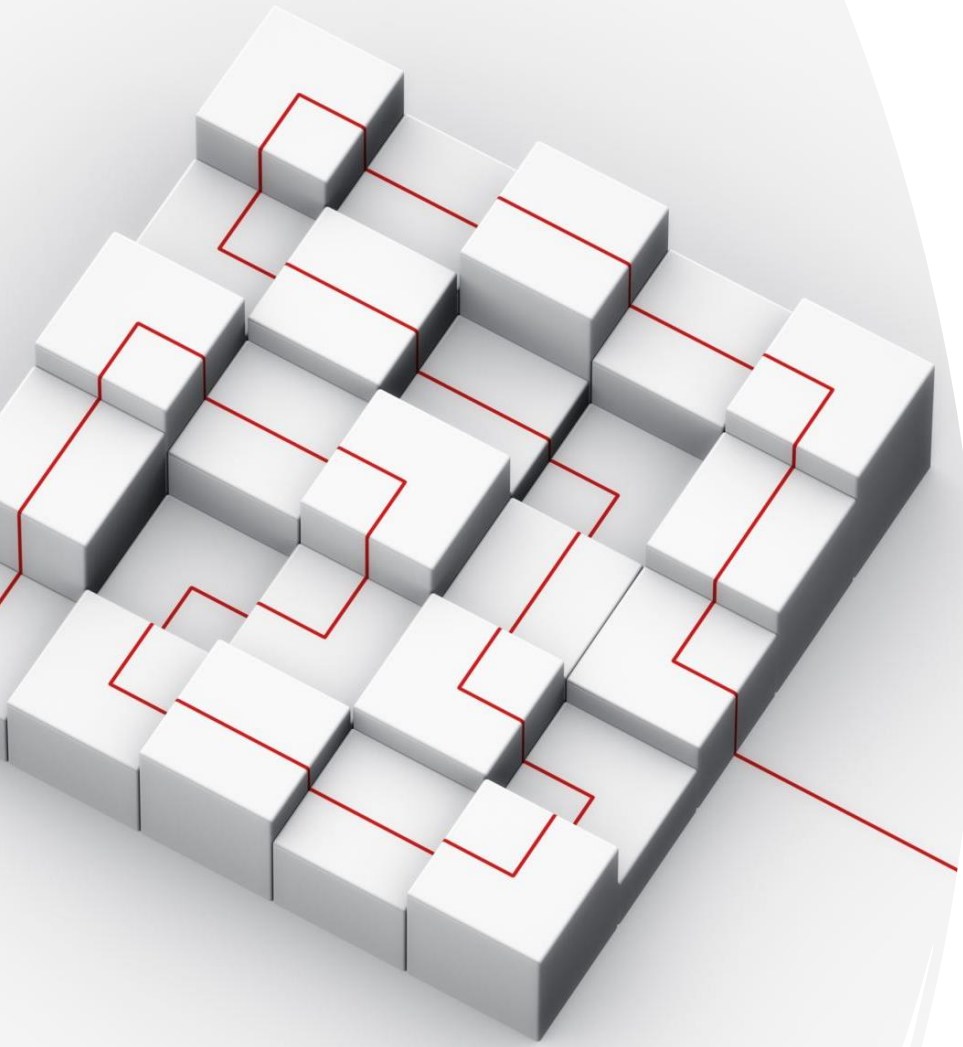
INT 161

Basic Backend Development

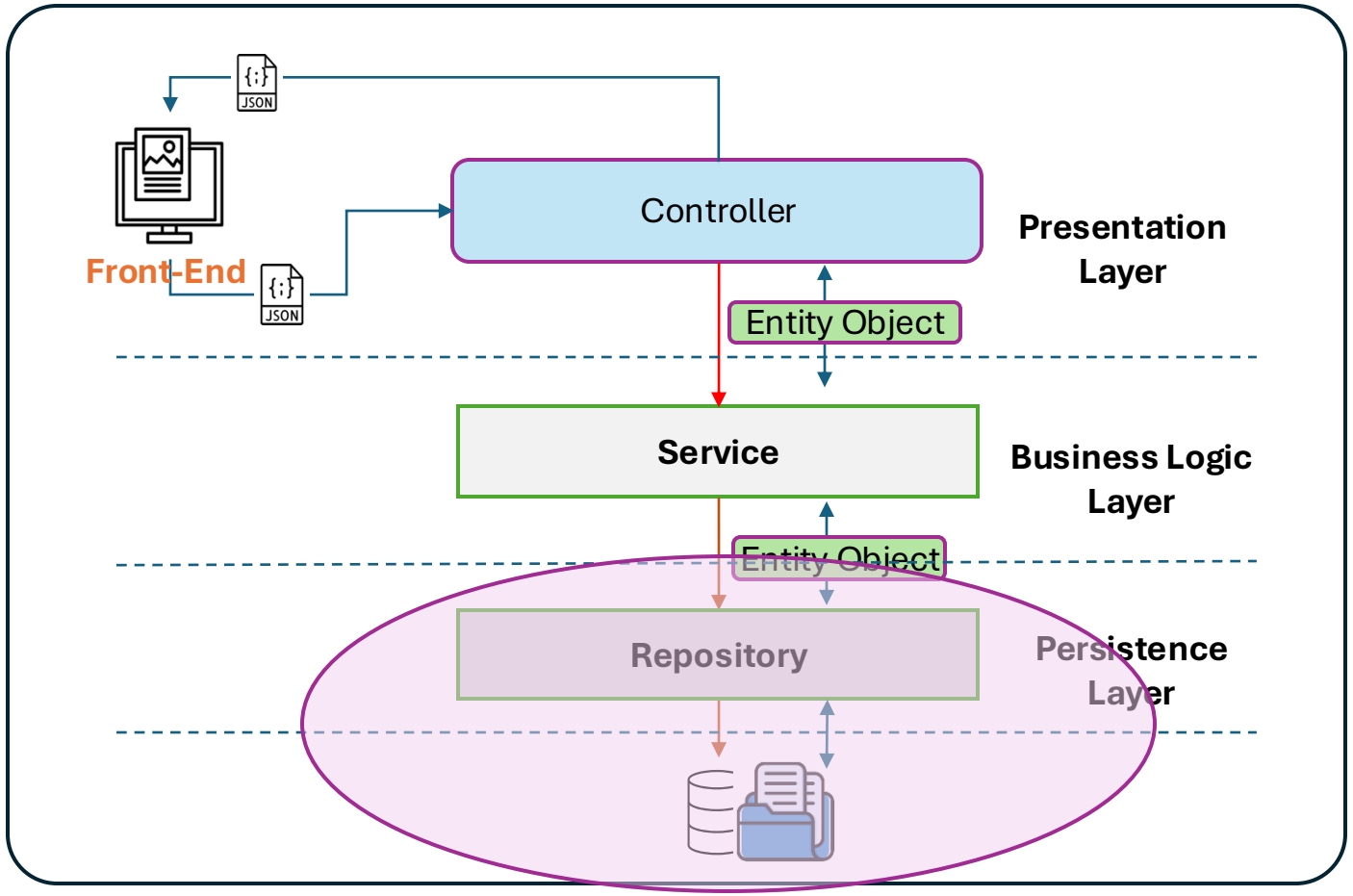
DATA MODELING



Unit Objectives

- After completing this unit, you should be able to:
  - Understand basic concept of Prisma
  - Explain step to using Prisma
  - Create Prisma Schema Data Model
  - Create REST API CRUD with Prisma

