I’m building a mini DNS API for a backend assessment. Need to setup the project foundation. requirements are 1. nextjs project + typescript 2. DB is progres 3. want to use knex for orm and yup for api validation 4. may use redis 5. will be adding migration script preferrable using knex.

**ChatGPT said:**

Great! Here's a clear and modular plan to set up your mini DNS API project with all your specified requirements:

**✅ 1. Create the Next.js + TypeScript Project**

bash

CopyEdit

npx create-next-app@latest dns-api --typescript

cd dns-api

**✅ 2. Install Dependencies**

**API Validation & Type Safety**

bash

CopyEdit

npm install yup

**Knex + PostgreSQL**

bash

CopyEdit

npm install knex pg

**Redis (optional, if you use caching)**

bash

CopyEdit

npm install ioredis

**Knex CLI for migrations/seeding**

bash

CopyEdit

npm install --save-dev knex

**✅ 3. Configure knex**

**Create a knex config file:**

bash

CopyEdit

npx knex init

This will create a knexfile.js. Rename it to knexfile.ts for TypeScript usage.

Update it like so:

ts

CopyEdit

// knexfile.ts

import type { Knex } from 'knex';

import dotenv from 'dotenv';

dotenv.config();

const config: { [key: string]: Knex.Config } = {

development: {

client: 'pg',

connection: process.env.DATABASE\_URL,

migrations: {

directory: './db/migrations',

},

seeds: {

directory: './db/seeds',

},

},

};

export default config;

You will also need to enable TypeScript support for the CLI (see next step).

**✅ 4. Add Knex Migration Script & Setup**

Create folder structure:

bash

CopyEdit

mkdir -p db/migrations db/seeds

Create a helper script for running knex with TypeScript:

**package.json**

json

CopyEdit

"scripts": {

"dev": "next dev",

"build": "next build",

"start": "next start",

"migrate": "ts-node --esm ./node\_modules/knex/bin/cli.js migrate:latest --knexfile knexfile.ts",

"migrate:rollback": "ts-node --esm ./node\_modules/knex/bin/cli.js migrate:rollback --knexfile knexfile.ts"

}

Install ts-node and @types/node if needed:

bash

CopyEdit

npm install --save-dev ts-node @types/node

**✅ 5. Setup Postgres Environment**

Add .env.local:

env

CopyEdit

DATABASE\_URL=postgresql://username:password@localhost:5432/dnsdb

REDIS\_URL=redis://localhost:6379

Replace with your actual credentials.

**✅ 6. Setup DB Client**

You can add a basic knex instance:

ts

CopyEdit

// lib/db.ts

import knex from 'knex';

import config from '../knexfile';

const db = knex(config.development);

export default db;

**✅ 7. Setup Redis (Optional)**

ts

CopyEdit

// lib/redis.ts

import Redis from 'ioredis';

const redis = new Redis(process.env.REDIS\_URL || '');

export default redis;

**✅ 8. Example Migration**

bash

CopyEdit

npm run migrate -- make create\_dns\_records\_table

Then edit the migration file:

ts

CopyEdit

// db/migrations/XXXX\_create\_dns\_records\_table.ts

import { Knex } from 'knex';

export async function up(knex: Knex): Promise<void> {

return knex.schema.createTable('dns\_records', (table) => {

table.increments('id').primary();

table.string('name').notNullable();

table.string('type').notNullable(); // e.g., A, CNAME, MX

table.string('value').notNullable();

table.integer('ttl').defaultTo(3600);

table.timestamps(true, true);

});

}

export async function down(knex: Knex): Promise<void> {

return knex.schema.dropTable('dns\_records');

}

**✅ 9. Setup API with Yup Validation Example**

ts

CopyEdit

// pages/api/dns.ts

import { NextApiRequest, NextApiResponse } from 'next';

import \* as yup from 'yup';

import db from '@/lib/db';

const schema = yup.object({

name: yup.string().required(),

type: yup.string().oneOf(['A', 'CNAME', 'MX']).required(),

value: yup.string().required(),

ttl: yup.number().default(3600),

});

export default async function handler(req: NextApiRequest, res: NextApiResponse) {

if (req.method !== 'POST') return res.status(405).end();

try {

const data = await schema.validate(req.body);

const [record] = await db('dns\_records').insert(data).returning('\*');

res.status(201).json(record);

