Prisma 1 is in maintenance mode. Learn more in the Prisma 2 documentation

Datamodel & Migrations

Datamodel (MySQL)

Overview

The datamodel of your service configuration has two major roles:

- **Define the underlying database schema** (the models and fields are mapped to database tables).
- It is the foundation for the **auto-generated CRUD operations** of your Prisma client.

The datamodel is written using a subset of the GraphQL Schema Definition Language (SDL) and stored in one or more `.prisma`-files. These `.prisma`-files need to be referenced in your `prisma.yml` under the `datamodel` property. For example:

```
endpoint: __YOUR_PRISMA_ENDPOINT__
datamodel: datamodel.prisma
```

COPY

Building blocks of the datamodel

There are several available building blocks to shape your datamodel:

- **Types** consist of multiple fields and typically represent entities from your application domain (e.g. `User`, `Car`, `Order`). Each type in your datamodel is mapped to a database table and CRUD operations are exposed in the generated Prisma client API.
- **Relations** describe *relationships* between types.
- **Directives** covering different use cases such as type constraints or cascading delete behaviour.

Example

A simple example `datamodel.prisma` file:

```
type Tweet {
  id: ID! @id
  createdAt: DateTime! @createdAt
  text: String!
  owner: User! @relation(link: INLINE)
  location: Location!
}

type User {
  id: ID! @id
  createdAt: DateTime! @createdAt
  updatedAt: DateTime! @updatedAt
  handle: String! @unique
  name: String
  tweets: [Tweet!]!
}

type Location {
  id: ID! @id
  latitude: Float!
  longitude: Float!
}
```

This example illustrates a few important concepts when working with your datamodel:

- The three types `Tweet`, `User` and `Location` are mapped to database tables.
- There is a *bidirectional* relation between `User` and `Tweet` (via the `owner` and `tweets` fields). The relation is represented via a *foreign key* on the `Tweet` type.
- There is a *unidirectional* relation from `Tweet` to `Location` (via the `location` field).
- Except for the `name` field on `User`, all fields are *required* in the datamodel (as indicated by the `!` following the type).
- The fields annotated with the `@id`, `@createdAt` and `@updatedAt` directives are managed by Prisma and *read-only* in the exposed Prisma API.
- The `@unique` directive expresses a *unique constraint*, meaning Prisma ensures that there never will be two records with the same values for the annotated field.

Creating and updating your datamodel is as simple as writing and saving the datamodel file. Once you're happy with your datamodel, you can save the file and apply the changes to your Prisma service by running `prisma1 deploy`:

```
$ prisma1 deploy

Changes:

  Tweet (Type)
  + Created type `Tweet`
  + Created field `id` of type `ID!`
  + Created field `createdAt` of type `DateTime!`
  + Created field `text` of type `String!`
  + Created field `owner` of type `User!`
  + Created field `location` of type `Location!`

  User (Type)
  + Created type `User`
  + Created field `id` of type `ID!`
  + Created field `createdAt` of type `DateTime!`
  + Created field `updatedAt` of type `DateTime!`
  + Created field `handle` of type `String!`
  + Created field `name` of type `String`
  + Created field `tweets` of type `[Tweet!]!`

  Location (Type)
  + Created type `Location`
  + Created field `id` of type `ID!`
  + Created field `latitude` of type `Float!`
  + Created field `longitude` of type `Float!`

  TweetToUser (Relation)
  + Created an inline relation between `Tweet` and `User` in the column `owner`

  LocationToTweet (Relation)
  + Created an inline relation between `Location` and `Tweet` in the column `lo

Applying changes 1.1s
```

Files

You can write your datamodel in a single `.prisma`-file or split it accross multiple ones.

The `.prisma`-files containing the datamodel need to be referenced in your `prisma.yml` under the `datamodel` property. For example:

```
datamodel:
  - user.prisma
  - order.prisma
```

If there is only a single file that defines the datamodel, it can be specified as follows:

```
datamodel: datamodel.prisma
```

Object types

An *object type* (or short *type*) defines the structure for one *model* in your datamodel. It is used to represent *entities* from your *application domain*.

