CRUD

This page describes how to perform CRUD operations with your generated Prisma Client API. CRUD is an acronym that stands for:

- Create
- Read
- Update
- Delete

Refer to the Prisma Client API reference documentation for detailed explanations of each method.

Example schema

All examples are based on the following schema:

Expand for sample schema

For **relational databases**, use `db push` command to push the example schema to your own database

```
$ npx prisma db push
```

For **MongoDB**, ensure your data is in a uniform shape and matches the model defined in the Prisma schema.

Create

Create a single record

The following query creates (`create()`) a single user with two fields:

```
})
```

Show query results

The user's `id` is auto-generated, and your schema determines which fields are mandatory.

Create a single record using generated types

The following example produces an identical result, but creates a `UserCreateInput` variable named `user` *outside* the context of the `create()` query. After completing a simple check ("Should posts be included in this `create()` query?"), the `user` variable is passed into the query:

```
import { PrismaClient, Prisma } from '@prisma/client'

const prisma = new PrismaClient()

async function main() {
  let includePosts: boolean = false
  let user: Prisma.UserCreateInput

  // Check if posts should be included in the query
  if (includePosts) {
    user = {
      email: 'elsa@prisma.io',
      name: 'Elsa Prisma',
      posts: {
        create: {
          title: 'Include this post!',
        },
      },
    }
  } else {
    user = {
      email: 'elsa@prisma.io',
      name: 'Elsa Prisma',
    }
  }

  // Pass 'user' object into query
  const createUser = await prisma.user.create({ data: user })
}

main()
```

For : Generated types.

Our site uses cookies to improve your experience, analyze site usage and assist in our marketing efforts.

Cr

Pris and later.

The skips any duplicates (`email` must be unique):

```
const createMany = await prisma.user.createMany({
  data: [
    { name: 'Bob', email: 'bob@prisma.io' },
    { name: 'Bobo', email: 'bob@prisma.io' }, // Duplicate unique key!
    { name: 'Yewande', email: 'yewande@prisma.io' },
    { name: 'Angelique', email: 'angelique@prisma.io' },
  ],
  skipDuplicates: true, // Skip 'Bobo'
})
```

Show query results

```
{
  count: 3
}
```

WARNING

Note `skipDuplicates` is not supported when using MongoDB, SQLServer, or SQLite.

`createMany()` uses a single `INSERT INTO` statement with multiple values, which is generally more efficient than a separate `INSERT` per row:

```
BEGIN
INSERT INTO "public"."User" ("id","name","email","profileViews","role","coinflips","testing"
COMMIT
SELECT "public"."User"."country", "public"."User"."city", "public"."User"."email", SUM("publ
```

> **Note**: Multiple `create()` statements inside a `$transaction` results in multiple `INSERT` statements.

The following video demonstrates how to use `createMany()` and faker.js to seed a database with sample data:

this content

Create records and connect or create related records

See Working with relations > Nested writes for information about creating a record and one or more related records at the same time.

Create and return multiple records

INFO

This feature is available in Prisma ORM version 5.14.0 and later for PostgreSQL, CockroachDB and SQLite.

You can use `createManyAndReturn()` in order to create many records and return the resulting objects.

```
const users = await prisma.user.createManyAndReturn({
  data: [
    { name: 'Alice', email: 'alice@prisma.io' },
    { name: 'Bob', email: 'bob@prisma.io' },
  ],
})
```

Show query results

WARNING

`relationLoadStrategy: join` is not available when using `createManyAndReturn()` .

Read

Get record by ID or unique identifier

The following queries return a single record (`findUnique()`) by unique identifier or ID:

```
// By unique identifier
const user = await prisma.user.findUnique({
  where: {
    email: 'elsa@prisma.io',
  },
})

// By ID
const user = await prisma.user.findUnique({
  where: {
    id: 99,
  },
})
```

If you are using the MongoDB connector and your underlying ID type is `ObjectId`, you can use the string representation of that `ObjectId`:

```
// By ID
const user = await prisma.user.findUnique({
  where: {
    id: '60d5922d00581b8f0062e3a8',
  },
})
```

Get all records

The following `findMany()` query returns *all* `User` records:

```
const users = await prisma.user.findMany()
```

You can also paginate your results.

Get the first record that matches a specific criteria

The following `findFirst()` query returns the *most recently created user* with at least one post that has more than 100 likes:

1. Order users by descending ID (largest first) - the largest ID is the most recent
2. [illegible] post that has more than 100 likes

```
const findUser = await prisma.user.findFirst({
  where: {
    posts: {
      some: {
        likes: {
          gt: 100,
        },
      },
    },
  },
  orderBy: {
    id: 'desc',
  },
})
```

Get a filtered list of records

Prisma Client supports filtering on record fields and related record fields.