} catch (err: any) {

res.status(400).json({ error: err.message });

}

}

**✅ 10. Type Safety with Knex**

Create types for your tables:

ts

CopyEdit

// types/db.ts

export interface DNSRecord {

id: number;

name: string;

type: 'A' | 'CNAME' | 'MX';

value: string;

ttl: number;

created\_at: string;

updated\_at: string;

}

Would you like me to generate:

* A docker-compose.yml for Postgres + Redis?
* A sample test (e.g., with Jest)?
* Or maybe CRUD endpoints for DNS records?

Let me know!

4o

**You said:**

I need to support A and CNAME records with real-world constraints. How should I structure my database schema for this?

**ChatGPT said:**

To support **realistic constraints** for **A** and **CNAME** DNS records in your schema, it's important to reflect DNS semantics in your **PostgreSQL schema** and **application logic**.

**✅ DNS Record Types Overview**

**A Record:**

* Maps a **domain name** to an **IPv4 address**.
* Must contain a valid IPv4 address.

**CNAME Record:**

* Maps a **domain name** to another **canonical domain name**.
* A domain with a CNAME **cannot have any other record types** (including A).

**✅ Suggested Table Schema**

sql

CopyEdit

CREATE TABLE dns\_records (

id SERIAL PRIMARY KEY,

name TEXT NOT NULL, -- e.g., 'www.example.com'

type TEXT NOT NULL CHECK (type IN ('A', 'CNAME')),

value TEXT NOT NULL, -- IP (for A) or domain (for CNAME)

ttl INTEGER NOT NULL DEFAULT 3600,

created\_at TIMESTAMPTZ DEFAULT NOW(),

updated\_at TIMESTAMPTZ DEFAULT NOW(),

-- Enforce uniqueness of (name), so one domain can’t have both A and CNAME

UNIQUE (name)

);

This schema:

* Prevents having multiple records (like both A and CNAME) for the same domain.
* Uses constraints to only allow A or CNAME in type.
* Stores value as text (validate via application layer).

**✅ Application-Layer Validation (Required)**

In your API, enforce:

**For A Records:**

* value must be a **valid IPv4 address** (e.g., 192.168.1.1)

ts

CopyEdit

import \* as yup from 'yup';

const ARecordSchema = yup.object({

type: yup.string().oneOf(['A']).required(),

name: yup.string().required(),

value: yup

.string()

.matches(

/^(?:\d{1,3}\.){3}\d{1,3}$/,

'Must be a valid IPv4 address'

)

.required(),

ttl: yup.number().default(3600),

});

**For CNAME Records:**

* value must be a **valid domain name** (e.g., example.com)

ts

CopyEdit

const CNAMERecordSchema = yup.object({

type: yup.string().oneOf(['CNAME']).required(),

name: yup.string().required(),

value: yup

.string()

.matches(

/^(?!:\/\/)([a-zA-Z0-9-\_]+\.)+[a-zA-Z]{2,}$/,

'Must be a valid domain name'

)

.required(),

ttl: yup.number().default(3600),

});

**✅ Enforcing No Mixed Record Types**

You can:

1. Enforce uniqueness on name in DB (already done).
2. In application logic:
   * Query for existing records with same name.
   * If any exist and the type is different → **reject the insert**.

ts

CopyEdit

// lib/validateDnsRecord.ts

import db from '@/lib/db';

export async function canInsertRecord(name: string, type: 'A' | 'CNAME') {

const existing = await db('dns\_records').where({ name }).first();

if (!existing) return true;

return existing.type === type;

