



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.....	4
2. Final System Architecture.....	4
2.1 Main Prototype Application.....	4
Technology Stack:.....	5
2.2 CS340 Phase 5 Demo Layer.....	5
Features:.....	5
Backend:.....	5
API Endpoints:.....	5
2.3 Data Layer.....	6
Supabase Responsibilities:.....	6
MySQL Responsibilities:.....	6
3. Authentication Implementation.....	7
Login Implementation:.....	7
Route Protection:.....	7
4. Database Implementation.....	8
4.1 MySQL Core Relational Tables.....	8
4.2 Supabase Application Support Tables.....	8
4.3 Appointment Functionality Across Both Layers.....	8
4.4 Family Members and Health Features.....	9
5. Frontend-Backend Integration.....	9
6. Code Samples.....	10
6.1 Supabase Client Configuration.....	10
6.2 Authentication Login.....	10
6.3 Protected Route.....	10
6.4 Fetching Data Through the API.....	10
6.5 Inserting Data Through the API.....	10
6.6 Updating Data Through the API.....	11
6.7 SQL Query Execution Through the Application.....	11
7. CRUD and SQL Query Execution.....	11
7.1 Query Usage within the Application.....	12
8. Query Distribution by Team Member.....	12
9. Deployment.....	12
10. Testing and Validation.....	13
11. Performance Considerations.....	13
12. Sustainability Reflection.....	13
13. Individual Contributions.....	13
14. Conclusion.....	13
Appendix A: SQL Queries Implementation (q01-q40).....	14

A.1 Basic Queries (q01-q20).....	14
A.2 Advanced Queries (q21-q40).....	20
Appendix B:.....	30
Database records.....	30
Appointments Table With Data.....	32
Login Page UI.....	33
Dashboard UI.....	34
Architecture diagram.....	35
Supabase Authentication Users.....	35
MySQL schema.....	36
Supabase Table Editor Overview.....	36
Phase 5 Demo.....	37
Vercel Deployment.....	37
Project Status.....	38
Important Links.....	38

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
- Appointment management
- Dashboard interfaces
- 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
- Joins
- Filtering
- 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
- Authentication testing
- API endpoint 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
- Query implementation
- Frontend development
- 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
- Academic integrity
- Practical functionality

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

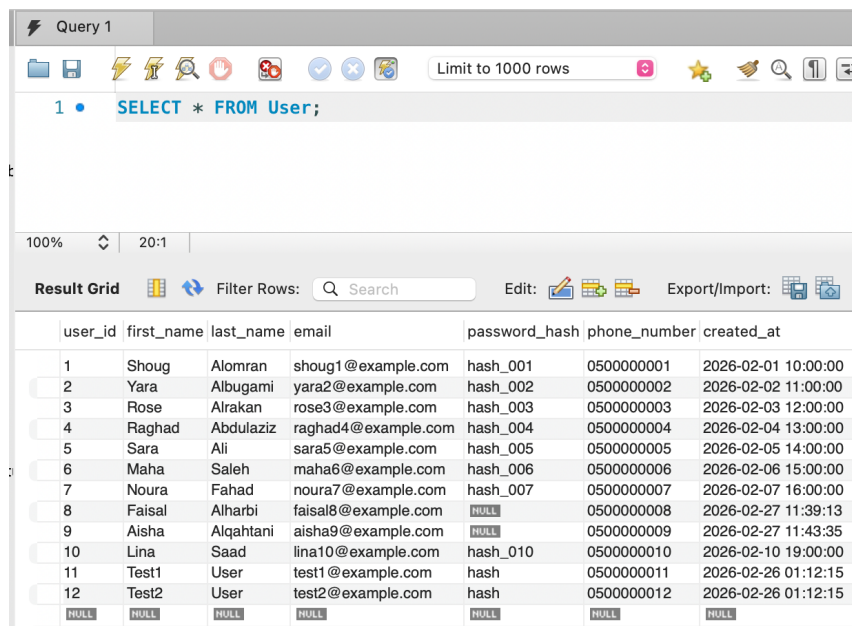
    FROM RiskAlert ra

    WHERE ra.member_id = fm.member_id

);
```

Appendix B:

Database records



The screenshot shows a database query tool interface. At the top, there's a toolbar with various icons and a 'Limit to 1000 rows' dropdown. Below the toolbar, the query editor shows a single query: `1 • SELECT * FROM User;`. The results are displayed in a 'Result Grid' below the query editor. The grid has columns for `user_id`, `first_name`, `last_name`, `email`, `password_hash`, `phone_number`, and `created_at`. The data is presented in a table with 12 rows. The first 10 rows show user records with names like Shoug, Yara, Rose, Raghad, Sara, Maha, Noura, Faisal, Aisha, and Lina. The last two rows are labeled 'Test1' and 'Test2'. The 'password_hash' column contains values like 'hash_001' through 'hash_010', and 'hash' for the test users. The 'phone_number' column contains values like '0500000001' through '0500000012'. The 'created_at' column shows timestamps. The bottom of the grid shows some null values for the last few rows.

