Steps to Create a New Project Based on the Transcript

1. Create a Supabase Account

- Open Supabase.
- Sign up for a free account using your **GitHub account** or email.
- (Optional) Explore the **Pricing Page** to understand the free tier limitations.

2. Set Up an Organization

- During the signup process, create an organization. For example, use a name like "YourName Organization."
- If you skipped this during signup, go to the dashboard and create one.

3. Create a New Project

- Navigate to the Supabase dashboard.
- Click the **"Create New Project"** button.
- Fill in the required fields:
  - **Project Name**: Example: *The Wild Oasis*.
  - **Database Password**: Generate a secure password and store it in a safe location (e.g., a password manager, not in your codebase).
  - **Region**: Choose the region closest to your primary user base.
- Select the **Free Plan**.

4. Wait for Project Setup

- Supabase will now initialize your project and database.
- Once the setup is complete, you will have access to:
  - **Project URL**
  - **JavaScript Client Documentation**

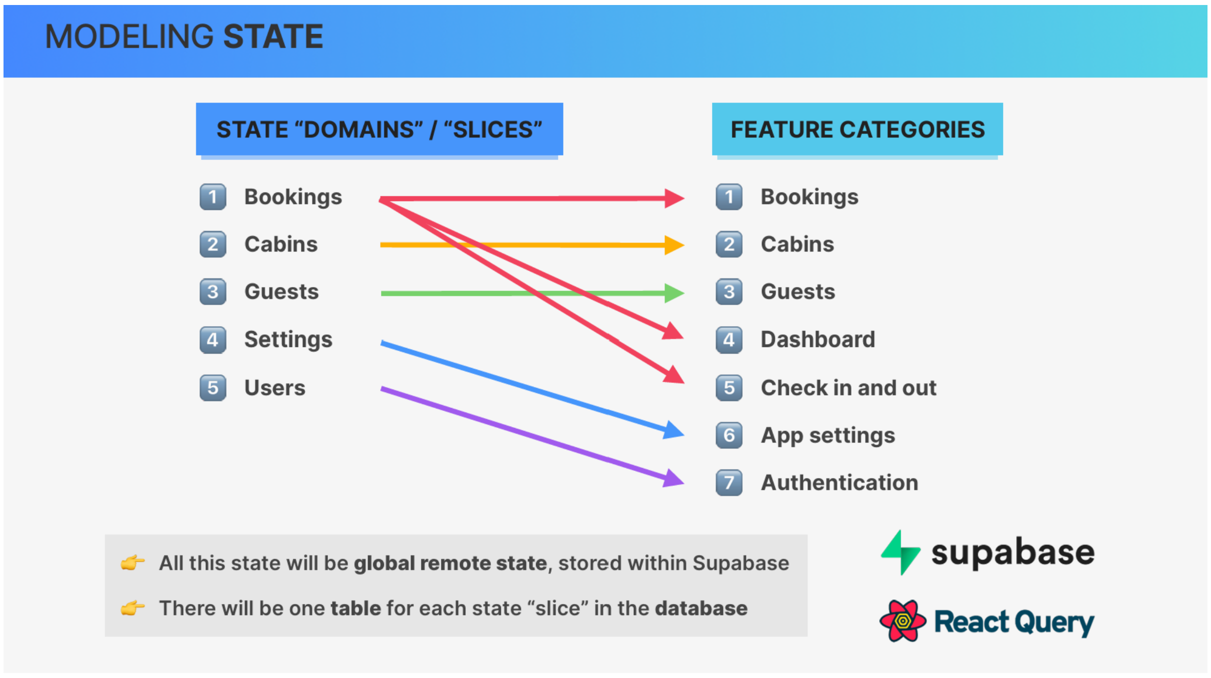
5. Explore Supabase Features

- Familiarize yourself with the key areas in the dashboard:
  - **Table Editor**: Create tables and add data.
  - **Authentication**: Set up user authentication.
  - **Storage**: Upload files (e.g., PDFs, images).
  - **API Documentation**: Learn how to use the Supabase API.

    - **Project Settings**: Adjust configurations as needed.