Each object type is mapped to the database. For relational databases, one *table* is created per type. For schemaless databases, an equivalent structure is used (e.g. a *document*). Note that Prisma enforces a schema even for schemaless databases!

Data Model

Generated Database Tables

A type has a *name* and one or multiple *fields*. Type names can only contain **alphanumeric characters** and need to start with an uppercase letter. They can contain **at most 64 characters**.

An instantiation of a type is called a *node*. This term refers to a node inside your *data graph*.

Defining an object type

An object type is defined in the datamodel with the keyword `type`:

```
type Article {
  id: ID! @id
  title: String!
  text: String
  isPublished: Boolean! @default(value: false)
}
```

The type defined above has the following properties:

- Name: `Article`
- Fields: `id`, `title`, `text` and `isPublished` (with the default value `false`)

`id` and `title` and `isPublished` are required (non-nullable) as indicated by the `!` following the type, `text` is nullable.

Generated API operations for types

The types in your datamodel affect the available operations in the Prisma API. Here is an overview of the generated CRUD and realtime operations for every type in your Prisma API:

- Queries let you fetch one or many nodes of that type
- Mutations let you create, update or delete nodes of that type
- Subscriptions let you get notified of changes to nodes of that type (i.e. new nodes are *created* or existing nodes are *updated* or *deleted*)

Fields

Fields are the building blocks of a type, giving a node its *shape*. Every field is referenced by its name and is either scalar or a relation field.

Field names can only contain **alphanumeric characters** and need to start with a lowercase letter. They can contain **at most 64 characters**.

Scalar fields

STRING

A `String` holds text. This is the type you would use for a username, the content of a blog post or anything else that is best represented as text.

String values are currently limited to 256KB in size on Demo servers. This limit can be increased on other clusters using the cluster configuration.

Here is an example of a `String` scalar definition:

```
type User {
  name: String
}
```

INTEGER

An `Int` is a number that cannot have decimals. Use this to store values such as the weight of an ingredient required for a recipe or the minimum age for an event.

`Int` values range from -2147483648 to 2147483647.

Here is an example of an `Int` scalar definition:

```
type User {
  age: Int
}
```

FLOAT

A `Float` is a number that can have decimals. Use this to store values such as the price of an item in a store or the result of complex calculations.

In queries or mutations, `Float` fields have to be specified without any enclosing characters and an optional decimal point: `float: 42`, `float: 4.2`.

Here is an example of a `Float` scalar definition:

```
type Item {
  price: Float
}
```

BOOLEAN

A `Boolean` can have the value `true` or `false`. This is useful to keep track of settings such as whether the user wants to receive an email newsletter or if a recipe is appropriate for vegetarians.

Here is an example of a `Boolean` scalar definition:

```
type User {
  overEighteen: Boolean
}
```

DATETIME

The `DateTime` type can be used to store date and/or time values. A good example might be a person's date of birth or the time/data when a specific event is happening.

Here is an example of a `DateTime` scalar definition:

```
type User {
  birthday: DateTime
}
```

When used as arguments in an operation, `DateTime` fields have to be specified in ISO 8601 format and are typically passed as strings, here are a few examples:

- `"2015"`
- `"2015-11"`
- `"2015-11-22"`
- `"2015-11-22T13:57:31.123Z"`.

ENUM

Like a `Boolean` an Enum can have one of a predefined set of values. The difference is that *you* can define the possible values (whereas for a `Boolean` the options are restriced to `true` and `false`). For example you could specify how an article should be formatted by creating an Enum with the possible values `COMPACT`, `WIDE` and `COVER`.

Enum values can only contain **alphanumeric characters and underscores** and need to start with an uppercase letter. The name of an enum value can be used in query filters and mutations. They can contain **at most 191 characters**.