	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

Query 1

Limit to 1000 rows

1 • **SELECT * FROM FamilyMember;**

100% 28:1

Result Grid Filter Rows: Search Edit: Export/Import:

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

Query 1

Limit to 1000 rows

1 • **SELECT * FROM Appointment;**

100% 27:1

Result Grid Filter Rows: Search Edit: Export/Import:

	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
13		NULL	NULL	NULL	NULL	NULL	NULL

Appointments Table With Data

appointments +						
Filter by id, patient_id, doctor_id... or ask AI						
Sort RLS policies Index Advisor Enable Realtime Role postgres Insert						
	id uuid	patient_id uuid	doctor_id uuid	clinic_id uuid	appointment_date timestamp	
	050d35d8-113f-402a-892e-49ff5af884c4	b7365c7f-cb86-49b3-abe4-11adf8...	f80b479b-1aad-4dbe-b5c8-d7ae8...	NULL	2026-03-1	
	3d909cde-e9ee-4e6b-88f7-90cb377be67	b7365c7f-cb86-49b3-abe4-11adf8...	f80b479b-1aad-4dbe-b5c8-d7ae8...	8b45160c-20a4-4ceb-9fa6-b51a7f...	2026-03-1	
	55da11aa-6874-4f8b-a7f8-70c740befb76	b7365c7f-cb86-49b3-abe4-11adf8...	f80b479b-1aad-4dbe-b5c8-d7ae8...	47b7be6b-1fd7-466a-b28c-456b5...	2026-03-1	

Update row from appointments

id
uuid

55da11aa-6874-4f8b-a7f8-70c740befb76

patient_id
uuid

b7365c7f-cb86-49b3-abe4-11adf893390d

Has a foreign key relation to public.profiles.id

appointment_date
timestamp

14 Mar 2026 at 3:43 AM

Your local timezone will be automatically applied (+0300)

status
appt_status

completed

created_at
timestamp