6. Plan the Application Data Model

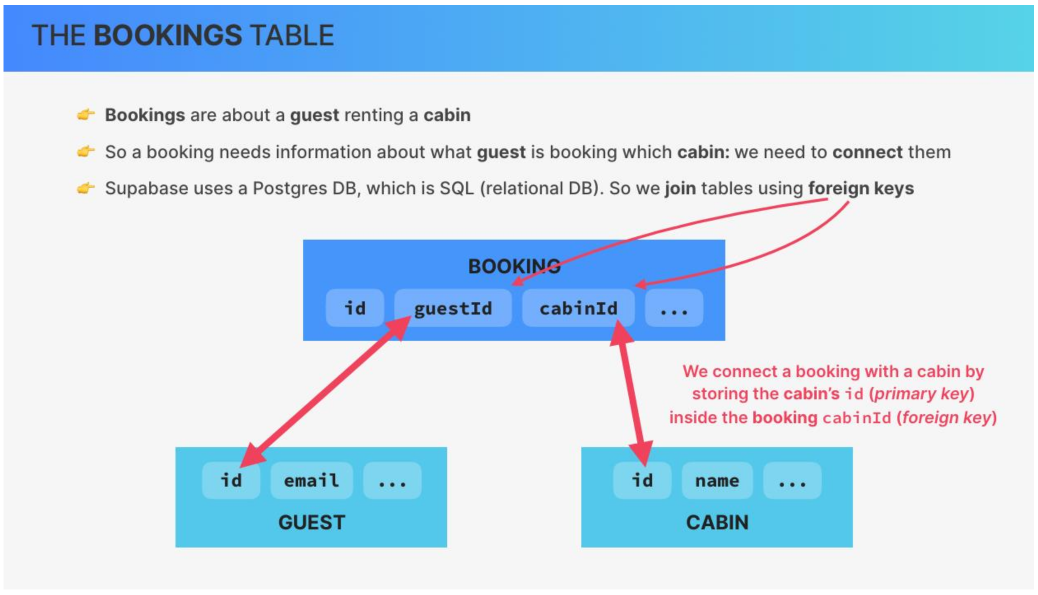
- Think about the application state and data requirements.
- Identify the tables, fields, and relationships needed for your app.
- For example, for *The Wild Oasis*, you might need tables like:
    - **Users**: For authentication.
    - **Cabins**: To store cabin details.
    - **Bookings**: To manage reservations



MODELING STATE

STATE "DOMAINS" / "SLICES"	FEATURE CATEGORIES
1 Bookings	1 Bookings
2 Cabins	2 Cabins
3 Guests	3 Guests
4 Settings	4 Dashboard
5 Users	5 Check in and out
	6 App settings
	7 Authentication

- 👉 All this state will be **global remote state**, stored within Supabase
- 👉 There will be one **table** for each state "slice" in the **database**



THE BOOKINGS TABLE

- 👉 **Bookings** are about a **guest** renting a **cabin**
- 👉 So a booking needs information about what **guest** is booking which **cabin:** we need to **connect** them
- 👉 Supabase uses a Postgres DB, which is SQL (relational DB). So we **join** tables using **foreign keys**

BOOKING
id
guestId
cabinId
...
id
email
...
GUEST
id
name
...
CABIN
We connect a booking with a cabin by storing the cabin's id (primary key) inside the booking cabinId (foreign key)

For relational data, use **foreign keys** to connect tables:

- **Bookings Table**:
  - guest_id: A foreign key referencing the **Guests Table**.
  - cabin_id: A foreign key referencing the **Cabins Table**.

1. Cabins Table

- **Name**: cabins
- **Row-Level Security**: Enable it.
- **Columns**:
    - id (default): Primary key, auto-generated.
    - created_at (default): Timestamp of row creation.
    - name (string): Name of the cabin (e.g., "001").
    - max_capacity (integer): Maximum number of occupants per cabin.
    - regular_price (integer): Regular price of the cabin.
    - discount (integer): Discounted amount (if any).
    - description (text): Description of the cabin.
    - image (text): URL of the cabin image stored in a Supabase bucket.
- **Insert Example Row**:

plaintext

CopyEdit

name: "001"

max_capacity: 2

regular_price: 250

discount: 50

description: "Small luxury cabin in the woods"

image: ""

2. Guests Table

- **Name**: guests
- **Row-Level Security**: Enable it.
- **Columns**:
    - id (default): Primary key, auto-generated.
    - created_at (default): Timestamp of row creation.
    - full_name (text): Guest's full name.