Here is an example of an enum definition:

```
enum ArticleFormat {
  COMPACT
  WIDE
  COVER
}

type Article {
  format: ArticleFormat
}
```

JSON

Sometimes you might need to store arbitrary JSON values for loosely structured data. The `Json` type makes sure that it is actually valid JSON and returns the value as a parsed JSON object/array instead of a string.

Json values are currently limited to 256KB in size.

Here is an example of a `Json` definition:

```
type Item {
  data: Json
}
```

ID

An `ID` value is a generated unique 25-character string based on cuid.

Fields of type `ID` that are annotated with the `@id` directive are system fields and maintained by Prisma. Only one `ID` field per model can be annotated with `@id`:

```
type User {
  id: ID! @id
}
```

Type modifiers

In a field definition, a type can be annotated with a *type modifier*. SDL supports two type modifiers:

- **Lists**: Annotate the type with a pair of enclosing `[]`, e.g. `friends: [User]`
- **Required fields**: Annotate the type with a `!`, e.g. `name: String!`

LIST

Scalar fields can be marked with the list field type. A field of a relation that has the many multiplicity will also be marked as a list.

You will often find list definitions looking similar to this:

```
type Article {
  tags: [String!]!
}
```

Notice the two `!` type modifiers, here is what they express:

- The first `!` type modifier (right after `String`) means that no item in the list can be `null`, e.g. this value for `tags` would not be valid: `["Software", null, "Prisma"]`
- The second `!` type modifier (after the closing square bracket) means that the list itself can never be `null`, it might be *empty* though. Consequently, `null` is not a valid value for the `tags` field but `[]` is.

REQUIRED

Fields can be marked as required (also referred to as "non-nullable"). Required fields are marked using a `!` after the field's type:

```
type User {
  name: String!
}
```

Field constraints

Fields can be configured with field constraints to add further semantics and enforce certain rules in your datamodel.

UNIQUE

Setting the *unique* constraint makes sure that two records of the model in question cannot have the same value for a certain field. The only exception is the `null` value, meaning that multiple records can have the value `null` without violating the constraint. Unique fields have a unique *index* applied in the underlying database.

A typical example would be an `email` field on a `User` models where the assumption is that every `User` should have a globally unique email address.

Only the first 191 characters in a String field are considered for uniqueness and the unique check is *case insensitive*. Storing two different strings is not possible if the first 191 characters are the same or if they only differ in casing.

To mark a field as unique, simply append the `@unique` directive to its definition:

```
type User {
  id: ID! @id
  email: String! @unique
  name: String!
}
```

For every field that's annotated with `@unique`, you're able to query the corresponding record by providing a value for that field as a query argument.

For example, considering the above datamodel, you can now retrieve a particular `User` node by its `email` address:

TypeScript JavaScript Flow Go

COPY

```
const user = await prisma.user({
  email: 'alice@prisma.io',
})
```

MORE CONSTRAINTS

More database constraints will be added soon. Please join the discussion in this feature request if you have wish to see certain constraints implemented in Prisma.

Default value

You can set a *default value* for non-list scalar fields. The value will be applied to newly created records when no value was supplied during the `create`-operation.

To specify a default value for a field, you can use the `@default` directive:

```
type Story {
  isPublished: Boolean @default(value: false)
  someNumber: Int! @default(value: 42)
  title: String! @default(value: "My New Post")
  publishDate: DateTime! @default(value: "2018-01-26")
  status: Status! @default(value: PUBLIC)
}

enum Status {
  PRIVATE
  PUBLIC
}
```

Relations

A *relation* defines the semantics of a connection between two types. Two types in a relation are connected via a relation field. When a relation is ambiguous, the relation field needs to be annotated with the `@relation` directive to disambiguate it.

There are two ways to represent a relation in the underlying database:

- With a **relation table**: Prisma tracks the relation via a dedicated table that contains two columns which refer to the IDs of each model.
- As an **inline relation**: Prisma tracks the relation via a *foreign key* in a column (not available for n:m relations).