27 Feb 2026 at 9:43 AM

Default: now()
Your local timezone will be automatically applied (+0300)

Optional Fields

These are columns that do not need any value

doctor_id
uuid

f80b479b-1aad-4dbe-b5c8-d7ae8e5492f4

Has a foreign key relation to public.profiles.id

clinic_id
uuid

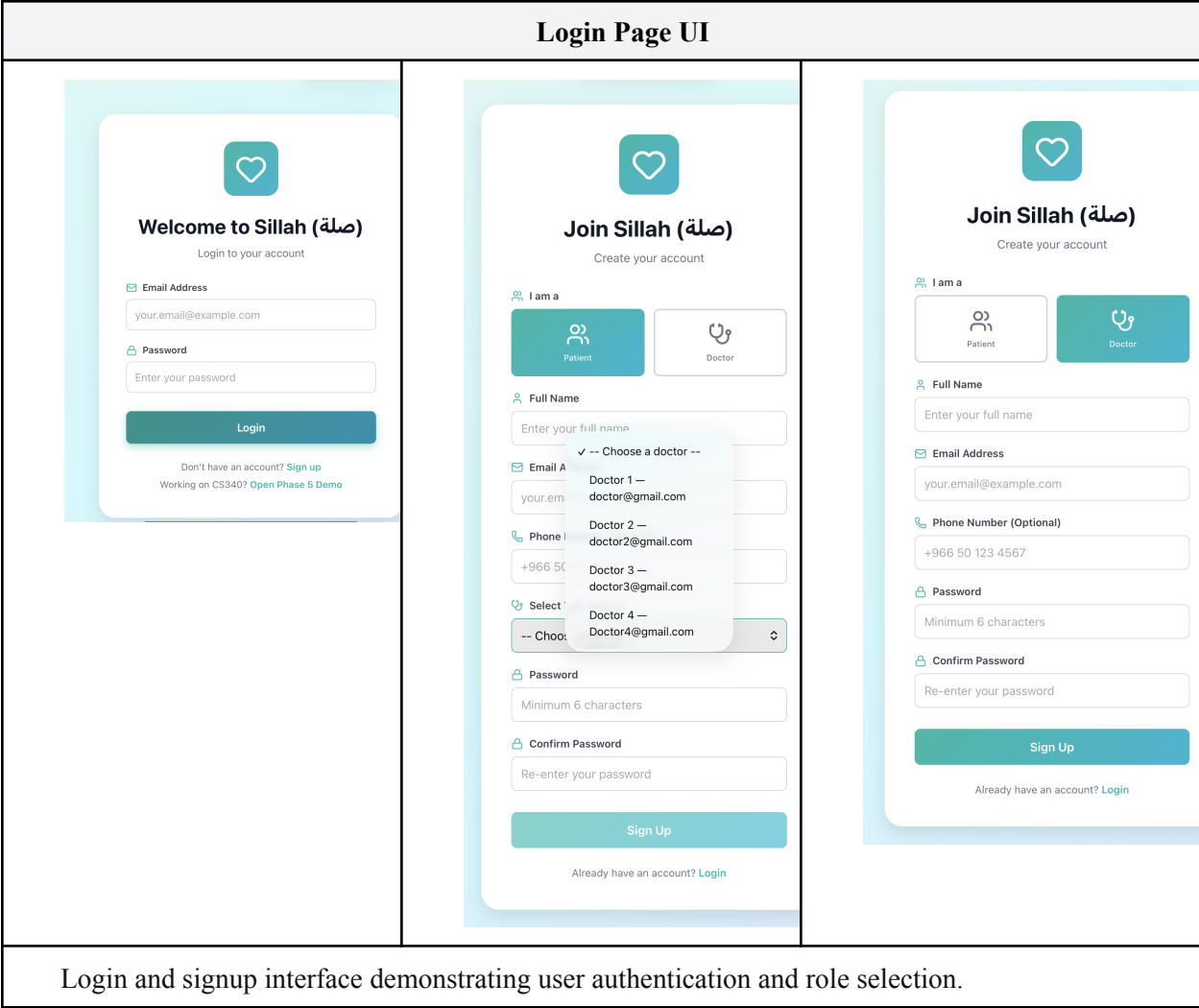
47b7be6b-1fd7-466a-b28c-456b52d4b591

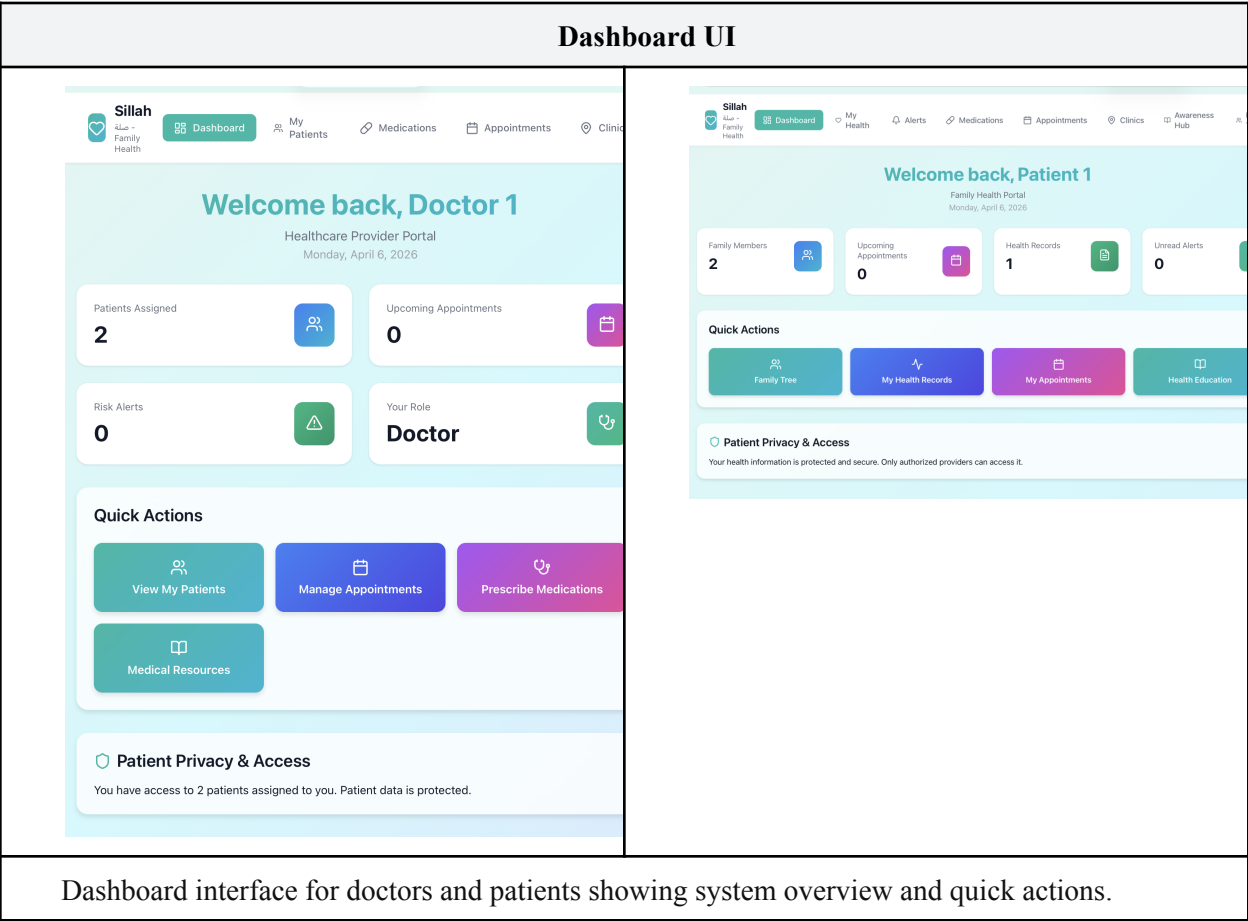
Has a foreign key relation to public.clinics.id

Cancel

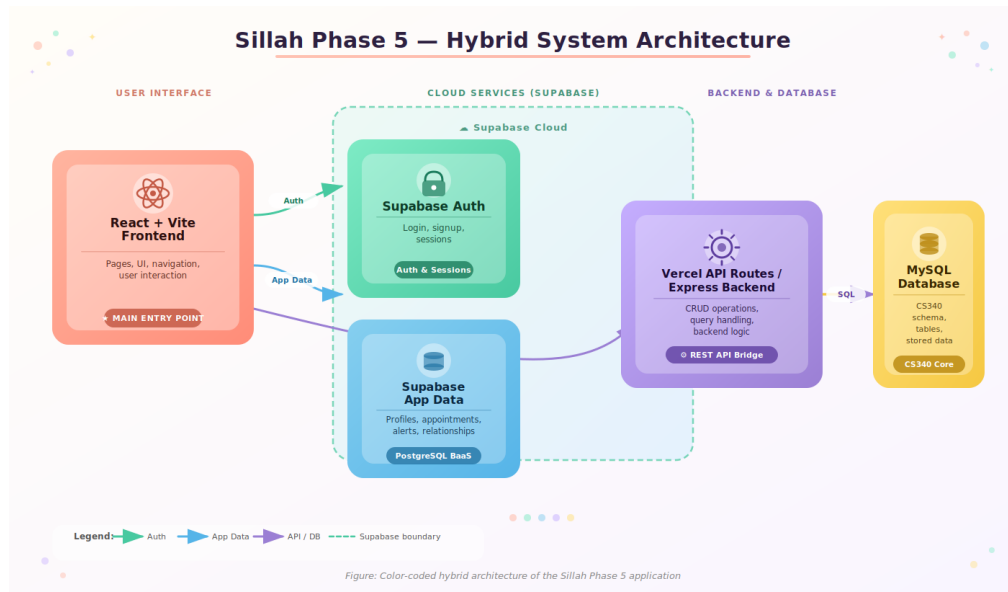
Save

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





Architecture diagram



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

Supabase Authentication Users

Sillah FREE / Sillah-v2 / main PRODUCTION Connect Feedback Search... 36 K ? ? ? ?

