Click Here

This is only possible when retrieving single records, not for lists. Meaning you can not query relation fields of records that are returned in a list, e.g.: const result = await prisma.users().posts() In this example, users() already returns a list, so it's not possible to query the posts relation of each user record inside the list. Prisma is open-source and can be used as a standalone infrastructure component hosted on your favorite cloud provider. Prisma Cloud is an application (used through a CLI & web interface) that provides tools and services around Prisma. When you're using Prisma, usage of Prisma Cloud is optional.The goal of Prisma Cloud is to ease the workflows in Prisma projects. It features a data browser, a deployment history (soon with automatic rollbacks), various team collaboration options as well as cloud provider integrations to make it easy for developers to setup and maintain their Prisma servers.Get started with Prisma Cloud here. Data to be imported needs to adhere to the Normalized Data Format (NDF). As of today, the conversion from any concrete data source (like MySQL, MongoDB or Firebase) to NDF must be performed manually. In the future, the Prisma CLI will support importing from these data sources directly.Here is a general overview of the data import process:+-----------+ +-----------+ +-----------+ | +-----------+ | | | | | | | | | | | | | SQL | | (1) transform | NDF | (2) chunked upload | Prisma | | | MongoDB | | +----------> | | +----------> | | | | JSON | | | | | | | | | | | | +-----------+ | +-----------+ +-----------+ +-----------+ As mentioned above, step 1 has to be performed manually. Step 2 can then be done by either using the raw import API or the prisma1 import command from the CLI.To view the current state of supported transformations in the CLI and submit a vote for the one you need, you can check out this GitHub issue.When uploading files in NDF, you need to provide the import data split across three different kinds of files:nodes: Data for individual nodes (i.e. databases records)lists: Data for a list of nodesrelations: Data for related nodesYou can upload an unlimited number of files for each of these types, but it's recommended that each file should be at most 1 MB large. Otherwise you might run into timeouts.General constraintsWhen importing data, the id field of a node can be at most 25 characters long.IdempotencyNote that import operations are not idempotent. This means running an import always adds data to your service. It never updates existing nodes. This means importing the same dataset multiple times will lead to undefined behaviour.For example, importing a node with the same id more than once will lead to undefined behaviour and likely break your service!Data ValidationThe import API does not perform any validation checks on the data to be imported. When using the CLI to import data, basic validation checks are executed.Importing invalid data leads to undefined behaviour and might break your service! As the service maintainer, you are responsible to ensure the validity of the imported data.Tip: A good way to ensure valid data when importing is inspecting the data of a previous export on a service with an identical datamodel.Data import with the CLIThe Prisma CLI offers the prisma1 import command. It accepts one option:--data (short: -d): A file path to a directory containing the data to be imported (this can either be regular or a zipped directory)Under the hood, the CLI uses the import API that's described in the next section. However, using the CLI provides some major benefits:uploading multiple files at once (rather than having to upload each file individually)leveraging the CLI's authentication mechanism (i.e. you don't need to manually send your authentication token)ability to pause and resume an ongoing importimport from various data sources like MySQL, MongoDB or Firebase (not available yet)Input formatWhen importing data using the CLI, the files containing the data in NDF need to be located in directories called after their type: nodes, lists and relations.NDF files are JSON files following a specific structure, so each file containing import data needs to end on .json. When placed in their respective directories (nodes, lists or relations), the .json-files need to be numbered incrementally, starting with 1, e.g. 1.json. The file name can be prepended with any number of zeros, e.g. 01.json or 0000001.json.ExampleConsider the following file structure defining a Prisma service:. ├── data | ├── lists | | | ├── 0001.json | | | ├── 0002.json | | | └── 0003.json | ├── nodes | | | ├── 0001.json | | | └── 0002.json | └── relations | └── 0001.json ├── datamodel.prisma └── prisma.yml data