1:1 relations

When defining a 1:1 relation between two models, you must add the `@relation` directive to one end of the relation. Otherwise Prisma doesn't know how it should lay out the relation in the underlying database.

INLINE

```
type User {
  id: ID! @id
  profile: Profile! @relation(link: INLINE)
}

type Profile {
  id: ID! @id
  user: User!
}
```

This stores the primary key of `Profile` in the `profile` column on the `User` table:

```
CREATE TABLE 'default$default'."User" (
    "id" varchar(25) NOT NULL,
    "profile" varchar(25),
    PRIMARY KEY ("id")
);
```

RELATION TABLE

Generatic relation table

```
type User {
  id: ID! @id
  profile: Profile! @relation(link: TABLE)
}

type Profile {
  id: ID! @id
```

```
  user: User!
}
```

This creates the following relation table:

```
CREATE TABLE 'default$default'."_PostToUser" (
  "A" varchar(25) NOT NULL,
  "B" varchar(25) NOT NULL
);
```

Relation table with a custom name (prepended with an underscore)

```
type User {
  id: ID! @id
  profile: Profile! @relation(link: TABLE, name: "MyRelation")
}

type Profile {
  id: ID! @id
  user: User!
}
```

This creates the following relation table:

```
CREATE TABLE 'default$default'."_MyRelation" (
  "A" varchar(25) NOT NULL,
  "B" varchar(25) NOT NULL
);
```

Customized relation table with `@relationTable`

```
type User {
  id: ID! @id
  profile: Profile! @relation(link: TABLE, name: "MyRelation")
}

type Profile {
  id: ID! @id
  user: User!
}

type MyRelation @relationTable {
  user: User!
  profile: Profile!
}
```

This creates the following relation table:

```
CREATE TABLE 'default$default'."MyRelation" (
  "profile" varchar(25) NOT NULL,
  "user" varchar(25) NOT NULL
);
```

The `@relationTable` directive should be used when you want to:

- remove the underscore in front of the relation name
- call the foreign key columns in the relation table something else than `A` and `B`

So you need it in the active case when you want to control that layout. More prominently you need it in the passive case, when you want to connect to an existing database that does not match the prisma conventions.

1:n relations

When defining a 1:n relation between two models, the `@relation` directive is optional. Prisma defaults to an inline relation.

INLINE

```
type User {
  id: ID! @id
  posts: [Post!]!
}

type Post {
  id: ID! @id
  author: User! @relation(link: INLINE)
}
```

This stores the primary key of `User` in the `author` column on the `Post` table:

```
CREATE TABLE 'relations$dev1'."Post" (
    "id" varchar(25) NOT NULL,
    "author" varchar(25),
    PRIMARY KEY ("id")
);
```

Note that in this case the `@relation` directive could also be omitted because this is the default behaviour for 1:n relations.

RELATION TABLE

Generatic relation table

```
type User {
  id: ID! @id
  posts: [Post!]!
}

type Post {
  id: ID! @id
  author: User! @relation(link: TABLE)
}
```

This creates the following relation table:

```
CREATE TABLE 'default$default'."_PostToUser" (
  "A" varchar(25) NOT NULL,
  "B" varchar(25) NOT NULL
);
```

Relation table with a custom name (prepended with an underscore)

```
type User {
  id: ID! @id
  posts: [Post!]!
}

type Post {
  id: ID! @id
  user: User! @relation(link: TABLE, name: "MyRelation")
}
```

This creates the following relation table:

```
CREATE TABLE "default$default"."_MyRelation" (
  "A" varchar(25) NOT NULL,
  "B" varchar(25) NOT NULL
);
```

Customized relation table with `@relationTable`

```
type User {
  id: ID! @id
  posts: [Post!]!
}

type Post {
  id: ID! @id
  user: User! @relation(link: TABLE, name: "MyRelation")
}

type MyRelation @relationTable {
  user: User!
  post: Post!
}
```

