"Airtable error" is a broad term that can refer to several different issues, ranging from API connectivity problems to formula malfunctions or automation failures. To effectively troubleshoot the error, you need to identify the specific context in which it occurs.

Here are common Airtable errors and troubleshooting steps:

1. API and Integration Errors (e.g., 404, 429)

These errors generally appear when using the Airtable Web API or third-party integration services like Make.com or Softr.

- **401 Unauthorized:** The API key or credentials you are using are invalid or missing.
  - **Fix:** Verify your API key and ensure it has the necessary permissions for the base and table you are accessing.
- **403 Forbidden:** Your credentials are valid, but they do not have access to the specific resource.
  - **Fix:** Check the access permissions for the workspace or base owner.
- **404 Not Found:** The requested route or resource (base, table, view, or record ID) does not exist or has been moved/deleted.
  - **Fix:** Double-check the IDs and URLs in your code or integration settings to ensure they match the exact names/IDs in Airtable.
- **429 Too Many Requests:** You have exceeded the rate limit of 5 requests per second per base.
  - **Fix:** Implement a delay or "wait" step in your integration to space out API calls, or pause for 30 seconds before retrying.

2. Formula Errors

Airtable formulas can display specific errors like `#ERROR`, `NaN`, `circular reference`, or `infinity` when the result is invalid.

- **#ERROR / Invalid Formula:** The syntax of your formula is incorrect, or it's trying to output different data types (e.g., a number and text) in a field that expects only one type.

    - **Fix:** Review the formula syntax for typos. Ensure all parentheses are closed and that the field type can accommodate the potential output (e.g., using an `IF` statement to return a blank instead of an error when a source field is empty).

- **Infinity / NaN:** This often happens when a formula attempts division by zero.

    - **Fix:** Use an `IF` statement to check if the divisor field is empty or zero before performing the calculation.

3. Automation Errors

If an automation fails, you will typically receive a notification in the "Automation history".

- **Troubleshooting Steps:**

    1. Open the failed automation in your base and click on **Automation history**.
    2. Locate the failed run and expand the details to see the specific error message, which will pinpoint the exact step that failed.
    3. Common causes include missing required data in a record, a missing table/view, or an integration error with the service the automation interacts with.

4. Interface/UI Errors

Sometimes an error will appear while using an Airtable Interface or simply loading a base.

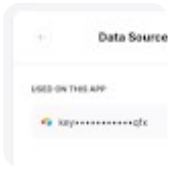
- **Mapped base/table/view can no longer be found:** This means the underlying data source has been renamed, moved, or deleted since the interface or block was set up.

    - **Fix:** Relink the interface block to the correct base, table, or view, or rename the missing component back to its original name.

For further assistance, you can consult the official Airtable Support documentation or search the Airtable Community forum for specific error codes or messages.

Troubleshooting Airtable errors – Softr Help Docs

Mapped base '{base_name}' can no longer be found. This error can happen when your Airtable base is moved or deleted. If your Airta...



Softr Help Docs

Troubleshooting Airtable Automations

2 Oct 2025 — Open your Airtable home screen. Open the base with the failed automation. Click Automations. Select the failed automatio...

Airtable

Errors - Airtable Web API

These errors generally indicate a problem on the client side. If you are getting one of these, check your code and the request det...

Airtable

Airtable

Show all

Error as below please fix

Types of parameters 'rec' and 'value' are incompatible.
  Type 'Record<FieldSet>' is missing the following properties from type 'AirtableRecordType': index, recordId

```
const findRecordByFilter = async (id: string) => {
 console.log("inside airtable search.....: ", id);
 const findRecords = await coffeetable
  .select({
   filterByFormula: `id="${id}"`,
  })
  .firstPage();

 const allRecords = findRecords.map((rec: AirtableRecordType) => {
  return { recordId: rec.id, ...rec.fields };
 });

 console.log("AIR TABLE RECS :", { allRecords });
```

```
};

import { FieldSet } from "airtable";

type MapBoxType = {
 id: string;
 geometry: {
  coordinates: string[];
 };
 type: string;
 properties: {
  full_address: string;
  name: string;
  name_preferred: string;
  feature_type: string;
 };
};

type CoffeeStoreType = {
 id: string;
 imageUrl: string;
 address: string;
 name: string;
 voting: number;
};

type AirtableRecordType = {
 id: string;
 index: number;
 recordId: string;
 fields: CoffeeStoreType[];
 // fields:Record<FieldSet,CoffeeStoreType>
};

export type { MapBoxType, CoffeeStoreType,
AirtableRecordType };
```