Authentication	Users
MANAGE	Email address Search by email All columns Sorted by user ID Add user
Users	
OAuth Apps	
NOTIFICATIONS	
Email	
CONFIGURATION	
Policies	
Sign In / Providers	
OAuth Server	
Sessions	
Rate Limits	
Multi-Factor	
URL Configuration	
Attack Protection	
Auth Hooks	
Audit Logs	
Performance	

	UID	Display name	Email
☐	f80b479b-1aad-4dbe-b5c8-d7ae8e5492f4	Doctor 1	doctor@gmail.com
☐	9d7983f8-14c8-47ce-80d0-548e646dfa29	Doctor 2	doctor2@gmail.com
☐	dfd99776-760c-42f8-a038-33931a229ab2	Doctor 3	doctor3@gmail.com
☐	cd3f2db3-0745-436d-a2fa-487925fa66ef	Doctor 4	doctor4@gmail.com
☐	b7365c7f-cb86-49b3-abe4-11ad7f893390d	Patient 1	patient@gmail.com
☐	b55f8335-78a0-4adb-98c3-abd0f8b90461	Patient 2	patient2@gmail.com
☐	052c206e-9df9-40ae-96d4-f21c992dc994	Patient 3	patient3@gmail.com
☐	e0bet139d-2668-4624-8f56-c5acb3838e2e	Patient 4	patient4@gmail.com
☐	81566aef-8349-4313-87e5-f2c8658877a7	Patient 5	patient5@gmail.com

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

MySQL schema

⚡ Query 1

Result Grid



Filter Rows:

Tables_in_cs340_project

Appointment

AwarenessContent

Clinic

FamilyMember

HealthCondition

HealthEvent

MedicalHistory

RiskAlert

User

Main tables from the Phase 5 MySQL database schema.

Supabase Table Editor Overview

Table Editor

schema public

+ New table

appointments

awareness_content

clinics

doctor_patient

family_members

health_conditions

medical_history

medications

profiles

risk_alerts

36

Phase 5 Demo

Sillah -- Phase 5 Demo

This page connects the React UI to the MySQL/Express backend and runs SQL through the application.

Users **Family Members** Clinics

Users (CRUD)

Form fields: first name, last name, email, phone number.

Buttons: Add User, Edit User, Delete User.

user_id	first_name	last_name	email	phone_number	action
12	Test2	User	test2@example.com	0500000012	Delete
11	Test1	User	test1@example.com	0500000011	Delete
10	Lina	Isaak	lina10@example.com	0500000010	Delete
9	Khadi	Alghafar	khadi9@example.com	0500000009	Delete
8	Faisal	Ahmed	faisal8@example.com	0500000008	Delete
7	Noura	Fahad	noura7@example.com	0500000007	Delete
6	Maha	Saleh	maha6@example.com	0500000006	Delete
5	Sara	Ali	sara5@example.com	0500000005	Delete
4	Raghad	Abdulaziz	raghad4@example.com	0500000004	Delete
3	Rose	Alrakan	rose3@example.com	0500000003	Delete
2	Yara	Albugami	yara2@example.com	0500000002	Delete
1	Shoug	Alomran	shoug1@example.com	0500000001	Delete

Sillah -- Phase 5 Demo

This page connects the React UI to the MySQL/Express backend and runs SQL through the application.