}

**✅ Optional: Split into Two Tables?**

If you'd like more normalization, you could use:

sql

CopyEdit

CREATE TABLE dns\_records (

id SERIAL PRIMARY KEY,

name TEXT NOT NULL UNIQUE,

type TEXT NOT NULL CHECK (type IN ('A', 'CNAME')),

ttl INTEGER NOT NULL DEFAULT 3600,

created\_at TIMESTAMPTZ DEFAULT NOW(),

updated\_at TIMESTAMPTZ DEFAULT NOW()

);

CREATE TABLE a\_records (

dns\_record\_id INTEGER PRIMARY KEY REFERENCES dns\_records(id) ON DELETE CASCADE,

ip\_address INET NOT NULL

);

CREATE TABLE cname\_records (

dns\_record\_id INTEGER PRIMARY KEY REFERENCES dns\_records(id) ON DELETE CASCADE,

canonical\_name TEXT NOT NULL

);

But for your case (limited types and simplicity), keeping it in a single dns\_records table is both sufficient and practical.

Would you like me to:

* Generate the migration file for this schema?
* Add sample insert APIs with this validation?
* Help implement logic to check conflicts on insert?

Let me know!

4o

**You said:**

Can I use PostgreSQL enum types for record type? Would that be a good idea for future extensibility?

**ChatGPT said:**

Yes, using **PostgreSQL enum types** for the type field (e.g., 'A', 'CNAME', etc.) is a **good idea**, especially if:

* You want **strong type safety** at the database level.
* You want to **limit allowed values** and keep them centralized.
* You expect to **extend with more record types** in the future (like MX, AAAA, TXT, etc.).

**✅ Pros of Using PostgreSQL ENUM**

* **Validation enforced at DB level**: No invalid types can be inserted.
* **Readable and self-documenting**: The allowed values are easy to inspect.
* **Future-proof**: Easily extendable as new record types are needed.
* Prevents typos in record types (e.g., 'a', 'Cname').

**⚠️ Considerations**

* **Altering enums in PostgreSQL is not as flexible as a lookup table** (prior to PG 10).
  + But in modern PostgreSQL (≥ v10), adding new values is straightforward:

sql

CopyEdit

ALTER TYPE dns\_record\_type ADD VALUE 'MX';

* You can’t easily remove enum values once in use, so choose naming carefully.

**✅ How to Implement ENUM in PostgreSQL (via Knex)**

**1. Define the enum in a migration**

ts

CopyEdit

export async function up(knex: Knex): Promise<void> {

await knex.raw(`CREATE TYPE dns\_record\_type AS ENUM ('A', 'CNAME')`);

await knex.schema.createTable('dns\_records', (table) => {

table.increments('id').primary();

table.string('name').notNullable().unique();

table

.specificType('type', 'dns\_record\_type')

.notNullable();

table.string('value').notNullable();

table.integer('ttl').notNullable().defaultTo(3600);

table.timestamps(true, true);

});

}

export async function down(knex: Knex): Promise<void> {

await knex.schema.dropTable('dns\_records');

await knex.raw(`DROP TYPE dns\_record\_type`);

}

**2. (Optional) Type Mapping in TypeScript**

ts

CopyEdit

export type DnsRecordType = 'A' | 'CNAME';

export interface DNSRecord {

id: number;

name: string;

type: DnsRecordType;

value: string;

ttl: number;

created\_at: string;

updated\_at: string;

}

**✅ Future Extensibility Example**

To add a new type like MX:

sql

CopyEdit

ALTER TYPE dns\_record\_type ADD VALUE 'MX';

Then, just update your validation logic in the app.

**✅ Summary**

| **Feature** | **Enum Type** | **Text + Validation** |
| --- | --- | --- |
| Type safety | ✅ | ❌ (app only) |
| DB-level enforcement | ✅ | ❌ |
| Easy to inspect | ✅ | ❌ |
| Easily changeable | ❌ | ✅ |

**Verdict**: If you’re working with a limited and well-defined set of DNS record types (as you are), **PostgreSQL enums are a great fit**.

