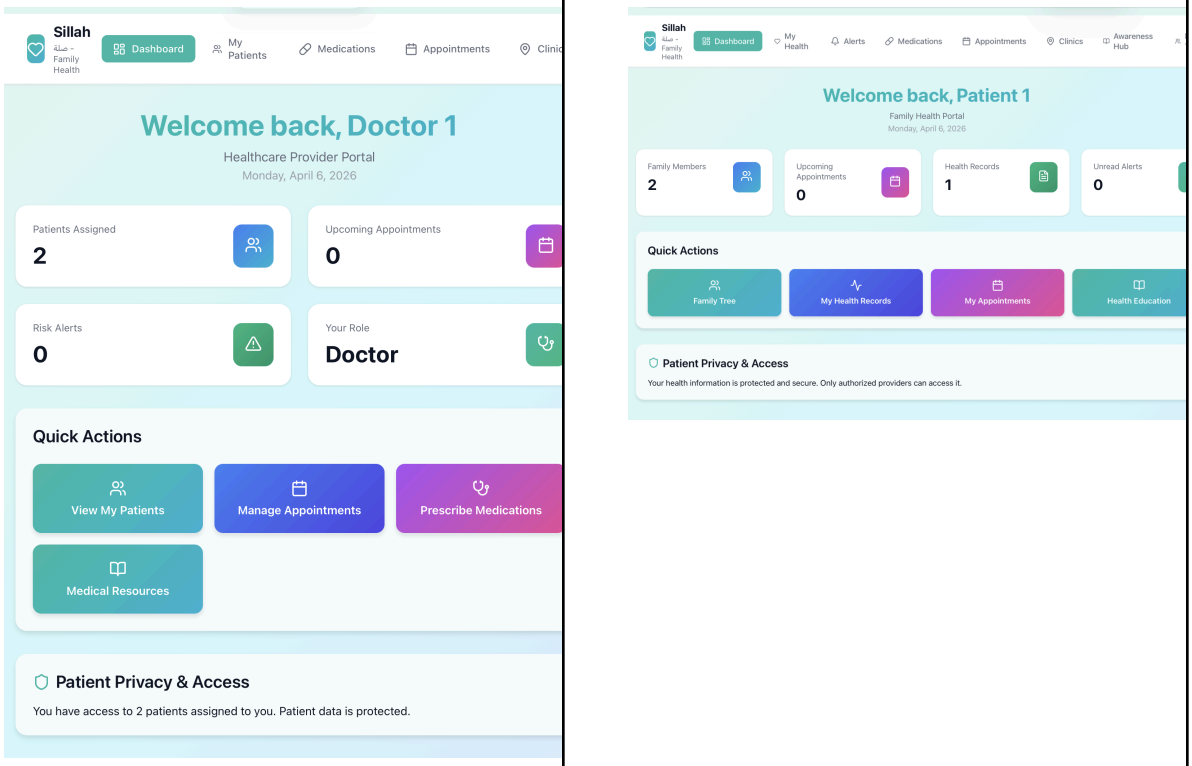
The following table demonstrates real appointment records stored in the Supabase application table, including relationships between patients, doctors, and clinics.

Login Page UI



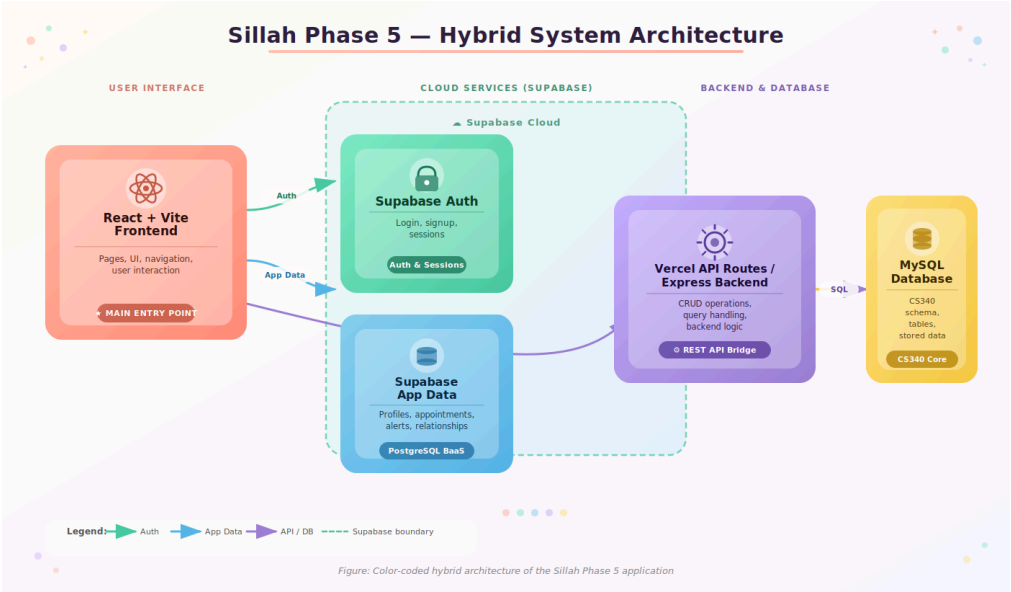
Login and signup interface demonstrating user authentication and role selection.