    - email (text): Email address of the guest.
    - nationality (text): Guest's nationality.
    - country_flag (text): Emoji or code representing the guest's country flag.
    - national_id (text): National ID number of the guest.
- **Insert Example Row**:

plaintext

CopyEdit

full_name: "Test User"

email: "test@example.com"

nationality: "German"

country_flag: ""

national_id: ""

3. Settings Table

- **Name**: settings
- **Row-Level Security**: Enable it.
- **Columns**:
    - id (default): Primary key, auto-generated.
    - created_at (default): Timestamp of row creation.
    - min_booking_length (integer): Minimum booking length in days.
    - max_booking_length (integer): Maximum booking length in days.
    - max_guests_per_booking (integer): Maximum number of guests per booking.
    - breakfast_price (float): Price of breakfast per guest.
- **Insert Example Row**:

plaintext

CopyEdit

min_booking_length: 3

max_booking_length: 90

max_guests_per_booking: 8

breakfast_price: 15.00

Steps to Create a bookings Table with Relationships:

1. **Create the Table**:
   - Name the table bookings.
2. **Add Columns**:
   - start_date: A timestamp to store the start date of the booking.
   - end_date: A timestamp to store the end date of the booking.
   - nights: An integer to store the number of nights for the booking (calculated automatically but stored for convenience).
   - guests: An integer to store the number of guests.
   - cabin_price: A float to store the price for the cabin.
   - extras_price: A float to store the price for extras (e.g., breakfast).
   - total_price: A float to store the total price (sum of cabin_price and extras_price).
   - status: A text column to store the booking status (e.g., "checked in", "checked out", "unconfirmed").
   - has_breakfast: A boolean to indicate whether breakfast is included.
   - is_paid: A boolean to indicate if the booking has been paid.
   - observations: A text column to store additional notes or observations.
3. **Establish Relationships**:
   - cabin_id: Add a foreign key column referencing the ID column of the cabins table to link a booking to a specific cabin.
   - guest_id: Add a foreign key column referencing the ID column of the guests table to link a booking to a specific guest.
4. **Add Foreign Key Constraints**:
   - Define cabin_id as a foreign key referencing the ID column in the cabins table.
   - Define guest_id as a foreign key referencing the ID column in the guests table.
5. **Populate the Table**:
   - Insert a sample row:

        - start_date: E.g., "2025-01-29".
        - end_date: E.g., "2025-02-02".
        - nights: 4.
        - guests: 2.
        - cabin_price: 300.
        - extras_price: 120 (e.g., breakfast for 2 guests at 15 each for 4 nights).
        - total_price: 420.
        - status: "unconfirmed".
        - has_breakfast: true.
        - is_paid: true.
        - observations: E.g., "Arrival at 10:00 PM."
        - cabin_id: Reference the ID of the selected cabin (e.g., 1).
        - guest_id: Reference the ID of the selected guest (e.g., 1).

6. **Understand Relationships**:
    - Each booking is linked to exactly one cabin (cabin_id) and one guest (guest_id).
    - A cabin can have many bookings, but each booking is specific to one cabin.
    - Similarly, a guest can make many bookings, but each booking is specific to one guest.

Steps to Add Security Policies in Supabase Based on the Transcript:

1. **Access API Documentation:**
   - Navigate to the **API Docs** section in Supabase.
   - Review the documentation for your table (e.g., "cabins") to understand how to access data via the API.
2. **Enable Row-Level Security (RLS):**
   - Confirm that **Row-Level Security (RLS)** is enabled for the table. This ensures that the database is secure by default, preventing unauthorized operations even with the API key.
3. **Verify Access Attempt:**
   - Test access to the data using a tool like curl or a REST client.
   - Use the project's **public API key** (anon key) for the request.
   - If no data is returned (e.g., an empty array), it indicates that RLS is correctly restricting access.
4. **Set Up a New Policy:**
   - Go to the **Authentication > Policies** section in Supabase.
   - Select the table for which you want to configure a policy (e.g., "cabins").
5. **Create a Policy from a Template:**
   - Click **Create a New Policy** and choose **Get Started Quickly** to use predefined templates.
   - Select the appropriate template for your use case. For example:
     - To allow read access for everyone, use the "Enable read access for everyone" template.
     - For stricter control, choose templates that restrict access to authenticated users only.
6. **Review and Save the Policy:**
   - Review the policy settings generated by the template.
   - Click **Review and Save** to apply the policy.
7. **Test the Updated Access:**

   - Re-run your API request (e.g., using curl) to confirm that the data is now accessible according to the new policy.
   - If the policy is for read access, you should see the data for the table.
8. **Future Enhancements:**
   - Consider refining policies to allow more specific access:
     - Allow only authenticated users to access the data.
     - Restrict write or delete operations to admin users or specific roles.
   - Update policies based on application requirements.
9. **Implement in Your Application:**
   - Use the updated API access in your application (e.g., React app) to fetch data securely, ensuring compliance with the defined policies.