contains the files data to be imported. Further, all files ending on .json are adhering to NDF. To import the data from these files, you can simply run the following command in the terminal:prisma1 import --data data Data import using the raw import APIThe raw import API is exposed under the /import path of your service's HTTP endpoint. For example: //eu1.prisma.sh/my-app/prod/importOne request can upload JSON data (in NDF) of at most 10 MB in size. Note that you need to provide your authentication token in the HTTP Authorization header of the request!Here is an example curl command for uploading some JSON data (of NDF type nodes):curl ' -H 'Content-Type: application/json' -H 'Authorization: Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJPbmxpbmUgSldUIEJ1aWxkZXIiLCJpYXQiOjE1MTM1OTQzMTEsImV4cCI6MTU0NTEzMDMxMSwiYXVkIjasd3d3LmV4YW1wbGUuY29tiwic3ViIjoianJvY2tldEBleGFtcGxlLmNvbSIsIkdpdmVuTmFtZSI6IkpvaG5ueSIsIlN1cm5hbWUiOiJSb2xlIjpbIk1hbmFnZXIiLCJQcm9qZWN0IEFkbWluaXN0cmF0b3IiXX0.L7DwH7vIfTSmuwfxBI82D64DlgoLBLXOwR5iMjZ_7nI'
-d '{"valueType":"nodes","values":[{"_typeName":"Model0","id":"0","a":"test","b":0,"createdAt":"2017-11-29 14:35:13"},{"_typeName":"Model1","id":"1","a":"test","b":1},{"_typeName":"Model2","id":"2","a":"test","b":2,"createdAt":"2017-11-29 14:35:13"},{"_typeName":"Model0","id":"3","a":"test","b":3},{"_typeName":"Model3","id":"4","a":"test","b":4,"createdAt":"2017-11-29 14:35:13"},{"_typeName":"Model3","id":"5","a":"test","b":5},
{"_typeName":"Model3","id":"6","a":"test","b":6},{"_typeName":"Model4","id":"7"},{"_typeName":"Model4","id":"8","string":"test","int":4,"boolean":true,"dateTime":"1015-11-29 14:35:13","float":13.333,"createdAt":"2017-11-29 14:35:13","updatedAt":"2017-11-29 14:35:13"},{"_typeName":"Model5","id":"9","string":"test","int":4,"boolean":true,"dateTime":"1015-11-29 14:35:13","float":13.333,"createdAt":"2017-11-29 14:35:13","updatedAt":"2017-11-29 14:35:13"}]}' -sSv
The generic version for curl (using placeholders) would look as follows:curl '__SERVICE_ENDPOINT__/import' -H 'Content-Type: application/json' -H 'Authorization: Bearer __JWT_AUTH_TOKEN__' -d '{"valueType":"__NDF_TYPE__","values": __DATA__ }' -sSv Exporting data can be done either using the CLI or the raw export API. In both cases, the downloaded data is formatted in JSON and adheres to the Normalized Data Format (NDF). As the exported data is in NDF, it can directly be imported into a service with an identical schema. This can be useful when test data is needed for a service, e.g. in a dev stage.Data export with the CLIThe Prisma CLI offers the prisma1 export command. It accepts one option:--export-path (short: -e): A file path to a .zip-directory which will be created by the CLI and where the exported data is storedUnder the hood, the CLI uses the export API that's described in the next section. However, using the CLI provides some major benefits:leveraging the CLI's authentication mechanism (i.e. you don't need to manually send your authentication token)writing downloaded data directly to file systemcursor management in case multiple requests are needed to export all application data (when doing this manually you need to send multiple requests and adjust the cursor upon each)Output formatThe data is exported in NDF and will be placed in three directories that are named after the different NDF types: nodes, lists and relations.Data export using the raw export APIThe raw export API is exposed under the /export path of your service's HTTP endpoint. For example: //database.prisma.sh/my-app/prod/exportOne request can download JSON data (in NDF) of at most 10 MB in size. Note that you need to provide your authentication token in the HTTP Authorization header of the request!The endpoint expects a POST request where the body contains JSON with the following contents:{ "fileType": "nodes", "cursor": { "table": 0, "row": 0, "field": 0, "array": 0 } } The values in cursor describe the offsets in the database from where on data should be exported. Note that each response for an export request will return a new cursor with either of two states:Terminated (not full): If all the values for table, row, field and array are returned as -1 it means the export has completed.Non-terminated (_full): If any of the values for table, row, field or array is different from -1, it means the maximum size of 10 MB for this response has been reached. If this happens, you can use the returned cursor values as the input for your next export request.ExampleHere is an example curl command for uploading some JSON data (of NDF type nodes):curl ' -H 'Content-Type: application/json' -H 'Authorization: Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJPbmxpbmUgSldUIEJ1aWxkZXIiLCJpYXQiOjE1MTM1OTQzMTEsImV4cCI6MTU0NTEzMDMxMSwiYXVkIjasd3d3LmV4YW1wbGUuY29tiwic3ViIjoianJvY2tldEBleGFtcGxlLmNvbSIsIkdpdmVuTmFtZSI6IkpvaG5ueSIsIlN1cm5hbWUiOiJSb2xlIjpbIk1hbmFnZXIiLCJQcm9qZWN0IEFkbWluaXN0cmF0b3IiXX0.L7DwH7vIfTSmuwfxBI82D64DlgoLBLXOwR5iMjZ_7nI'