This creates the following relation table:

```
CREATE TABLE "default$default"."MyRelation" (
    "post" varchar(25) NOT NULL,
    "user" varchar(25) NOT NULL
);
```

n:m relations

INLINE RELATION

Not applicable

RELATION TABLE

Generatic relation table

```
type Category {
  id: ID! @id
  posts: [Post!]!
}

type Post {
  id: ID! @id
  categories: [Category!]! @relation(link: TABLE)
}
```

This creates the following relation table:

```
CREATE TABLE "default$default"."_PostToCategory" (
  "A" varchar(25) NOT NULL,
  "B" varchar(25) NOT NULL
);
```

Relation table with a custom name (prepended with an underscore)

```
type Category {
  id: ID! @id
  posts: [Post!]!
}

type Post {
  id: ID! @id
  categories: [Category!]! @relation(link: TABLE, name: "MyRelation")
}
```

This creates the following relation table:

```
CREATE TABLE "default$default"."_MyRelation" (
  "post" varchar(25) NOT NULL,
  "category" varchar(25) NOT NULL
);
```

Customized relation table with `@relationTable`

```
type Category {
  id: ID! @id
  posts: [Post!]!
}

type Post {
  id: ID! @id
  categories: [Category!]! @relation(link: TABLE, name: "MyRelation")
}

type MyRelation @relationTable {
  user: User!
  post: Post!
}
```

This creates the following relation table:

```
CREATE TABLE "default$default"."MyRelation" (
    "post" varchar(25) NOT NULL,
    "cagtegory" varchar(25) NOT NULL
);
```

Self relations

A relation can also connect a type with itself. It is then referred to as a *self-relation*:

```
type User {
  id: ID! @id
  friends: [User!]!
}
```

A self-relation can also be bidirectional:

```
type User {
  id: ID! @id
  following: [User!]! @relation(name: "FollowRelation")
  followers: [User!]! @relation(name: "FollowRelation")
}
```

Note that in this case the relation needs to be annotated with the `@relation` directive and the `name` argument must be provided.

Required relations

For a *to-one* relation field, you can configure whether it is *required* or *optional*. The `!` type modifier means that this field can never be `null`. A field for the address of a user would therefore be of type `Address` or `Address!`.

Records for a type that contains a required *to-one* relation field can only be created using a declarative nested write to ensure the respective field will not be `null`.

Consider again the following relation:

```
type User {
  id: ID! @id
  car: Car!
}

type Car {
  id: ID! @id
  owner: User!
  color: String!
}
```

A `Car` can never be created without a `User` and the other way around because that would violate the required constraint. You therefore need to create both at the same time using a declarative nested write:

TypeScript JavaScript Flow Go

COPY

```
const newUser = await prisma.createUser({
  car: {
    create: {
      color: 'Yellow',
    },
  },
})
```

Note that a *to-many* relation field is always set to required. For example, a field that contains many user addresses always uses the type `[Address!]!` and can never be of type `[Address!]`, `[Address]!` or `[Address]`.

The `@relation` directive

When defining relations between types, you *can* use the `@relation` directive which provides meta-information about the relation. If a relation is ambiguous, you *must* use the `@relation` directive to disambiguate it.

It can take three arguments:

- `name`: An identifier for this relation, provided as a string.
- `link`: Specifies how the relation should be represented in the underlying database. The input values for this argument are defined as an enum with the following possible values:
  - `INLINE`: The relation is represented with foreign keys.
  - `TABLE`: The relation is represented via a dedicated relation table.
- `onDelete`: Specifies the *deletion behaviour* and enables *cascading deletes*. In case a node with related nodes gets deleted, the deletion behaviour determines what should happen to the related nodes. The input values for this argument are defined as an enum with the following possible values:
  - `SET_NULL` (default): Set the related node(s) to `null`.
  - `CASCADE`: Delete the related node(s). Note that is not possible to set *both* ends of a bidirectional relation to `CASCADE`.