Let me know if you want a complete migration script and updated API validation using this enum!

4o

**You said:**

Should I allow a record with multiple A records for the same hostname? And how do I handle CNAME conflicts correctly?

**ChatGPT said:**

Great questions. Here's a detailed breakdown to help you handle **A record duplication** and **CNAME conflicts** according to **real-world DNS rules**:

**✅ Should You Allow Multiple A Records for the Same Hostname?**

**Yes, absolutely.**  
It is **standard practice** to allow **multiple A records** for the same name (hostname).

**🔍 Why?**

* This is how **load balancing** and **redundancy** are achieved in DNS.
* When a DNS resolver queries the hostname, it gets multiple IP addresses.

**✅ Implementation**

* Allow multiple A records for the same name, each with a different value (IP).
* So: name + value must be unique per A record — but multiple values per name are allowed.

sql

CopyEdit

-- Example rows:

| name | type | value |

|----------------|------|---------------|

| www.example.com| A | 192.168.1.1 |

| www.example.com| A | 192.168.1.2 |

**✅ Enforce via Constraint**

Optional: prevent exact duplicates using a composite unique index:

sql

CopyEdit

UNIQUE (name, type, value)

**❌ Should You Allow CNAME with A Records for Same Hostname?**

**No.**  
**CNAME records conflict with any other record types for the same hostname.**

**🔍 Why?**

* The DNS spec (RFC 1034/1035) says a hostname with a CNAME record **must not** have any other records (A, MX, etc.).
* So: if name = www.example.com has a CNAME, it **cannot** also have A or any other type.

**✅ Implementation Plan**

**1. Schema: Allow multiple rows per name**

But no two records with the same name should have different types **if one is a CNAME**.

So, don’t use a simple UNIQUE(name) constraint — manage this in logic.

**2. Validation Logic (App-level)**

Before inserting a record:

ts

CopyEdit

async function validateDnsInsert(name: string, type: 'A' | 'CNAME') {

const existingRecords = await db('dns\_records').where({ name });

if (existingRecords.length === 0) return true;

const existingTypes = new Set(existingRecords.map(r => r.type));

if (type === 'CNAME') {

// Can't add CNAME if any other records exist

throw new Error(`CNAME record cannot coexist with other records for ${name}`);

}

if (existingTypes.has('CNAME')) {

// Can't add A if CNAME already exists

throw new Error(`Cannot add A record to ${name} because a CNAME exists`);

}

return true;

}

**✅ Final Schema Strategy**

ts

CopyEdit

table.increments('id').primary();

table.string('name').notNullable(); // domain name

table.specificType('type', 'dns\_record\_type').notNullable(); // 'A' or 'CNAME'

table.string('value').notNullable(); // IP or canonical name

table.integer('ttl').notNullable().defaultTo(3600);

table.timestamps(true, true);

table.unique(['name', 'type', 'value']); // Prevent exact duplicates

**✅ Summary**

| **Rule** | **Enforced How** |
| --- | --- |
| Multiple A records for same name → ✅ allowed | No constraint |
| Same name with CNAME and A/MX/etc → ❌ not allowed | Application logic |
| Prevent exact duplicates | DB unique(name, type, value) |

Would you like a full knex migration + TypeScript validation example for this logic?

4o

**You said:**

For hostname validation in TypeScript, can I use a regex or is there a better library for validating domain format?

**ChatGPT said:**

You **can** use a regex for validating hostnames/domains in TypeScript — but for **accuracy, maintainability, and edge cases**, it's **better to use a specialized library**.

**✅ Option 1: Use a Library (Recommended)**

**Use** [**is-valid-domain**](https://www.npmjs.com/package/is-valid-domain)

bash

CopyEdit

npm install is-valid-domain

**Example:**

ts

CopyEdit

import isValidDomain from 'is-valid-domain';

isValidDomain('example.com'); // true

isValidDomain('sub.example.co.uk'); // true

isValidDomain('localhost'); // false (unless configured)

isValidDomain('123'); // false

**Configuration Example:**

ts

CopyEdit

isValidDomain('localhost', { subdomain: false }); // false

isValidDomain('example.com', { allowUnicode: true }); // true

**✅ Benefits:**

* Correctly handles rules for TLDs, labels, punycode, Unicode, etc.
* Easier to read and maintain than regex.
* More robust to odd edge cases.

