Mapped types

> warning **Warning** This chapter applies only to the code first approach.

As you build out features like CRUD (Create/Read/Update/Delete) it's often useful to construct variants on a base entity type. Nest provides several utility functions that perform type transformations to make this task more convenient.

Partial

When building input validation types (also called Data Transfer Objects or DTOs), it's often useful to build **create** and **update** variations on the same type. For example, the **create** variant may require all fields, while the **update** variant may make all fields optional.

Nest provides the `PartialType()` utility function to make this task easier and minimize boilerplate.

The `PartialType()` function returns a type (class) with all the properties of the input type set to optional. For example, suppose we have a **create** type as follows:

```
@InputType()
class CreateUserInput {
  @Field()
  email: string;

  @Field()
  password: string;

  @Field()
  firstName: string;
}
```

By default, all of these fields are required. To create a type with the same fields, but with each one optional, use `PartialType()` passing the class reference (`CreateUserInput`) as an argument:

```
@InputType()
export class UpdateUserInput extends PartialType(CreateUserInput) {}
```

> info **Hint** The `PartialType()` function is imported from the `@nestjs/graphql` package.

The `PartialType()` function takes an optional second argument that is a reference to a decorator factory. This argument can be used to change the decorator function applied to the resulting (child) class. If not specified, the child class effectively uses the same decorator as the **parent** class (the class referenced in the first argument). In the example above, we are extending `CreateUserInput` which is annotated with the `@InputType()` decorator. Since we want `UpdateUserInput` to also be treated as if it were decorated with `@InputType()`, we didn't need to pass `InputType` as the second argument. If the parent and child types are different, (e.g., the parent is decorated with `@ObjectType`), we would pass `InputType` as the second argument. For example:

```
@InputType()
export class UpdateUserInput extends PartialType(User, InputType) {}
```

Pick

The `PickType()` function constructs a new type (class) by picking a set of properties from an input type. For example, suppose we start with a type like:

```
@InputType()
class CreateUserInput {
  @Field()
  email: string;

  @Field()
  password: string;

  @Field()
  firstName: string;
}
```

We can pick a set of properties from this class using the `PickType()` utility function:

```
@InputType()
export class UpdateEmailInput extends PickType(CreateUserInput, [
  'email',
] as const) {}
```

> info **Hint** The `PickType()` function is imported from the `@nestjs/graphql` package.

Omit

The `OmitType()` function constructs a type by picking all properties from an input type and then removing a particular set of keys. For example, suppose we start with a type like:

```
@InputType()
class CreateUserInput {
  @Field()
  email: string;

  @Field()
  password: string;
  @Field()
  firstName: string;
}
```

We can generate a derived type that has every property **except** `email` as shown below. In this construct, the second argument to `OmitType` is an array of property names.

```
@InputType()
export class UpdateUserInput extends OmitType(CreateUserInput, [
  'email',
] as const) {}
```

> info **Hint** The `OmitType()` function is imported from the `@nestjs/graphql` package.

Intersection

The `IntersectionType()` function combines two types into one new type (class). For example, suppose we start with two types like:

```
@InputType()
class CreateUserInput {
  @Field()
  email: string;

  @Field()
  password: string;
}

@ObjectType()
export class AdditionalUserInfo {
  @Field()
  firstName: string;

  @Field()
  lastName: string;
}
```

We can generate a new type that combines all properties in both types.

```
@InputType()
export class UpdateUserInput extends IntersectionType(
  CreateUserInput,
  AdditionalUserInfo,
) {}
```

> info **Hint** The `IntersectionType()` function is imported from the `@nestjs/graphql` package.

Composition

The type mapping utility functions are composable. For example, the following will produce a type (class) that has all of the properties of the `CreateUserInput` type except for `email`, and those properties will be set to optional:

```
@InputType()
export class UpdateUserInput extends PartialType(
  OmitType(CreateUserInput, ['email'] as const),
) {}
```