Here is an example of a datamodel where the `@relation` directive is used:

```
type User {
  id: ID! @id
  stories: [Story!]! @relation(name: "StoriesByUser", onDelete: CASCADE)
}

type Story {
  id: ID! @id
  text: String!
  author: User @relation(name: "StoriesByUser")
}
```

The relation is named `StoriesByUser` and the deletion behaviour is as follows:

- When a `User` record gets deleted, all its related `Story` records will be deleted as well.
- When a `Story` record gets deleted, it will simply be removed from the `stories` list on the related `User` record.

> It is currently not possible to rename relations that are specified via the `@relation` directive.

OMITTING THE `@relation` DIRECTIVE

In the simplest case, where a relation between two types is unambiguous and the default deletion behaviour (`SET_NULL`) should be applied, the corresponding relation fields do not have to be annotated with the `@relation` directive. Note that on 1:1 relations, the `@relation` directive is always required because Prisma needs to know what `link` should be used.

Here we are defining a bidirectional *one-to-many* relation between the `User` and `Story` types. Since `onDelete` has not been provided, the default deletion behaviour is used: `SET_NULL`:

```
type User {
  id: ID! @id
  stories: [Story!]!
}
```

```
type Story {
  id: ID! @id
  text: String!
  author: User
}
```

The deletion behaviour in this example is as follows:

- When a `User` record gets deleted, the `author` field on all its related `Story` records will be set to `null`. Note that if the `author` field was marked as required, the operation would result in an error.
- When a `Story` record gets deleted, it will simply be removed from the `stories` list on the related `User` record.

USING THE `name` ARGUMENT OF THE `@relation` DIRECTIVE

In certain cases, your datamodel may contain ambiguous relations. For example, consider you not only want a relation to express the "author-relationship" between `User` and `Story`, but you also want a relation to express which `Story` nodes have been *liked* by a `User`.

In that case, you end up with two different relations between `User` and `Story`! In order to disambiguate them, you need to give the relation a name:

```
type User {
  id: ID! @id
  writtenStories: [Story!]! @relation(name: "WrittenStories")
  likedStories: [Story!]! @relation(name: "LikedStories")
}

type Story {
  id: ID! @id
  text: String!
  author: User! @relation(name: "WrittenStories")
  likedBy: [User!]! @relation(name: "LikedStories")
}
```

If the `name` wasn't provided in this case, there would be no way to decide whether `writtenStories` should relate to the `author` or the `likedBy` field.

USING THE `onDelete` ARGUMENT OF THE `@relation` DIRECTIVE

As mentioned above, you can specify a dedicated deletion behaviour for the related nodes. That's what the `onDelete` argument of the `@relation` directive is for.

Consider the following example:

```
type User {
  id: ID! @id
  comments: [Comment!]! @relation(name: "CommentAuthor", onDelete: CASCADE)
  blog: Blog @relation(name: "BlogOwner", onDelete: CASCADE)
}

type Blog {
  id: ID! @id
  comments: [Comment!]! @relation(name: "Comments", onDelete: CASCADE)
  owner: User! @relation(name: "BlogOwner", onDelete: SET_NULL)
}

type Comment {
  id: ID! @id
  blog: Blog! @relation(name: "Comments", onDelete: SET_NULL)
  author: User @relation(name: "CommentAuthor", onDelete: SET_NULL)
}
```

Let's investigate the deletion behaviour for the three types:

- When a `User` node gets deleted,
  - all related `Comment` nodes will be deleted.
  - the related `Blog` node will be deleted.
- When a `Blog` node gets deleted,
  - all related `Comment` nodes will be deleted.
  - the related `User` node will have its `blog` field set to `null`.
- When a `Comment` node gets deleted,
  - the related `Blog` node continues to exist and the deleted `Comment` node is removed from its `comments` list.
  - the related `User` node continues to exist and the deleted `Comment` node is removed from its `comments` list.