Dashboard UI



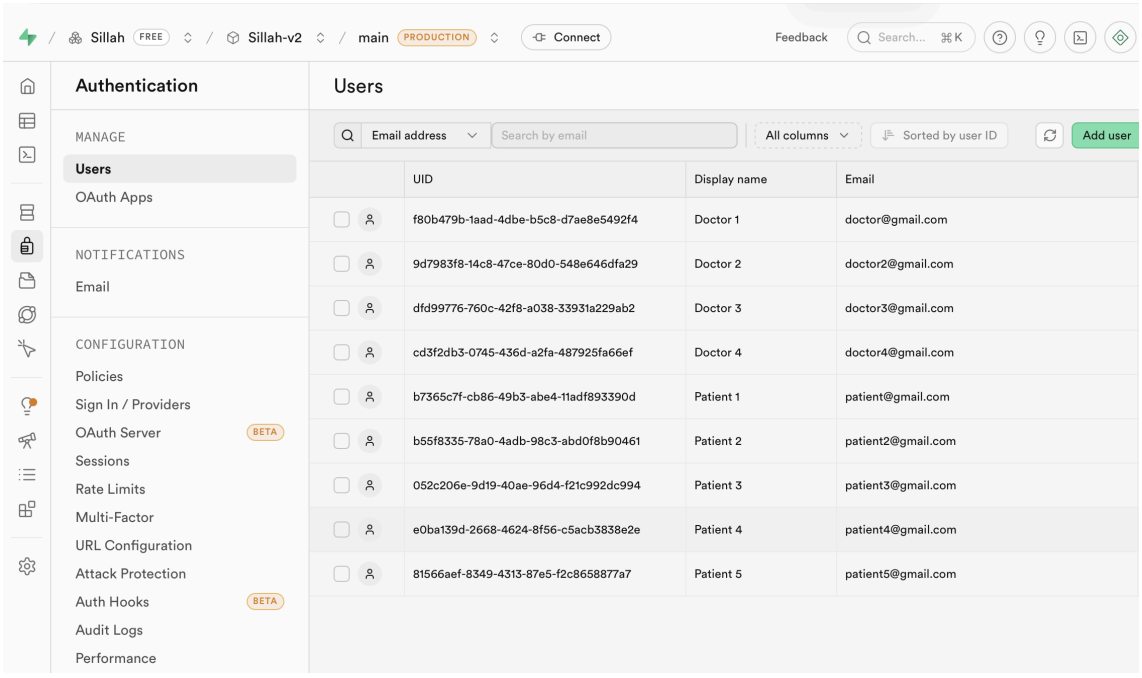
Dashboard interface for doctors and patients showing system overview and quick actions.