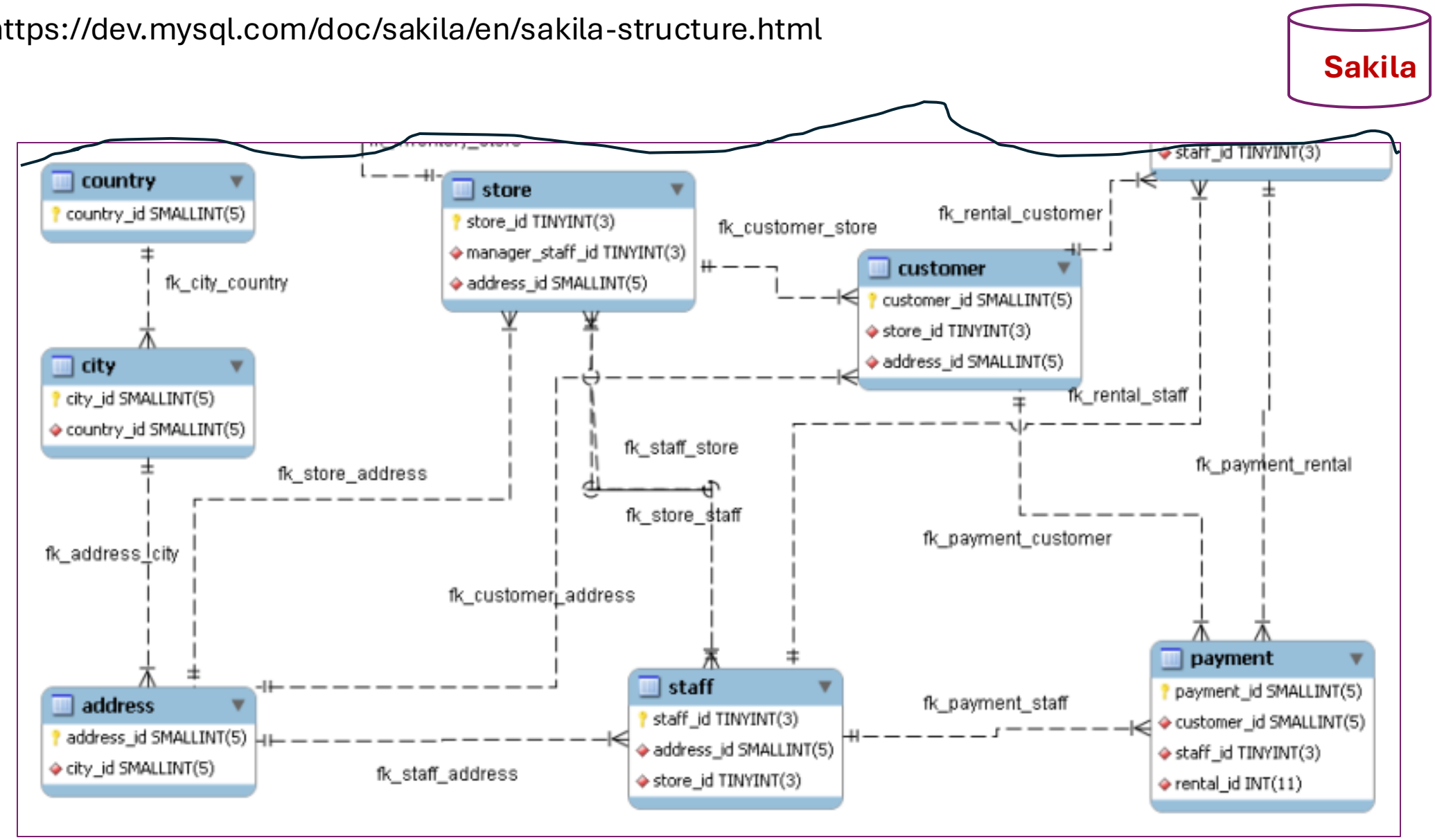
Layered System



The **Layered System** principle means that a REST API is designed as a set of layers, where each layer has a specific role, and a client does not need to know whether it's communicating directly with the end server or through intermediaries.