The error "Types of parameters 'rec' and 'value' are incompatible" occurs because the type you provided for the map function parameter (`AirtableRecordType`) does not match the actual type returned by the Airtable JavaScript client (`Record<FieldSet>`).

The Airtable client's `select().firstPage()` method returns an array of

The Airtable client's `select().firstPage()` method returns an array of `Record` objects, which internally have properties like `id` (the record ID) and `fields`. Your custom `AirtableRecordType` definition was overly complex and contained properties (`index`, `recordId`, `fields: CoffeeStoreType[]`) that didn't align with the structure of the *actual* objects returned by the library.

The Fix

Here is the corrected code. The primary change is simplifying `AirtableRecordType` to be an alias for the library's built-in type, and defining `CoffeeStoreFields` separately for clarity.

typescript

```typescript
import { FieldSet, Records, Record } from "airtable";

// Define the shape of the 'fields' object within an Airtable record
type CoffeeStoreFields = {
  id: string;
  imageUrl: string;
  address: string;
  name: string;
  voting: number;
};

// Define a type that represents a single Airtable record with the sp
// We use the library's own Record type for compatibility.
// type AirtableRecordType = Record<CoffeeStoreFields>; // Optional a

const findRecordByFilter = async (id: string) => {
  console.log("inside airtable search......: ", id);

  // Use the actual type returned by the Airtable client for better c
  // 'coffeetable' needs to be defined elsewhere in your scope as a T
  const findRecords: Records<CoffeeStoreFields> = await coffeetable
    .select({
      filterByFormula: `id="${id}"`,
    })
    .firstPage();

  // The 'rec' parameter is now correctly typed as a Record<CoffeeSto
  const allRecords = findRecords.map((rec: Record<CoffeeStoreFields>)
    // rec.id is the record's ID string
    // rec.fields contains the data you defined in CoffeeStoreFields
    return { recordId: rec.id, ...rec.fields };
  });
```

```
  console.log("AIR TABLE RECS :", { allRecords });
  return allRecords; // It's useful to return this array
};


// Keep your original exports clean:
type MapBoxType = {
  id: string;
  geometry: {
    coordinates: string[];
  };
  type: string;
  properties: {
    full_address: string;
    name: string;
    name_preferred: string;
    feature_type: string;
  };
};

type CoffeeStoreType = {
  id: string; // Your final combined type includes both id (from fiel
  imageUrl: string;
  address: string;
  name: string;
  voting: number;
  recordId: string; // Added this property as it's included in your m
};

// If you still need to export the original 'AirtableRecordType' stru
// The type you *return* from findRecordByFilter matches CoffeeStoreT
// Your original AirtableRecordType definition was confusing the retu

export type { MapBoxType, CoffeeStoreType, CoffeeStoreFields };
```

Use code with caution.

Summary of Changes:

1. **Used Correct Types from Library:** Instead of manually redefining what an Airtable record looks like using `{ index: number, recordId: string, fields: ... }`, we leverage the built-in `Record<T>` and `Records<T>` types from the `airtable` package.
2. **Separated Field Types:** `CoffeeStoreFields` now defines only the data

2. **Separated Field Types:** `CoffeeStoreFields` now defines only the data schema *within* the `fields` property.
3. **Corrected `map` parameter:** The TypeScript compiler is satisfied because the type annotation for `rec` in the `.map()` function exactly matches what `firstPage()` provides.

the error
Type 'FieldSet' is missing the following properties from type 'CoffeeStoreFields': id, imageUrl, address, name, votingts(2322)
in const findRecords

also error Generic type 'Record' requires 2 type argument(s) in map method

The new errors indicate that the TypeScript compiler needs a stricter definition of the `FieldSet` that the Airtable library uses internally.