-d '{"fileType":"nodes","cursor":{"table":0,"row":0,"field":0,"array":0}}' -sSv
The Normalized Data Format (NDF) is used as an intermediate data format for import and export in Prisma services. NDF describes a specific structure for JSON.NDF value typesWhen using the NDF, data is split across three different "value types":Nodes: Contains data for the scalar fields of nodesLists: Contains data for list fields of nodesRelations: Contains data to connect two nodes via a relation by their relation fieldsStructureThe structure for a JSON document in NDF is an object with the following two keys:valueType: Indicates the value type of the data in the document (this can be either "nodes", "lists" or "relations")values: Contains the actual data (adhering to the value type) as an arrayThe examples in the following are based on this datamodel:type User { id: String! @unique firstName: String! lastName: String! hobbies: [String!]! partner: User } NodesIn case the valueType is "nodes", the structure for the objects inside the values array is as follows:{ "valueType": "nodes", "values": [{ "_typeName": STRING, "id": STRING, "": ANY, "": ANY, ..., "": ANY } , ...] } The notations expresses that the fields _typeName and id are of type string. _typeName refers to the name of the SDL type from your datamodel. The -placeholders will be the names of the scalar fields of that SDL type.For example, the following JSON document can be used to import the scalar values for two User nodes:{ "valueType": "nodes", "values": [{ "_typeName": "User", "id": "johndoe", "firstName": "John", "lastName": "Doe" }, { "_typeName": "User", "id": "sarahdoe", "firstName": "Sarah", "lastName": "Doe" }] } ListsIn case the valueType is "lists", the structure for the objects inside the values array is as follows:{ "valueType": "lists", "values": [{ "_typeName": STRING, "id": STRING, "": [ANY] } , ...] } The notations expresses that the fields _typeName and id are of type string. _typeName refers to the name of the SDL type from your datamodel. The -placeholder is the name of the of the list fields of that SDL type. Note that in contrast to the scalar list fields, each object can only values only for one field.For example, the following JSON document can be used to import the values for the hobbies list field of two User nodes:{ "valueType": "lists", "values": [{ "_typeName": "User", "id": "johndoe", "hobbies": ["Fishing", "Cooking"] } , { "_typeName": "User", "id": "sarahdoe", "hobbies": ["Biking", "Coding"] }] } RelationsIn case the valueType is "relations", the structure for the objects inside the values array is as follows:{ "valueType": "relations", "values": [{ "_typeName": STRING, "id": STRING, "fieldName": STRING } , { "_typeName": STRING, "id": STRING, "fieldName": STRING }] , ...] } The notations expresses that the fields _typeName, id and fieldName are of type string. _typeName refers to a name of an SDL type from your datamodel. The -placeholder is the name of the of the relation field of that SDL type. Since the goal of the relation data is to connect two nodes via a relation, each element inside the values array by itself is a pair (written as an array which always contains exactly two elements) rather than a single object as was the case for "nodes" and "lists".For example, the following JSON document can be used to create a relation between two User nodes via the partner relation field:{ "valueType": "relations", "values": [{ " _typeName": "User", "id": "johndoe", "fieldName": "partner" } , { " _typeName": "User", "id": "sarahdoe", "fieldName": "partner" }] }] } To get insights into the internals of your Prisma server, you can check the logs of the Docker container.Accessing Docker logsView the raw logs from the running Docker containers:docker-compose logs Verify Docker containersVerify that the containers are running:docker ps You should see an output similar to this:$ docker ps CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES 2b799c529e73 prismagraphql/prisma:1.34 "/bin/sh -c /app/sta..." 17 hours ago Up 7 hours 0.0.0.0:4466->4466/tcp myapp_prisma_1 757dfba212f7 mysql:5.7 "docker-entrypoint.s..." 17 hours ago Up 7 hours 3306/tcp prisma-db This is helpful when you're getting an error message saying Error response from daemon: No such container:Logging slow queriesTo monitor and log slow queries, you can set the following environment variables in your Docker Compose file:SLOW_QUERIES_LOGGING: Enable slow query logging with true or