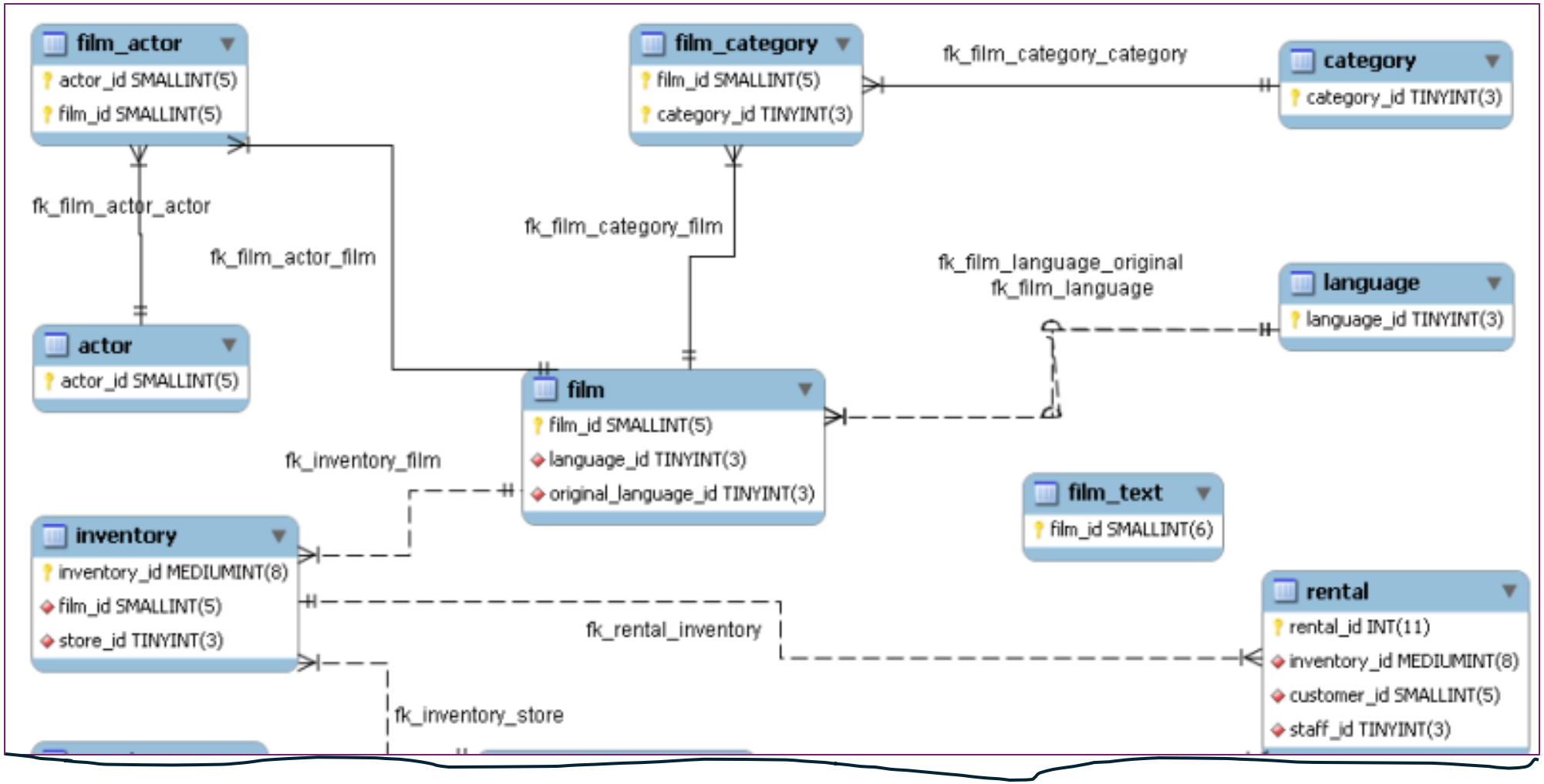
This allows **scalability, flexibility, and separation of concerns**.