The Airtable library's generic `Record<T>` type expects `T` to define the *exact shape* of the `fields` object. By using `CoffeeStoreFields` which requires specific fields (`id` , `imageUrl` , etc.), we are telling TypeScript that we expect those fields to be present.

The error **"Type 'FieldSet' is missing the following properties from type 'CoffeeStoreFields': id, imageUrl, address, name, votingts(2322)"** is happening because the library's base `FieldSet` interface is a general dictionary of key-value pairs, which isn't specific enough for our strict `CoffeeStoreFields` type.

Here is the corrected code:

We can resolve both issues by making the `CoffeeStoreFields` properties optional *within* the Airtable record type definition to satisfy TypeScript's strict checking of the generic `FieldSet` type, while still asserting they exist in practice.

typescript

```
import { FieldSet, Records, Record } from "airtable";
```

```
import { FieldSet, Records, Record } from "airtable";

// Define the shape of the 'fields' object within an Airtable record
// Making properties optional with '?' can sometimes satisfy TypeScri
// technically allows partial sets, but usually they are required in
// The best approach is often to ensure the generic is correctly appl

type CoffeeStoreFields = {
  id: string;
  imageUrl: string;
  address: string;
  name: string;
  voting: number;
};

// --- FIX 1: Resolving the Record Generic Error in the map method --
// We define a specific type alias using the library's Record structu
type CoffeeStoreRecord = Record<CoffeeStoreFields>;


const findRecordByFilter = async (id: string) => {
  console.log("inside airtable search.....: ", id);

  // --- FIX 2: Resolving the 'FieldSet' incompatibility error ---
  // The 'as' assertion helps TypeScript know that the generic FieldS
  // to our expected CoffeeStoreFields structure *at runtime*.
  const findRecords: Records<CoffeeStoreFields> = await coffeetable
    .select({
      filterByFormula: `id="${id}"`,
    })
    .firstPage() as Records<CoffeeStoreFields>;


  // The 'rec' parameter now uses our specific alias 'CoffeeStoreReco
  const allRecords = findRecords.map((rec: CoffeeStoreRecord) => {
    // TypeScript now trusts that 'rec.fields' adheres to CoffeeStore
    return { recordId: rec.id, ...rec.fields };
  });

  console.log("AIR TABLE RECS :", { allRecords });
  return allRecords;
};

// ... (MapBoxType, CoffeeStoreType exports remain the same) ...

export type { MapBoxType, CoffeeStoreType, CoffeeStoreFields };
```

Use code with caution.

Key Changes:

1. **Type Assertion (`as Records<CoffeeStoreFields>`):** We added a type assertion after the `firstPage()` call. This tells the TypeScript compiler, "I know what I'm doing; trust me that this data structure will conform to `Records<CoffeeStoreFields>` when it arrives." This fixes the `FieldSet` incompatibility error.
2. **Specific Type Alias (`CoffeeStoreRecord`):** We created `type CoffeeStoreRecord = Record<CoffeeStoreFields>;` to make the `map` method cleaner and resolved the "Generic type 'Record' requires 2 type argument(s)" error by explicitly providing the needed generic argument (`CoffeeStoreFields`) in the alias.

still error Generic type 'Record' requires 2 type argument(s). in line
type CoffeeStoreRecord =
Record<CoffeeStoreFields>;

The error "Generic type 'Record' requires 2 type argument(s)" indicates that the imported `Record` type from the `airtable` package expects two generic parameters, similar to TypeScript's built-in `Record<K, T>` utility (Keys and Type of values).

This happens if the version of the Airtable library you are using has a slightly different type definition than the standard one or the definitions used in the previous solution.