falseSLOW_QUERIES_LOGGING_THRESHOLD: The treshold (in miliseconds) for queries for queries to be logged. To log all queries, simply provide the value "0".Here is an example:version: '3' services: prisma: image: prismagraphql/prisma: _LATEST_PRISMA_VERSION __restart: always ports: - '4466:4466' environment: SLOW_QUERIES_LOGGING: 'true' SLOW_QUERIES_LOGGING_THRESHOLD: '0' PRISMA_CONFIG: | managementApiSecret: __YOUR_MANAGEMENT_API_SECRET__ port: __YOUR_PRISMA_SERVER_PORT__ databases: default: connector: __YOUR_DATABASE_CONNECTOR__ host: __YOUR_DATABASE_HOST__ port: __YOUR_DATABASE_PORT__ user: __YOUR_DATABASE_USER__ password: __YOUR_DATABASE_PASSWORD__ connectionLimit: __YOUR_CONNECTION_LIMIT__ In the logs of your Docker container (which you can access with docker logs __CONTAINER_ID__), you can then see the the execution time for queries above specified threshold, for example:SLOW QUERY - DURATION: 17 QUERY: [{"query":"{ users { id email name } } ","variables":{}},{"query":"{ users { id email name } } ","variables":{}}] You can find more info in this GitHub issue.Hard resetting the Docker environmentIf your local Prisma server is in an unrecoverable state, it might be necessary to completely reset it:docker-compose kill docker-compose down docker-compose up -d Be careful as these commands will reset all data in your local Prisma server (including deployed Prisma APIs). Prisma CLI & ConfigurationCLI Command ReferenceEdit this page on GithubLast updated 4 days ago Before you start with the implementation, go ahead and start your development server to generate the missing TS definitions:npm run dev QueriespublishedPosts: Fetch all published postspost: Fetch a single post by its idpostsByUser: Fetch all the posts of a certain userMutationscreateUser: Create a new usercreateDraft: Create a new draft (i.e. an unpublished post)publish: Publish a postdeletePost: Delete an existing postThe following GraphQL Nexus code creates a GraphQL schema that implements these requirements. Add it to index.ts:import { prisma } from './generated/prisma-client' import datamodelInfo from './generated/nexus-prisma' import * as path from 'path' import { stringArg, idArg } from 'nexus' import { prismaObjectType, makePrismaSchema } from 'nexus-prisma' import { GraphQLServer } from 'graphql-yoga' const Query = prismaObjectType({ name: 'Query', definition(t) { t.prismaFields(['post']) t.list.field('feed', { type: 'Post', resolve: (_, args, ctx) => ctx.prisma.posts({ where: { published: true } }), }) t.list.field('postsByUser', { type: 'Post', args: { email: stringArg() }, resolve: (_, { email }, ctx) => ctx.prisma.posts({ where: { author: { email } } }), }) }, }) const Mutation = prismaObjectType({ name: 'Mutation', definition(t) { t.prismaFields(['createUser', 'deletePost']) t.field('createDraft', { type: 'Post', args: { title: stringArg(), authorId: idArg({ nullable: true }), }, resolve: (_, { title, authorId }, ctx) => ctx.prisma.createPost({ title, author: { connect: { id: authorId } } }, }), }) t.field('publish', { type: 'Post', nullable: true, args: { id: idArg() }, resolve: (_, { id }, ctx) => ctx.prisma.updatePost({ where: { id }, data: { published: true } }), }) }, }) const schema = makePrismaSchema({ types: [Query, Mutation], prisma: { datamodelInfo, client: prisma, }, outputs: { schema: path.join(__dirname, './generated/schema.graphql'), typegen: path.join(__dirname, './generated/nexus.ts'), }, },)) Page 2Congratulations for finishing the quickstart tutorial and building your first GraphQL server with Prisma! Here is where you can go next. Edit this page on GithubLast updated 4 days agoContentOverviewLimitations If one of more of the provided IDs does not actually exist in the database, the operation will not return an error. The returned count indicates how many records actually were updated. The Prisma client is generated from your datamodel. Its API exposes CRUD and other operations for the models defined in the datamodel.For this page, we'll assume your Prisma project is based on the following datamodel:type Post { id: ID! @id createdAt: DateTime! @createdAt updatedAt: DateTime! @updatedAt title: String! published: Boolean! @default(value: false) author: User comments: [Comment!]! } type User { id: ID! @id name: String email: String! @unique role: Role! @default(value: USER) posts: [Post!]! comments: [Comment!]