Users **Family Members** Clinics

Family Members (CRUD)

Form fields: first name, last name, email, date of birth, relationship.

Buttons: Add Family Member, Edit Family Member, Delete Family Member.

member_id	user_id	name	relationship	date_of_birth	user_email	action
21	10	Inda Hishm	Aunt	2004-12-01	lina10@example.com	Delete
19	7	Nawal Fahad	Aunt	1985-04-17	noura7@example.com	Delete
18	6	Khadi Saleh	Mother	2002-12-01	maha6@example.com	Delete
17	6	Khadi Ali	Mother	1978-09-09	sara5@example.com	Delete
16	4	Salim Abdulaziz	Father	1968-02-14	raghad4@example.com	Delete
15	3	Maryam Alrakan	Cousin	1999-06-30	yara2@example.com	Delete
14	2	Al Abugami	Uncle	1980-11-11	yara2@example.com	Delete
13	2	Khadi Albugami	Sister	2005-01-05	yara2@example.com	Delete
12	1	Fawaz Alomran	Father	1970-08-20	shoug1@example.com	Delete
11	1	Fatemah Alomran	Mother	1975-03-12	shoug1@example.com	Delete
9	7	Nawal Fahad	Aunt	1985-04-17	noura7@example.com	Delete
8	6	Khadi Saleh	Sister	2002-12-01	maha6@example.com	Delete
7	6	Khadi Ali	Mother	1978-09-09	sara5@example.com	Delete
6	4	Salim Abdulaziz	Father	1968-02-14	raghad4@example.com	Delete
5	3	Maryam Alrakan	Cousin	1999-06-30	yara2@example.com	Delete
4	2	Al Abugami	Uncle	1980-11-11	yara2@example.com	Delete
3	2	Khadi Albugami	Sister	2005-01-05	yara2@example.com	Delete
2	1	Fawaz Alomran	Father	1970-08-20	shoug1@example.com	Delete
1	1	Fatemah Alomran	Mother	1975-03-12	shoug1@example.com	Delete

Phase 5 demo page showing frontend interaction with backend APIs.

Vercel Deployment

Find...

Overview

Deployments

Logs

Analytics

Speed Insights

Observability

Firewall

CDN

Domains

Integrations

Storage

Flags

Agent

AI Gateway

shoug-alomran

Production Deployment

Repository

Instant Rollback

Visit

Deployment

sillah-v2-jwpmr6er6-shoug-alomrans-projects.vercel.app

Domains

sillah-app.shoug-tech.com

Status

Created

Ready

3h ago by Shoug-Alomran

Source

main

03a659c Add SQL script to seed demo clinics

Deployment Settings

3 Recommendations

To update your Production Deployment, push to the main branch.

Deployments

Firewall 24h

Active · All systems normal

No recent events

Observability 6h

Edge Requests

84

Function Invocations

0

Error Rate

0%

Analytics 1w

No data

0 online

Frontend deployment on Vercel.

Project Status

Sillah-v2

NANO

https://pcgtljmuwhhtcuholzj.supabase.co

Copy

STATUS

Healthy

LAST MIGRATION

No migrations

LAST BACKUP

No backups

RECENT BRANCH

No branches

Primary Database

Oceania (Sydney)

ap-southeast-2 • t4g.nano

32 Total Requests

Last 60 minutes

Project deployment and service status.

Important Links
Phase 5 Demo
Documentation Site

38