**🚫 Option 2: Regex (Works but Limited)**

Here's a **reasonably solid** regex for hostname/domain validation:

ts

CopyEdit

const domainRegex = /^(?!:\/\/)(?=.{1,253}$)(?:(?!-)[a-zA-Z0-9-\_]{1,63}(?<!-)\.)+[a-zA-Z]{2,}$/;

function isValidHostname(hostname: string): boolean {

return domainRegex.test(hostname);

}

**⚠️ Caveats:**

* Doesn’t support punycode (e.g., xn-- domains).
* Won’t catch all invalid TLDs.
* Can be brittle for edge cases like leading/trailing hyphens, Unicode, etc.

**✅ Summary**

| **Option** | **Pros** | **Cons** |
| --- | --- | --- |
| is-valid-domain | ✅ Accurate, robust, easy to use | Slightly larger dependency |
| Regex | ✅ Lightweight, no install needed | ❌ Harder to maintain, brittle |

**Recommendation**: Use is-valid-domain unless you're constrained to zero-dependency environments.

Would you like me to wire up this validation into your yup schema for CNAME or domain inputs?

4o

**You said:**

how do i use this is-valid-domain in yup validation?

**ChatGPT said:**

To use is-valid-domain inside a **Yup schema**, you can integrate it using Yup’s .test() method, which allows you to run custom validation logic.

**✅ Step-by-Step: Use is-valid-domain with Yup**

**1. Install the library**

bash

CopyEdit

npm install is-valid-domain

**2. Import and use in Yup schema**

ts

CopyEdit

import \* as yup from 'yup';

import isValidDomain from 'is-valid-domain';

const dnsRecordSchema = yup.object({

name: yup

.string()

.required()

.test(

'is-valid-domain',

'Invalid domain name',

(value) => !!value && isValidDomain(value, { subdomain: true })

),

type: yup.string().oneOf(['A', 'CNAME']).required(),

value: yup

.string()

.required()

.when('type', {

is: 'A',

then: (schema) =>

schema.matches(

/^(?:\d{1,3}\.){3}\d{1,3}$/,

'Must be a valid IPv4 address'

),

otherwise: (schema) =>

schema.test(

'is-valid-cname-target',

'Must be a valid domain name',

(value) => !!value && isValidDomain(value, { subdomain: true })

),

}),

ttl: yup.number().default(3600).min(0),

});

**✅ Example Usage**

ts

CopyEdit

try {

const validated = await dnsRecordSchema.validate({

name: 'www.example.com',

type: 'CNAME',

value: 'target.example.com',

ttl: 300,

});

// Safe to use `validated` now

} catch (error) {

console.error(error.errors); // ["Invalid domain name", etc.]

}

**🔄 Summary**

| **Field** | **Validation Logic** |
| --- | --- |
| name | Required, valid domain (e.g., foo.com) |
| type | Must be 'A' or 'CNAME' |
| value | IPv4 if type = A, domain if CNAME |
| ttl | Optional, must be ≥ 0 |

This setup gives you robust, DNS-aware input validation in your API using yup + is-valid-domain.

Let me know if you want this extracted into a reusable validator or used in your API handler!

4o

**You said:**

But this wont support IPs. right? i feel regex would be better as i will have full control over it

**ChatGPT said:**

You're absolutely right — **is-valid-domain does not validate IP addresses**, only domain names. So for scenarios like supporting both:

* **A records** (which need valid **IPv4 addresses**)
* **CNAME records** (which need valid **hostnames**)

you'd **need separate logic anyway**.