! } type Comment { id: ID! @id createdAt: DateTime! @createdAt text: String! post: Post! writtenBy: User! } enum Role { USER ADMIN } Whenever a database record is queried using the Prisma client, all scalar fields of that record are fetched. This is true no matter if a single record or a list of records is queried.For example, the following query returns all scalar fields of a single User:email := "ada@prisma.io" user, err := client.User(prisma.UserWhereUniqueInput{ Email: &email, }).Exec(ctx) In this case, the returned user record will have four properties (that correspond to the scalar fields of the User model): id, name, email and role. The posts and comments fields are both relation fields and are therefore not included in the response.Here is an example of fetching a list of User records:users, err := client.Users(nil).Exec(ctx) Similar to the previous request, each object inside the users array only has the scalar and no relation fields.Basic filters let you specify certain criteria to constrain which records should be returned in a list. The filters are specified in the Where struct of the params argument which is accepted by any list query.The type of the Where struct depends on the model for which it was generated.It is also possible to combine multiple filters using the AND and OR fields.ExamplesFetch users that have an A in their names:letter := "A" usersWithAInName, err := client.Users(&prisma.UsersParams{ Where: &prisma.UserWhereInput{ NameContains: &letter, }, }).Exec(ctx) Fetch users called Ada or Grace:names := []string{"Ada", "Grace"} users, err := client.Users(&prisma.UsersParams{ Where: &prisma.UserWhereInput{ NameIn: names, }, }).Exec(ctx) Fetch comments created before December 24, 2019:christmas := "2019-12-24" comments, err := client.Comments(&prisma.CommentsParams{ Where: &prisma.CommentWhereInput{ CreatedAtLt: &christmas, }, }).Exec(ctx) Dates and times in the Prisma client API follow the ISO 8601 standard which generally is of the form: YYYY-MM-DDThh:mm:ss. Learn more.Fetch posts that have prisma or graphql in their title and were created in 2019:filter1 := "prisma" filter2 := "graphql" filteredPosts, err := client.Posts(&prisma.PostsParams{ Where: &prisma.PostWhereInput{ Or: []prisma.PostWhereInput{ prisma.PostWhereInput{ TitleContains: &filter1, }, prisma.PostWhereInput{ TitleContains: &filter2, }, }, }, }).Exec(ctx) When querying a list of records, you can fetch certain parts (i.e. pages) of that list by supplying pagination arguments.Seeking forward and backward with first and lastYou can seek forwards or backwards through the list and supply an optional starting point:To seek forward, use first; specify the id of a starting record with after:To seek backward, use last; specify the id of a starting record with before.You cannot combine first with before or last with after. If you do so, before/after will simply be ignored and only first or last is actually applied (at the very beginning or end of the list, depending on which you're using).Note that you can query for more records than actually exist in the database without an error message.Skipping elements with skipYou can also skip an arbitrary amount of records in whichever direction you are seeking by supplying the skip argument:When using first, skip skips records at the beginning of the list.When using last, skip skips records from the end of the list.For the following examples, we're assuming a list of exactly 30 records:Fetch the first 3 posts (seeking forward):first := int32(3) posts, err := client.Posts(&prisma.PostsParams{ First: &first, }).Exec(ctx) Fetch the posts from position 6 to position 10 (seeking forward):skip := int32(5) first := int32(5) posts, err := client.Posts(&prisma.PostsParams{ First: &first, Skip: &skip, }).Exec(ctx) Fetch the last 3 posts (seeking backward):last := int32(3) posts, err := client.Posts(&prisma.PostsParams{ Last: &last, }).Exec(ctx) Fetch the posts from position 21 to position 27 (seeking backward):skip := int32(3) last := int32(7) posts, err := client.Posts(&prisma.PostsParams{ Last: &last, Skip: &skip, }).Exec(ctx) Fetch the first 3 posts after the posts with cixnen24p33lo0143bexvr52n as id:first := int32(3) after := "cjsyqxwqo000j0982da8cvw7o" posts, err := client.Posts(&prisma.PostsParams{ First: &first, After: &after, }).Exec(ctx) Fetch the first 5 posts after the post with cixnen24p33lo0143bexvr52n as id and skipping 3 posts:first := int32(5) skip := int32(3) after := "cjsyqxwqo000j0982da8cvw7o" posts, err := client.Posts(&prisma.PostsParams{ First: &first, After: &after, Skip: &skip, }).Exec(ctx) Fetch the last 5 posts before the post with cixnen24p33lo0143bexvr52n as id:last := int32(5) before := "cixnen24p33lo0143bexvr52n" posts, err := client.Posts(&prisma.PostsParams{ Last: &last, Before: &before, }).Exec(ctx) Fetch the last 3 posts before the record with cixnen24p33lo0143bexvr52n as id and skipping 5 posts:last := int32(3) before := "cixnen24p33lo0143bexvr52n" skip: := int32(5) posts, err := client.Posts(&prisma.PostsParams{ Last: &last, Before: &before, Skip: &skip, }).Exec(ctx)