Adding Prisma Migrate to an existing project

This guide describes how to add Prisma Migrate to an existing project.

This guide **does not apply for MongoDB**.
Instead of `migrate dev` , `db push` is used for MongoDB.

Overview of the steps

The steps involved in adding Prisma Migrate to your project are:

1. Introspect your database to update your Prisma schema
2. Create a baseline migration
3. Update your schema or migration to workaround features not supported by Prisma Schema Language
4. Apply the baseline migration
5. Commit the migration history and Prisma schema

Introspect to create or update your Prisma schema

Make sure your Prisma schema is in sync with your database schema. This should already be true if you are using a previous version of Prisma Migrate.

1. Introspect the database to make sure that your Prisma schema is up-to-date:

```
$  prisma db pull
```

Create a baseline migration

Baselining is the process of initializing a migration history for a database that:

- ✔ Existed before you started using Prisma Migrate
- ✔ Contains data that must be maintained (like production), which means that the database cannot be reset

Baselining tells Prisma Migrate to assume that one or more migrations have **already been applied**. This prevents generated migrations from failing when they try to create tables and fields that already exist.

To create a baseline migration:

1. If you have a `prisma/migrations` folder, delete, move, rename, or archive this folder.
2. Run the following command to create a `migrations` directory inside with your preferred name. This example will use `0_init` for the migration name:

```
$  mkdir -p prisma/migrations/0_init
```

The `0_` is important because Prisma Migrate applies migrations in a lexicographic order. You can use a different value such as the current timestamp.

3. Generate a migration and save it to a file using `prisma migrate diff`

```
$  npx prisma migrate diff \
$  --from-empty \
$  --to-schema-datamodel prisma/schema.prisma \
$  --script > prisma/migrations/0_init/migration.sql
```

4. Review the generated migration

Work around features not supported by Prisma Schema Language

To include unsupported database features that already exist in the database, you must replace or modify the initial migration SQL:

1. Open the `migration.sql` file generated in the Create a baseline migration section.
2. Modify the generated SQL. For example:
   - If the changes are minor, you can append additional custom SQL to the generated migration - the following example creates a partial index:

```
/* Generated migration SQL */

CREATE UNIQUE INDEX tests_success_constraint ON posts (subject, target)
 WHERE success;
```

   - If the changes are significant, it can be easier to replace the entire migration file with the result of a database dump (mysqldump, pg_dump)

Note that the order of the tables matters when creating all of them at once, since foreign keys are created at the same step. Therefore, either re-order them or move constraint creation to the last step after all tables are created, so you won't face `can't create constraint` errors

Apply the initial migrations

To apply your initial migration(s):

1. Run the following command against your database:

```
$  npx prisma migrate resolve --applied 0_init
```

2. Review the database schema to ensure the migration leads to the desired end-state (for example, by comparing the schema to the production database).

The new migration history and the database schema should now be in sync with your Prisma schema.

Commit the migration history and Prisma schema

Commit the following to source control:

- The entire migration history folder
- The `schema.prisma` file

Going further

- Refer to the Deploying database changes with Prisma Migrate guide for more on deploying migrations to production.
- Refer to the Production Troubleshooting guide to learn how to debug and resolve failed migrations in production using `prisma migrate diff`, `prisma db execute` and/ or `prisma migrate resolve`.