**✅ Using Regex Gives You Full Control**

For your DNS API use case — where:

* You control which record types are allowed (A, CNAME)
* You want precision
* You don't need internationalized domain names (IDNs)

➡️ **Regex is a solid choice** and keeps things centralized and dependency-free.

**✅ Recommended Regex + Yup Schema**

**1. Regex for IPv4 Address**

ts

CopyEdit

const ipv4Regex = /^(25[0-5]|2[0-4]\d|1\d{2}|[1-9]?\d)(\.(25[0-5]|2[0-4]\d|1\d{2}|[1-9]?\d)){3}$/;

* Covers full IPv4 spec
* No overmatching like 999.999.999.999

**2. Regex for Hostname / Domain Name**

ts

CopyEdit

const domainRegex = /^(?!:\/\/)(?=.{1,253}$)(?:(?!-)[a-zA-Z0-9-]{1,63}(?<!-)\.)+[a-zA-Z]{2,}$/;

* Handles:
  + Subdomains
  + TLDs
  + Hyphen rules
  + Max label + domain length limits

**3. Full Yup Schema**

ts

CopyEdit

import \* as yup from 'yup';

const ipv4Regex = /^(25[0-5]|2[0-4]\d|1\d{2}|[1-9]?\d)(\.(25[0-5]|2[0-4]\d|1\d{2}|[1-9]?\d)){3}$/;

const domainRegex = /^(?!:\/\/)(?=.{1,253}$)(?:(?!-)[a-zA-Z0-9-]{1,63}(?<!-)\.)+[a-zA-Z]{2,}$/;

const dnsRecordSchema = yup.object({

name: yup

.string()

.required('Hostname is required')

.matches(domainRegex, 'Invalid domain name'),

type: yup

.string()

.oneOf(['A', 'CNAME'])

.required('Type is required'),

value: yup

.string()

.required('Value is required')

.when('type', {

is: 'A',

then: yup

.string()

.matches(ipv4Regex, 'Invalid IPv4 address'),

otherwise: yup

.string()

.matches(domainRegex, 'Invalid CNAME target (domain name)'),

}),

ttl: yup.number().default(3600).min(0),

});

**✅ Benefits of Regex Approach for You**

* Full control over rules
* No extra dependency
* Easily adaptable (e.g., add IPv6 or IDN later)
* Faster validation (regex is native)

Would you like:

* A utility function to wrap and reuse these regex patterns?
* Tests for these validators?

Let me know how deep you want to go.

4o

**You said:**

For query logging, can I do a fire-and-forget style logging in the background without blocking the response? Is setImmediate() enough or should I use a queue?

**ChatGPT said:**

Great question — and one that touches on **performance**, **reliability**, and **scalability**.

**✅ Goal**

You want to **log DNS queries (or API activity)** *without blocking* the main response — e.g., **fire-and-forget logging**.

**⚡ Option 1: setImmediate() / process.nextTick() / plain Promise.resolve().then()**

**✅ Pros:**

* Super lightweight
* Doesn’t block the response
* Good for **very fast, low-risk tasks** (like writing to in-memory logs or appending to a file)

**⚠️ Cons:**

* Still runs in the same event loop tick queue
* If the logging code fails (e.g., DB error), it can throw unhandled promise rejections
* No retry mechanism or resilience

**Example:**

ts

CopyEdit

res.status(200).json({ success: true });

// Fire-and-forget logging

setImmediate(() => {

logQuery({ userId, query }).catch((err) => {

console.error('Failed to log query', err);

});

});

✅ Fine for simple, non-critical logging in small apps.

**🔁 Option 2: In-Memory Logging Queue (Custom)**

Maintain a lightweight **queue in memory** (array or better yet a ring buffer) and flush in batches on a timer.

**✅ Pros:**

* Doesn’t block request
* Resilient to spikes (batching)
* Simple to implement
* Can retry/log errors

**⚠️ Cons:**

* Data can be lost if the server crashes
* Needs concurrency protection if writing to DB

**Use case: You want a better-than-fire-and-forget but don’t want Redis/Kafka/etc.**

**🚀 Option 3: Job Queue (e.g., BullMQ with Redis)**

Use a job queue (like **BullMQ**, **Bree**, or **RabbitMQ**) and push query logs as jobs.