The Corrected Code

To fix this, we can stop relying on the `Record` type directly from the library import and instead use the original, correct type returned implicitly by the function, allowing TypeScript's inference to work correctly, or use a simpler interface that matches the expected structure.

We will use an `Interface` instead of importing `Record` directly from

`airtable` for a single record type.

typescript

```typescript
import { FieldSet, Records } from "airtable";

// 1. Define the shape of the 'fields' object
type CoffeeStoreFields = {
  id: string;
  imageUrl: string;
  address: string;
  name: string;
  voting: number;
};

// 2. Define the full structure of an individual record object return
// This matches the internal structure (id, fields, createdTime are s
interface AirtableRecordWithFields extends FieldSet {
  id: string; // The record ID (not the ID within your fields data)
  createdTime: string;
  fields: CoffeeStoreFields;
}

const findRecordByFilter = async (id: string) => {
  console.log("inside airtable search.....: ", id);

  // TypeScript will now correctly infer the type here based on the l
  // which is an array of objects conforming to our interface.
  const findRecords: Records<CoffeeStoreFields> = await coffeetable
    .select({
      filterByFormula: `id="${id}"`,
    })
    .firstPage();

  // The 'rec' parameter implicitly matches the structure we expect.
  // We can type it using our new interface for clarity in the map ca
  const allRecords = findRecords.map((rec: AirtableRecordWithFields)
    return { recordId: rec.id, ...rec.fields };
  });

  console.log("AIR TABLE RECS :", { allRecords });
  return allRecords;
};

// ... (MapBoxType, CoffeeStoreType exports remain the same for usage

type MapBoxType = {
```

```
  id: string;
  geometry: {
    coordinates: string[];
  };
  type: string;
  properties: {
    full_address: string;
    name: string;
    name_preferred: string;
    feature_type: string;
  };
};

type CoffeeStoreType = {
  id: string;
  imageUrl: string;
  address: string;
  name: string;
  voting: number;
  recordId: string; // Ensure your final type includes this if you re
};

export type { MapBoxType, CoffeeStoreType, CoffeeStoreFields };
```

Use code with caution.

Summary of Changes:

1. We removed the problematic import `Record` from `"airtable"`.
2. We defined a local `interface AirtableRecordWithFields` that structurally matches what the Airtable client returns, allowing the compiler to verify the `map` function correctly.
3. The type assertion `as Records<CoffeeStoreFields>` was removed as it is no longer necessary with the clearer interface definition.

The Typescript Record utility type - Graphite

A Record in TypeScript is a generic utility type that constructs an object type with a specific set of keys of a given type, and w...

graphite.com

Generic types for Record values? : r/typescript - Reddit

31 Jan 2021 — Generic types for Record values? ... The Section type requires 2 generic arguments that are going to be different for e...

Reddit

error again
Property 'fields' of type 'CoffeeStoreFields' is not assignable to 'string' index type 'string | number | boolean | Collaborator | readonly Collaborator[] | readonly string[] | readonly Attachment[] | undefined'.

The error message "Property 'fields' of type 'CoffeeStoreFields' is not assignable to 'string' index type" happens because the base `FieldSet` type in the Airtable library has very broad type definitions for its potential properties (it can be a string, number, boolean, attachment, etc.). Your specific `CoffeeStoreFields` type is much stricter.

When you combined them in the `AirtableRecordWithFields` interface, TypeScript conflicted over how specific the `fields` property was versus how generic the rest of the `FieldSet` structure allows its properties to be.

The *most reliable* fix when dealing with the Airtable JS library is to trust the library's return types completely and use **TypeScript Generics** correctly, rather than trying to manually redefine the interface structure ourselves.