Filter by a single field value

The following query returns all `User` records with an email that ends in `"prisma.io"`:

```
const users = await prisma.user.findMany({
  where: {
    email: {
      endsWith: 'prisma.io',
    },
  },
})
```

Filter by multiple field values

The following query uses a combination of operators to return users whose name start with `E` *or* administrators with at least 1 profile view:

```
const users = await prisma.user.findMany({
  where: {
    OR: [
      {
        name: {
          startsWith: 'E',
        },
      },
      {
        AND: {
          profileViews: {
            gt: 0,
          },
          role: {
            equals: 'ADMIN',
          },
        },
      },
    ],
  },
})
```

Filter by related record field values

The following query returns users with an email that ends with `prisma.io` *and* have at least *one* post (`some`) that is not published:

```
const users = await prisma.user.findMany({
  where: {
    email: {
      endsWith: 'prisma.io',
    },
    posts: {
      some: {
        published: false,
      },
    },
  },
})
```

See [Working with relations]() for more examples of filtering on related field values.

Sel

The `email` and `name` fields of a specific `Use`

```
const user = await prisma.user.findUnique({
  where: {
    email: 'emma@prisma.io',
  },
  select: {
    email: true,
    name: true,
  },
})
```

Show query results

For more information about including relations, refer to:

- Select fields
- Relation queries

Select a subset of related record fields

The following query uses a nested `select` to return:

- The user's `email`
- The `likes` field of each post

```
const user = await prisma.user.findUnique({
  where: {
    email: 'emma@prisma.io',
  },
  select: {
    email: true,
    posts: {
      select: {
        likes: true,
      },
    },
  },
})
```

Show query results

For more information about including relations, see Select fields and include relations.

Sel

See ield values.

Inc

The following query returns all `ADMIN` users and includes each user's posts in the result:

```
const users = await prisma.user.findMany({
  where: {
    role: 'ADMIN',
  },
  include: {
    posts: true,
  },
})
```

Show query results

For more information about including relations, see Select fields and include relations.

Include a filtered list of relations

See Working with relations to find out how to combine `include` and `where` for a filtered list of relations - for example, only include a user's published posts.

Update

Update a single record

The following query uses `update()` to find and update a single `User` record by `email`:

```
const updateUser = await prisma.user.update({
  where: {
    email: 'viola@prisma.io',
  },
  data: {
    name: 'Viola the Magnificent',
  },
})
```

Show query results

Update multiple records

The ords that contain `prisma.io`:

```
const updateUsers = await prisma.user.updateMany({
  where: {
    email: {
      contains: 'prisma.io',
    },
  },
  data: {
    role: 'ADMIN',
  },
})
```

Show query results

Update and return multiple records

INFO

This feature is available in Prisma ORM version 6.2.0 and later for PostgreSQL, CockroachDB, and SQLite.

You can use `updateManyAndReturn()` in order to update many records and return the resulting objects.

```
const users = await prisma.user.updateManyAndReturn({
  where: {
    email: {
      contains: 'prisma.io',
    }
  },
  data: {
    role: 'ADMIN'
  }
})
```

Show query results

WARNING

`relationLoadStrategy: join` is not available when using `updateManyAndReturn()` .

Up

The vith a specific email address, or create tha

```
const upsertUser = await prisma.user.upsert({
  where: {
    email: 'viola@prisma.io',
  },
  update: {
    name: 'Viola the Magnificent',
  },
  create: {
    email: 'viola@prisma.io',
    name: 'Viola the Magnificent',
  },
})
```

Show query results

INFO

From version 4.6.0, Prisma Client carries out upserts with database native SQL commands where possible. Learn more.

Prisma Client does not have a `findOrCreate()` query. You can use `upsert()` as a workaround. To make `upsert()` behave like a `findOrCreate()` method, provide an empty `update` parameter to `upsert()`.

WARNING

A limitation to using `upsert()` as a workaround for `findOrCreate()` is that `upsert()` will only accept unique model fields in the `where` condition. So it's not possible to use `upsert()` to emulate `findOrCreate()` if the `where` condition contains non-unique fields.

Update a number field

Use atomic number operations to update a number field **based on its current value** - for example, increment or multiply. The following query increments the `views` and `likes` fields by `1`:

```
const updatePosts = await prisma.post.updateMany({
  data: {
    views: {
      increment: 1,
    },
```

Connect and disconnect related records

Refer to Working with relations for information about disconnecting (`disconnect`) and connecting (`connect`) related records.

Delete

Delete a single record

The following query uses `delete()` to delete a single `User` record:

```
const deleteUser = await prisma.user.delete({
  where: {
    email: 'bert@prisma.io',
  },
})
```

Attempting to delete a user with one or more posts result in an error, as every `Post` requires an author - see cascading deletes.