https://dev.mysql.com/doc/sakila/en/sakila-structure.html



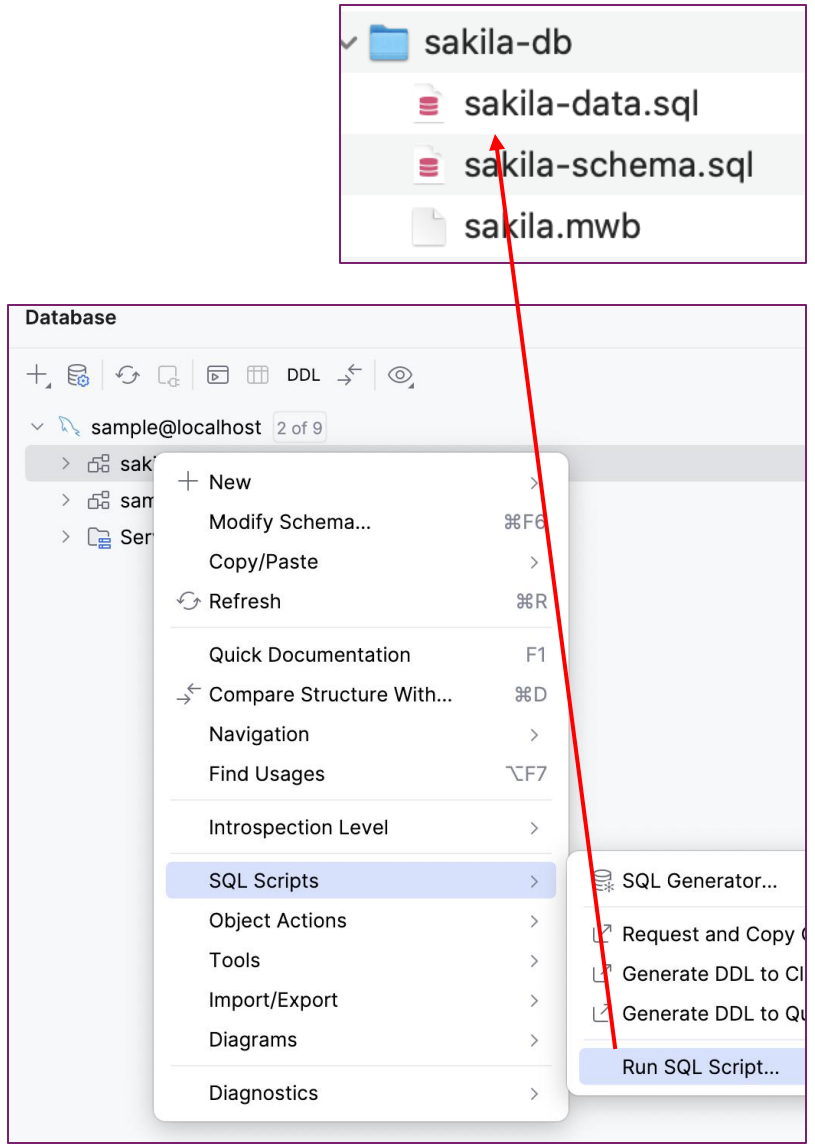
https://dev.mysql.com/doc/sakila/en/sakila-structure.html

Sakila



Database Setup

- Download sakila-db.zip from class materials then extract it
- Create schema: sakila
- Run sql script : sakila-schema.sql
- Run sql script : sakila-data.sql



.env

```
DATABASE_URL="mysql://root:143900@localhost:3306/sakila"
```

Prisma CRUD summary - Create

Method	Description	Example
create()	Creates a single record.	await prisma.user.create({ data: {name: 'Alice', email: 'alice@prisma.io' }, });

createMany()	Creates multiple records in one transaction.	await prisma.user.createMany({
		data: [{ name: 'Bob', email: 'bob@prisma.io', },
		{ name: 'Charlie', email: 'charlie@prisma.io' }
		],
		skipDuplicates: true,
		// Ignores duplicates if any, instead of failing.
		});

Prisma CRUD summary - Read

Method	Description	Example
findUnique()	Fetches a single, unique record by a unique identifier, like id or email.	await prisma.user.findUnique({ where: { id: 1, } });
findFirst()	Fetches the first record matching the search criteria.	await prisma.user.findFirst({ where: { name: 'Alice', } });

findMany()	Fetches all records matching the search criteria.	// Find all users
		await prisma.user.findMany();
		// Find users with a specific name await
		prisma.user.findMany({
		where: { name: 'Alice', }
		});

Prisma CRUD summary - Update

Method	Description	Example
update()	Updates a single, unique record.	await prisma.user.update({ where: { id: 1, }, data: { name: 'Alice Updated', } });
updateMany()	Updates multiple records matching a search criteria.	await prisma.user.updateMany({ where: { name: 'Alice', }, data: { name: 'Alice Renamed', } });
upsert()	Upsert (Update or Insert) creates a record if it doesn't exist, and updates it if it does.	await prisma.user.upsert({ where: { email: 'test@prisma.io', }, update: { name: 'Tester', }, create: { name: 'Tester', email: 'test@prisma.io' }, });

Prisma CRUD summary - Delete

Method	Description	Example
delete()	Deletes a single, unique record.	await prisma.user.delete({ where: { id: 1, }, });

deleteMany()	Deletes multiple records that match a search criteria.	await prisma.user.deleteMany({
		where: {
		name: {
		contains: 'Test' }
		}
		});