**✅ Pros:**

* Fully decoupled logging
* Handles retries, failure logs, delays, etc.
* Production-grade scalability

**⚠️ Cons:**

* Requires Redis or a queue backend
* More setup

**Use if:**

* Logging is critical (e.g., auditing)
* You want background processing (e.g., alerts, metrics, analytics)

**✅ Recommendation Summary**

| **Approach** | **Use Case** | **Blocking?** | **Reliable?** | **Complexity** |
| --- | --- | --- | --- | --- |
| setImmediate() | Small project, simple logging | No | No | Very Low |
| In-memory queue | Mid-scale, needs batching, better error handling | No | Somewhat | Medium |
| Redis/Bull queue | Scalable, async processing, production systems | No | Yes | Higher |

**✅ What Should You Use?**

For your **mini DNS API backend assessment**, this would be my suggestion:

* Use setImmediate() **initially** for simplicity.
* Wrap it in a logging function that can **evolve later** into a queue or job.

**Example:**

ts

CopyEdit

function logQueryAsync(query: QueryLog) {

setImmediate(async () => {

try {

await db('query\_logs').insert(query);

} catch (err) {

console.error('Log insert failed:', err);

}

});

}

This way, you can replace setImmediate() with a proper queue later — without touching the rest of your app.

Let me know if you want:

* A reusable log service with pluggable strategies (e.g., memory, Redis, file)?
* A batching version of the in-memory logger?

Happy to help!

4o

**You said:**

how about using void insrtead of setImmediate

**ChatGPT said:**

Using void to fire-and-forget an async function — like this:

ts

CopyEdit

void logQuery(queryData);

— is **valid and clean**, and can be a better option than setImmediate() in many cases, *with some caveats*.

**✅ What void Does**

In this context:

ts

CopyEdit

async function logQuery(data: QueryLog) {

await db('query\_logs').insert(data);

}

void logQuery(queryData); // fire and forget

* void discards the returned Promise.
* It **prevents TypeScript** or linters from warning about an unhandled await.
* It still runs the async function **asynchronously**, without blocking the request.

**✅ Pros of Using void**

| **Feature** | **Benefit** |
| --- | --- |
| ✅ Simpler than setImmediate | No extra call stack logic or scheduling needed |
| ✅ Fully async | The function runs in the microtask queue (Promise) |
| ✅ TypeScript-safe | Avoids “ignored Promise” warnings |
| ✅ Cleaner code | Easier to reason about |

**⚠️ Caveats**

| **Issue** | **Consideration** |
| --- | --- |
| ❌ No error handling | If the promise rejects and is unhandled, it may crash (Node ≥ v15 throws on unhandled rejections by default) |
| ❌ No delay in execution | It runs **immediately after the current call stack**, unlike setImmediate which defers to the event loop |

➡️ **Always add a .catch()** if you're not 100% okay with silent failures:

ts

CopyEdit

void logQuery(queryData).catch((err) => {

console.error('Failed to log query:', err);

});

**🔄 void vs setImmediate**

| **Feature** | **void** | **setImmediate()** |
| --- | --- | --- |
| Async timing | Runs in **microtask queue** (soon) | Runs in **macrotask queue** (later) |
| Stack behavior | Doesn't defer | Defers to end of event loop |
| Error handling | Must use .catch() | Must wrap in try/catch |
| Use case | ✅ Preferred for async functions | Useful if you want **delay** |

**✅ Final Recommendation**

Use this:

ts

CopyEdit

void logQuery(data).catch(err => console.error('Log failed:', err));

* It’s **clean**, **non-blocking**, **correct**, and works well in almost all real-world backend cases.
* Only use setImmediate() if you want a *delayed* execution (e.g., after releasing I/O).

Let me know if you want to wrap this in a safe utility or turn it into a background task runner.