Architecture diagram



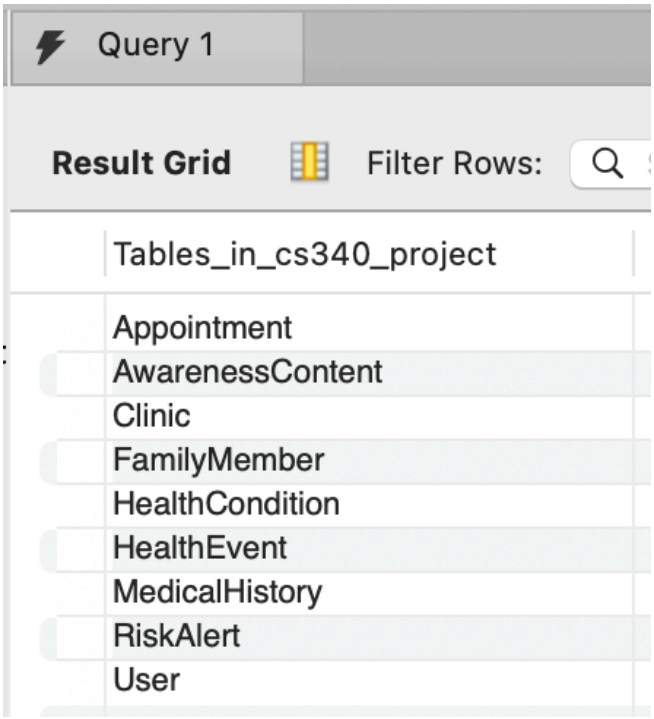
Hybrid system architecture showing React frontend, Supabase services, MySQL database, and API routes.

Supabase Authentication Users



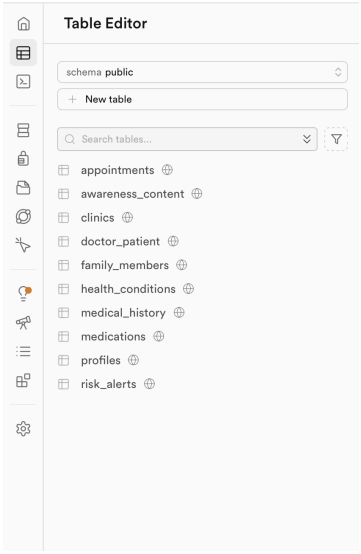
Supabase authentication dashboard displaying registered users and role-based accounts.

MySQL schema

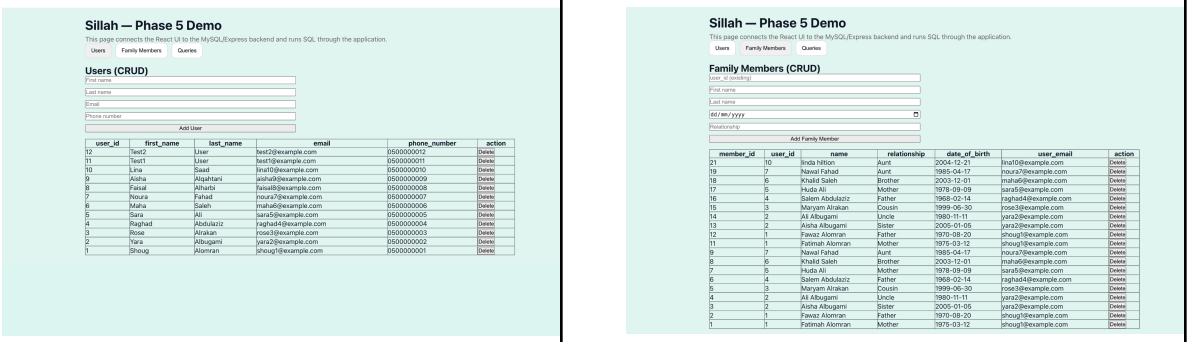


Main tables from the Phase 5 MySQL database schema.