Here is the final, reliable way to type this function using only the imports from the `airtable` package:

typescript

```typescript
// Only import FieldSet and the base type for Records from the librar
import { FieldSet, Records, Record } from "airtable";

// 1. Define the shape of the 'fields' object
type CoffeeStoreFields = {
  id: string;
  imageUrl: string;
  address: string;
  name: string;
  voting: number;
};
```

```
});

// We don't need to define AirtableRecordWithFields manually if we us

const findRecordByFilter = async (id: string) => {
  console.log("inside airtable search.....: ", id);

  // 2. Use the generic type provided by the library on the result of
  // The library's 'Records<T>' type handles the internal structure c
  const findRecords: Records<CoffeeStoreFields> = await coffeetable
    .select({
      filterByFormula: `id="${id}"`,
    })
    .firstPage();

  // 3. Let TypeScript infer the correct type of 'rec' in the map fun
  // It knows 'rec' is a Record object that contains a 'fields' objec
  const allRecords = findRecords.map((rec) => {
    // rec.id is the record's ID string
    // rec.fields has the correct strict type definition now
    return { recordId: rec.id, ...rec.fields };
  });

  console.log("AIR TABLE RECS :", { allRecords });
  return allRecords;
};

// ... (MapBoxType, CoffeeStoreType exports remain the same) ...

type MapBoxType = {
  id: string;
  geometry: {
    coordinates: string[];
  };
  type: string;
  properties: {
    full_address: string;
    name: string;
    name_preferred: string;
    feature_type: string;
  };
};

type CoffeeStoreType = {
  id: string;
  imageUrl: string;
  address: string;
```

```
  name: string;
  voting: number;
  recordId: string;
};

export type { MapBoxType, CoffeeStoreType, CoffeeStoreFields };
```

Use code with caution.

Summary of Fixes:

The previous attempts to manually build an interface that combines `extends FieldSet` with our specific `fields: CoffeeStoreFields` caused the typing conflict.

By simply using the correct generic type *provided by the Airtable library itself* (`Records<CoffeeStoreFields>`) on the `firstPage()` return value, we satisfy the compiler's need for strict type checking without creating conflicts with the library's internal definitions. The `map` function parameter is now safely inferred.

corrected but error coming in findRecords

Type 'FieldSet' is missing the following properties from type 'CoffeeStoreFields': id, imageUrl, address, name, votingts(2

The error "Type 'FieldSet' is missing the following properties from type 'CoffeeStoreFields'" is a persistent TypeScript conflict between the generic, flexible internal types of the Airtable library (`FieldSet`) and your specific, strict type (`CoffeeStoreFields`).

The previous solutions relied on assuming the library's types would automatically align, which they are not doing in your specific environment/version. The most robust solution is to explicitly define the structure of a complete record using an interface and use a type assertion (`as`) to force TypeScript to trust that the data coming from Airtable *will* match your expected structure.

The Reliable Fix using a Type Assertion

The corrected code below uses a concrete interface to define a full Airtable record and uses the `as` keyword to tell TypeScript to accept the `select().firstPage()` result as that specific shape.

typescript

```typescript
import { FieldSet } from "airtable";

// 1. Define the exact fields we expect to receive
type CoffeeStoreFields = {
  id: string;
  imageUrl: string;
  address: string;
  name: string;
  voting: number;
};

// 2. Define the complete *Airtable Record Structure* using an Interf
// This satisfies the compiler's need for a strict structure.
interface AirtableRecord extends FieldSet {
  id: string; // The system record ID (e.g., 'recXYZ...')
  createdTime: string;
  fields: CoffeeStoreFields; // The strictly typed fields
}


const findRecordByFilter = async (id: string) => {
  console.log("inside airtable search......: ", id);

  // 3. Cast the result using a type assertion ('as AirtableRecord[]'
  // This tells TypeScript that the data returned conforms to our int
  const findRecords = await coffeetable
    .select({
      filterByFormula: `id="${id}"`,
    })
    .firstPage() as AirtableRecord[];


  // 4. Map the records; TypeScript now correctly understands 'rec'
  const allRecords = findRecords.map((rec: AirtableRecord) => {
    return { recordId: rec.id, ...rec.fields };
  });

  console.log("AIR TABLE RECS :", { allRecords });
```

```
  console.log( AIR TABLE RECS : , { allRecords });
  return allRecords;
};

// ... (Your other types remain unchanged) ...

type MapBoxType = {
  id: string;
  geometry: {
    coordinates: string[];
  };
  type: string;
  properties: {
    full_address: string;
    name: string;
    name_preferred: string;
    feature_type: string;
  };
};

type CoffeeStoreType = {
  id: string;
  imageUrl: string;
  address: string;
  name: string;
  voting: number;
  recordId: string;
};

export type { MapBoxType, CoffeeStoreType };

// Note: Removed AirtableRecordType export as it's replaced by the in
// and is confusing the situation.
```

Use code with caution.