REST API (also known as RESTful API)

- REST stands for **RE**presentational **S**tate **T**ransfer and was created by computer scientist Roy Fielding.
- An application programming interface (API or web API) that conforms to the constraints of REST architectural style and allows for interaction with RESTful web services.
- In REST architecture, a REST Server simply provides access to resources and REST client accesses and modifies the **resources**.
- Each **resource** is **identified** by URIs/ global IDs.
- REST uses various representation to represent a resource like text, JSON, XML. JSON is the most popular one.

Core Concepts of REST

- **Statelessness:** No client context is stored on the server.
- **Resource Representation:** Data is represented in formats like JSON or XML.
- **Uniform Interface:** Standardized HTTP methods.
- **Layered System:** Architecture is modular and layered.

HTTP Methods

- **Method Purpose Example**
  - GET Retrieve data /users
  - POST Create new resource /users (with data)
  - PUT Update existing data /users/{id} (with data)
  - DELETE Remove a resource /users/{id}

Rules of REST API

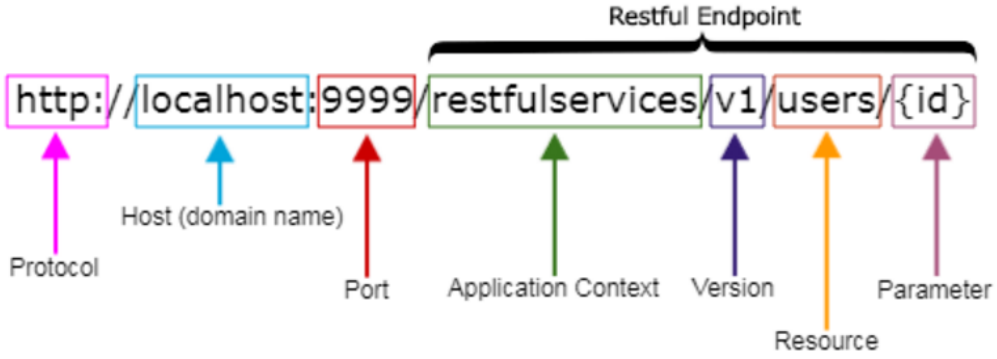
- There are certain rules which should be kept in mind while creating REST **API endpoints**.
  - REST is based on the resource or **noun** instead of action or verb based. It means that a URI of a REST API should always end with a noun. Example: **/api/users** is a good example.
  - HTTP verbs are used to identify the action. Some of the HTTP verbs are – GET, PUT, POST, DELETE, PATCH.
  - A web application should be organized into resources like users and then uses HTTP verbs like – GET, PUT, POST, DELETE to modify those resources. And as a developer it should be clear that what needs to be done just by looking at the endpoint and HTTP method used.

RESTful Resource Uniform Interface

URI	HTTP Verb	Description
api/offices	GET	Get all office
api/offices/1	GET	Get an office with id = 1
api/offices/1/employees	GET	Get all employee for office id = 1
api/offices/	POST	Add new office
api/offices/1/employees	POST	Add new employee to office id =1
api/offices/1	PUT	Update an office with id = 1
api/offices/1	DELETE	Delete an office with id = 1

Always use plurals in URL to keep an API URI consistent throughout the application.
Send a proper HTTP code to indicate a success or error status.

REST API URI Naming Conventions and Best Practices



- Singleton and Collection Resources

```
/customers                  // is a collection resource
/customers/{id}             // is a singleton resource
```

- Sub-collection Resources

```
/customers/{id}/orders                  // is a sub-collection resource
```

- Best Practices
  - https://medium.com/@nadinCodeHat/rest-api-naming-conventions-and-best-practices-1c4e781eb6a5
  - https://restfulapi.net/resource-naming

Example RESTful Resource for Sakila

URI	HTTP Verb	Description
/customers	GET	Get all customer
/customers/{id}	GET	Get a customer with specific id
/customers/{id}/addresses	GET	Get an address of customer id = ?
/customers/	POST	Add new customer
/customers/{id}	PUT	Update a customer with specific id
/customers/{id}	DELETE	Delete a customer with specific id