Supabase Table Editor Overview

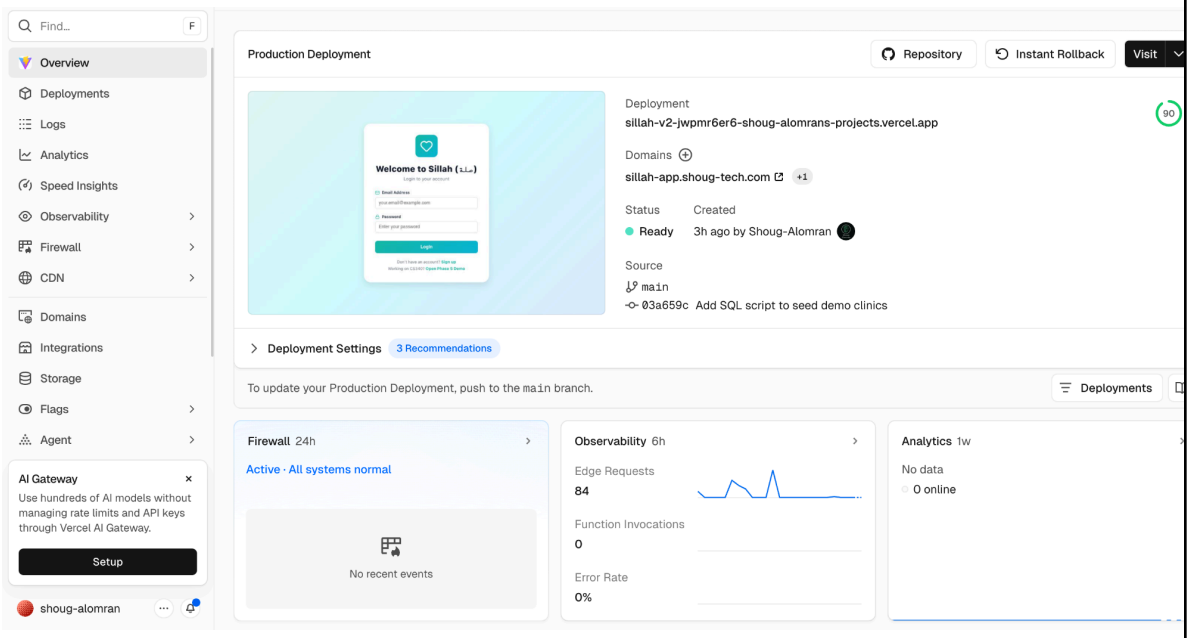


Phase 5 Demo

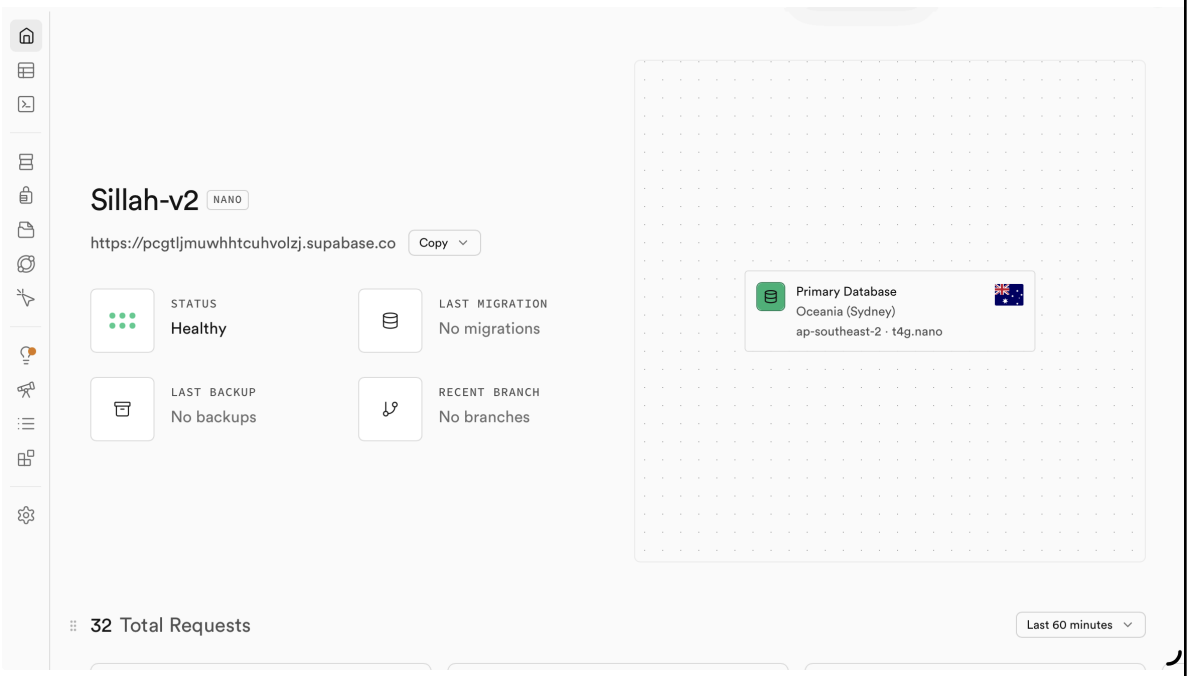


Phase 5 demo page showing frontend interaction with backend APIs.

Vercel Deployment



Frontend deployment on Vercel.

Project Status

Project deployment and service status.

Important Links
Phase 5 Demo
Documentation Site