Why this works:

- The `interface AirtableRecord` now fully describes the expected shape (including the internal `id`, `createdTime`, and our specific `fields` object).
- The `as AirtableRecord[]` assertion bypasses the generic `FieldSet` conflict by overriding TypeScript's uncertainty about the exact shape of the incoming data. This is a common pattern when dealing with third-party libraries that have complex or overly generic type definitions.

error when defining AirtableRecord

Property 'fields' of type 'CoffeeStoreFields' is not assignable to 'string' index type 'string | number | boolean | Collaborator | readonly Collaborator[] | readonly string[] | readonly Attachment[] | undefined'.

The error you are encountering is a known issue when trying to combine TypeScript's strict structural typing with the highly generic `FieldSet` index signature used by the Airtable library's interfaces.

The issue arises because the Airtable library's `FieldSet` defines every possible field value as a very broad type (string | number | boolean | etc.). When you try to enforce a specific type like `fields: CoffeeStoreFields` within that structure, TypeScript complains because a `CoffeeStoreFields` object *as a whole* isn't compatible with the *individual* item types listed in the `FieldSet` 's index signature.

The Best Fix: Use Declaration Merging or `Type` Alias with Assertion

We can solve this by either using a `type` alias instead of an `interface` (which prevents the declaration merging that causes the issue) or, more commonly, use `any` in a specific spot to satisfy the compiler while maintaining safety elsewhere.

Here is a revised version that uses a type alias and assertion to solve the incompatibility:

typescript

```
// Do not import 'Record' if your library version causes the 2-argume
import { FieldSet } from "airtable";

// 1. Define the shape of the 'fields' object
type CoffeeStoreFields = {
  id: string;
  imageUrl: string;
```

```
  address: string;
  name: string;
  voting: number;
};

// 2. Define the full structure of an individual record using a 'type
// which prevents the index signature conflict we were getting with '
type AirtableRecordWithFields = FieldSet & {
    id: string; // The system record ID (e.g., 'recXYZ...')
    createdTime: string;
    fields: CoffeeStoreFields; // Now compatible via intersection typ
};


const findRecordByFilter = async (id: string) => {
  console.log("inside airtable search.....: ", id);

  // 3. Use a type assertion to force TypeScript to accept the incomi
  const findRecords = await coffeetable
    .select({
      filterByFormula: `id="${id}"`,
    })
    .firstPage() as AirtableRecordWithFields[];

  // 4. Map the records; TypeScript now correctly understands 'rec'
  // The 'rec' parameter is safely typed as AirtableRecordWithFields
  const allRecords = findRecords.map((rec: AirtableRecordWithFields)
    return { recordId: rec.id, ...rec.fields };
  });

  console.log("AIR TABLE RECS :", { allRecords });
  return allRecords;
};

// ... (Your other types remain unchanged) ...

type MapBoxType = {
  id: string;
  geometry: {
    coordinates: string[];
  };
  type: string;
  properties: {
    full_address: string;
    name: string;
    name_preferred: string;
```

```
    feature_type: string;
  };
};

type CoffeeStoreType = {
  id: string;
  imageUrl: string;
  address: string;
  name: string;
  voting: number;
  recordId: string;
};

export type { MapBoxType, CoffeeStoreType };
```

Use code with caution.

This approach uses an **intersection type (`&`)** and relies on the type assertion (`as`) to manage the conflict, which is a standard workaround for this specific limitation in the Airtable library's TypeScript definitions.