This ensures that your data is secure while allowing the necessary level of access for your application

step-by-step guide to connect Supabase with your app

1. Install the Supabase JavaScript Library

Open your terminal and navigate to your project directory. Run the following command to install the Supabase client library:

bash

CopyEdit

```
npm install @supabase/supabase-js
```

---

2. Initialize Supabase

Create a new file to configure your Supabase client. Typically, this file is named supabase.js or supabaseClient.js.

Example:

javascript

CopyEdit

```
// supabaseClient.js
```

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


const supabaseUrl = 'https://hqrfpjhnhqvexdjovvkf.supabase.co'; // Replace with your Supabase URL
const supabaseKey = 'YOUR_SUPABASE_ANON_KEY'; // Replace with your anon key


const supabase = createClient(supabaseUrl, supabaseKey);


export default supabase;
```

Note: You can find your Supabase URL and anon key in the **Supabase dashboard** under **Project Settings > API**.

---

3. Create Services for Your Tables

For each database table, create a service file to manage API queries. For example, create a file apiCabins.js for the Cabins table.

Example:

javascript

CopyEdit

```
import supabase from './supabaseClient';


// Function to fetch cabins
export async function getCabins() {
  const { data, error } = await supabase.from('Cabins').select('*');


  if (error) {
    console.error('Cabins could not be loaded:', error);
    throw new Error(error.message);
  }
  return data;
}
```

---

4. Use the Service in Your React Component

To fetch data and display it in a React component, import the service and use it, for example, in a useEffect hook.

Example:

javascript

CopyEdit

```
import React, { useEffect, useState } from 'react';
import { getCabins } from './apiCabins';

function CabinsPage() {
  const [cabins, setCabins] = useState([]);

  useEffect(() => {
    async function fetchCabins() {
      try {
        const data = await getCabins();
        setCabins(data);
      } catch (error) {
        console.error(error);
      }
    }

    fetchCabins();
  }, []);
  return (
    <div>
      <h1>Cabins</h1>
      <ul>
        {cabins.map((cabin) => (
          <li key={cabin.id}>{cabin.name}</li>
        ))}
      </ul>
    </div>
  );
}

export default CabinsPage;
```

---

5. Test the Connection

- Start your React app:

bash

CopyEdit

```
npm start
```

- Visit the page/component where you’ve implemented the data fetching logic (e.g., /cabins).
- Open the browser's console (DevTools) and verify that the data is being fetched and displayed.

---

6. Security Considerations

Ensure that **Row Level Security (RLS)** is enabled in your Supabase project. This limits what users can do with the exposed keys. You can define policies to restrict access further under **Table Editor > Policies** in the Supabase dashboard.

he **steps to set up storage buckets in Supabase** and upload files:

---

1. Access the Storage Feature

- Log in to your **Supabase dashboard**.
- From the menu on the left, select **Storage**.

---

2. Create a Storage Bucket

- Click the **"New Bucket"** button.
- **Name your bucket**:
  - Example 1: avatars (for user avatars)
  - Example 2: cabin-images (for cabin images)
- **Set bucket visibility**:
  - Check the option to make the bucket **public**.
    - Public buckets allow files to be accessed via URL, but secure operations like uploads/deletions still require **role-level security (RLS)**.
- Click **"Create Bucket"**.

---

3. Upload Files

- Select the bucket you just created.
- Drag and drop files from your computer into the bucket.
  - For example:
    - Navigate to the folder on your computer where the images are stored.
    - Drag files (e.g., cabin1.jpg, cabin2.jpg) into the Supabase bucket.
- Wait for the upload to complete.

---

4. Manage and View Uploaded Files

- After the upload, click on a file to:
  - View its details (e.g., upload date, file size).

    - Preview the file if it's an image or video.
- To retrieve the URL of a file:
    - Right-click on the file or use the **"Get URL"** option.

---

5. Use File URLs in Your App

- Copy the file URL.
- Add it to your app or database:
    - Example: Update the **Cabins** table in the **Table Editor** with the file URL for each cabin.
    - Double-click a cell in the **Supabase Table Editor**, paste the URL, and save.

---

6. Display the Image on Your Webpage

- Add an <img> element in your frontend code using the file URL as the src:

jsx

CopyEdit

```
<img src="https://<bucket-url>/cabin1.jpg" alt="Cabin Image" />
```

---

7. Programmatic Uploads

In future lessons, you can learn to upload files programmatically using the Supabase SDK, enabling features like allowing users to upload their avatars directly from your application.