SDL directives

Directives are used to provide additional information and add specific behaviours to the types in your datamodel. They look like this: `@name(argument: "value")` or simply `@name` when there are no arguments.

`@id`

A record will automatically get assigned a globally unique identifier when it's created, this identifier is stored in the field annotated with the `@id` directive:

```
type User {
  id: ID! @id
}
```

An auto-generated `id` has the following properties:

- Consists of 25 alphanumeric characters (letters are always lowercase)
- Always starts with a (lowercase) letter, e.g. `c`
- Follows cuid (*collision resistant unique identifiers*) scheme

The type of an `id` field can be:

- `ID`
- `Int`

Please note that the `UUID` type is not currently supported with MySQL.

`@createdAt` and `@updatedAt`

The datamodel further provides two special field directives which you can add to your types:

- `createdAt: DateTime! @createdAt`: Stores the exact date and time for when a record of this object type was *created*.

- `updatedAt: DateTime! @updatedAt` : Stores the exact date and time for when a record of this object type was *last updated*.

If you to add this behaviour to your model, you can simply add the corresponding directives to the model definition, for example:

```
type User {
  id: ID! @id
  createdAt: DateTime! @createdAt
  updatedAt: DateTime! @updatedAt
}
```

`@unique`

The `@unique` directive marks a scalar field as unique. Unique fields will have a unique *index* applied in the underlying database.

```
type User {
  email: String @unique
}
```

Find more info about the `@unique` directive above.

`@db`

The `@db` directive can be applied to types/fields to determine the name of the table/column in the underlying database. For example:

```
type User @db(name: "user") {
  id: ID! @id
  name: String! @db(name: "full_name")
}
```

In this case, the underlying table for the `User` model is called `user` and the column representing the `name` field is called `full_name`.

`@default`

The `@default(value: String!)` directive sets a default value for a scalar field:

```
type Post {
  title: String! @default(value: "New Post")
  published: Boolean! @default(value: false)
  someNumber: Int! @default(value: 42)
}
```

`@scalarList`

This `@scalarList(strategy: STRATEGY!)` directive is required on any scalar list field. The only valid argument for the `strategy` argument is `RELATION`.

```
type Post {
  tags: [String!]! @scalarList(strategy: RELATION)
}
```

`@relation`

The directive `@relation(name: String, onDelete: ON_DELETE! = SET_NULL)` can be attached to a relation field.

See above for more information.

Naming conventions

Different objects you encounter in a Prisma service like types or relations follow separate naming conventions to help you distinguish them.

Types

The type name determines the name of derived queries and mutations as well as the argument names for nested mutations.

Here are the conventions for naming types:

- Choose type names in **singular**:
  - **Yes**: `type User { ... }`
  - **No**: `type Users { ... }`

Scalar and relation fields

The name of a scalar field is used in queries and in query arguments of mutations. The name of relation fields follows the same conventions and determines the argument names for relation mutations.Relation field names can only contain **alphanumeric characters** and need to start with an uppercase letter. They can contain **at most 64 characters**.

Field names are unique per type.

Here are the conventions for naming fields:

- Choose plural names for list fields:
  - **Yes**: `friends: [User!]!`
  - **No**: `friendList: [User!]!`
- Choose singular names for non-list fields:
  - **Yes**: `post: Post!`
  - **No**: `posts: Post!`

More SDL features

In this section, we describe further SDL features that are not yet supported for data modeling with Prisma.

Interfaces

"Like many type systems, [SDL] supports interfaces. An interface is an abstract type that includes a certain set of fields that a type must include to implement the interface." From the official Documentation

To learn more about when and how interfaces are coming to Prisma, check out this feature request.

Union types

"Union types are very similar to interfaces, but they don't get to specify any common fields between the types." From the official Documentation

To learn more about when and how union types are coming to Prisma, check out this feature request.

Prisma © 2018 – 2021