Delete multiple records

The following query uses `deleteMany()` to delete all `User` records where `email` contains `prisma.io` :

```
const deleteUsers = await prisma.user.deleteMany({
  where: {
    email: {
      contains: 'prisma.io',
    },
  },
})
```

Attempting to delete a user with one or more posts result in an error, as every `Post` requires an author - see cascading deletes.

Delete all records

The following query uses `deleteMany()` to delete all `User` records:

Our site uses cookies to improve your experience, analyze site usage and assist in our marketing efforts.

Be ords (such as posts). In this case, you nee

Ca

⚠ **WARNING**

In 2.26.0 and later it is possible to do cascading deletes using the **preview feature** referential actions.

The following query uses `delete()` to delete a single `User` record:

```
const deleteUser = await prisma.user.delete({
  where: {
    email: 'bert@prisma.io',
  },
})
```

However, the example schema includes a **required relation** between `Post` and `User`, which means that you cannot delete a user with posts:

```
The change you are trying to make would violate the required relation 'PostToUser' between t
```

To resolve this error, you can:

- Make the relation optional:

```
  model Post {
    id       Int   @id @default(autoincrement())
+   author   User? @relation(fields: [authorId], references: [id])
+   authorId Int?
-   author   User  @relation(fields: [authorId], references: [id])
-   authorId Int
  }
```

- Change the author of the posts to another user before deleting the user.
- Delete a user and all their posts with two separate queries in a transaction (all queries must succeed):

```
const deletePosts = prisma.post.deleteMany({
  where: {
    authorId: 7,
  },
```

```
const transaction = await prisma.$transaction([deletePosts, deleteUser])
```

Delete all records from all tables

Sometimes you want to remove all data from all tables but keep the actual tables. This can be particularly useful in a development environment and whilst testing.

The following shows how to delete all records from all tables with Prisma Client and with Prisma Migrate.

Deleting all data with `deleteMany()`

When you know the order in which your tables should be deleted, you can use the `deleteMany` function. This is executed synchronously in a `$transaction` and can be used with all types of databases.

```
const deletePosts = prisma.post.deleteMany()
const deleteProfile = prisma.profile.deleteMany()
const deleteUsers = prisma.user.deleteMany()

// The transaction runs synchronously so deleteUsers must run last.
await prisma.$transaction([deleteProfile, deletePosts, deleteUsers])
```

✅ **Pros**:

- Works well when you know the structure of your schema ahead of time
- Synchronously deletes each tables data

❌ **Cons**:

- When working with relational databases, this function doesn't scale as well as having a more generic solution which looks up and `TRUNCATE` s your tables regardless of their relational constraints. Note that this scaling issue does not apply when using the MongoDB connector.

> **Note**: The `$transaction` performs a cascading delete on each models table so they have to be called in order.

Deleting all data with raw SQL / `TRUNCATE`

If you are comfortable working with raw SQL, you can perform a `TRUNCATE` query on a table using `$ex`

In t a `TRUNCATE` on a Postgres database by RUNCATES all tables in a single query.

The a MySQL database. In this instance the executed, before being reinstated once finished. The whole process is run as a `$transaction`

PostgreSQL MySQL

```
const tablenames = await prisma.$queryRaw<
  Array<{ tablename: string }>
>`SELECT tablename FROM pg_tables WHERE schemaname='public'`

const tables = tablenames
  .map(({ tablename }) => tablename)
  .filter((name) => name !== '_prisma_migrations')
  .map((name) => `"public"."${name}"`)
  .join(', ')

try {
  await prisma.$executeRawUnsafe(`TRUNCATE TABLE ${tables} CASCADE;`)
} catch (error) {
  console.log({ error })
}
```

✅ **Pros**:

- Scalable
- Very fast

❌ **Cons**:

- Can't undo the operation
- Using reserved SQL key words as tables names can cause issues when trying to run a raw query

Deleting all records with Prisma Migrate

If you use Prisma Migrate, you can use `migrate reset`, this will:

1. Drop the database
2. Create a new database
3. Apply migrations
4. Seed the database with data

Advanced query examples

Cr

-
-
-

```
const u = await prisma.user.create({
  include: {
    posts: {
      include: {
        categories: true,
      },
    },
  },
  data: {
    email: 'emma@prisma.io',
    posts: {
      create: [
        {
          title: 'My first post',
          categories: {
            connectOrCreate: [
              {
                create: { name: 'Introductions' },
                where: {
                  name: 'Introductions',
                },
              },
              {
                create: { name: 'Social' },
                where: {
                  name: 'Social',
                },
              },
            ],
          },
        },
        {
          title: 'How to make cookies',
          categories: {
            connectOrCreate: [
              {
                create: { name: 'Social' },
                where: {
                  name: 'Social',
                },
              },
              {
                create: { name: 'Cooking' },
```

```
    },
  },
  })
```

Edit this page on GitHub