Example RESTful Resource for Sakila

URI	HTTP Verb	Description
/countries	GET	Get all country
/countries/{id}	GET	Get a country with specific id
/countries/{id}/cities	GET	Get all city of country id = ?
/countries	POST	Add new country
/countries/{id}/cities	POST	Add new city to country id = ?
/countries/{id}	PUT	Update a country with specific id
/countries/{id}	DELETE	Delete a country with specific id

Example RESTful Resource for Sakila

URI	HTTP Verb	Description
/cities	GET	Get all city
/cities/{id}	GET	Get a city with specific id
/cities/{id}/addresses	GET	Get all address in city id = ?
/cities	POST	Add new city
/cities/{id}/addresses	POST	Add address to city id = ?
/cities/{id}	PUT	Update a city with specific id
/cities/{id}	DELETE	Delete a city with specific id

Prisma Data Modelling (1:m, m:1 relationship)

```
model City {
  id          Int       @id @default(autoincrement()) @db.UnsignedSmallInt @map("city_id")
  city        String    @db.VarChar(50)
  countryId   Int       @map("country_id") @db.UnsignedSmallInt
  lastUpdate  DateTime  @default(now()) @db.Timestamp(0) @map("last_update")
  addresses   Address[]
  country     Country   @relation(fields: [countryId], references: [id]
                       , map: "fk_city_country")

  @@index([id], map: "idx_fk_country_id")
  @@map("city")
}

model Country {
  id          Int      @id @default(autoincrement()) @db.UnsignedSmallInt @map("country_id")
  country     String   @db.VarChar(50)
  lastUpdate  DateTime @default(now()) @map("last_update") @db.Timestamp(0)
  cities      City[]

  @@map("country")
}
```

Prisma Data Modelling (1:m, m:1 relationship)

```
model Address {
  id          Int        @id @default(autoincrement()) @map("address_id") @db.UnsignedSmallInt
  address     String     @db.VarChar(50)
  address2    String?    @db.VarChar(50)
  district    String     @db.VarChar(20)
  cityId      Int        @map("city_id") @db.UnsignedSmallInt
  postalCode  String?    @db.VarChar(10) @map("postal_code")
  phone       String     @db.VarChar(20)
  lastUpdate  DateTime   @default(now()) @map("last_update") @db.Timestamp(0)
  city        City       @relation(fields: [cityId], references: [id], map: "fk_address_city")
  customers   Customer[]

  @@index([id], map: "idx_fk_city_id")
  @@map("address")
}
```

Prisma Data Modelling (m:1 relationship)

```
model Customer {
  id          Int       @id @default(autoincrement()) @map("customer_id")
                       @db.UnsignedSmallInt
  firstName   String    @map("first_name") @db.VarChar(45)
  lastName    String    @map("last_name") @db.VarChar(45)
  email       String?   @db.VarChar(50)
  addressId   Int       @map("address_id") @db.UnsignedSmallInt
  active      Boolean   @default(true)
  createDate  DateTime  @map("create_date") @db.DateTime(0)
  lastUpdate  DateTime? @default(now()) @map("last_update") @db.Timestamp(0)
  address     Address   @relation(fields: [addressId], references: [id],
                       map: "fk_customer_address")

  @@index([id], map: "idx_fk_address_id")
  @@index([lastName], map: "idx_last_name")
  @@map("customer")
}
```

Retrieved nested relations with `include`

```
const result = await prisma.model.findMany ({
  include: {
     relationA: true,        // include all fields
     relationB: {
          select: {            // include selected field(s)
              field1: true,
              field2:true
           }
     },
     relationC: {              // all fields nested include
        include: {
            subRelation: {
                 include: {
                      deepRelation: true
                 }
            }
        }
     }
  }
});
```

`include` Example (1)

```
const { PrismaClient } =
require("../generated/prisma/sakila");
const prisma = new PrismaClient();

module.exports = {
    findAll: async  function () {
        return await prisma.country.findMany();
    },
    findById: async function (id, includeCity = false) {
        return await prisma.country.findUnique({
            where: { id: id },
            include: {
                cities: includeCity
            }
        });
    },
```

`include` Example (2)

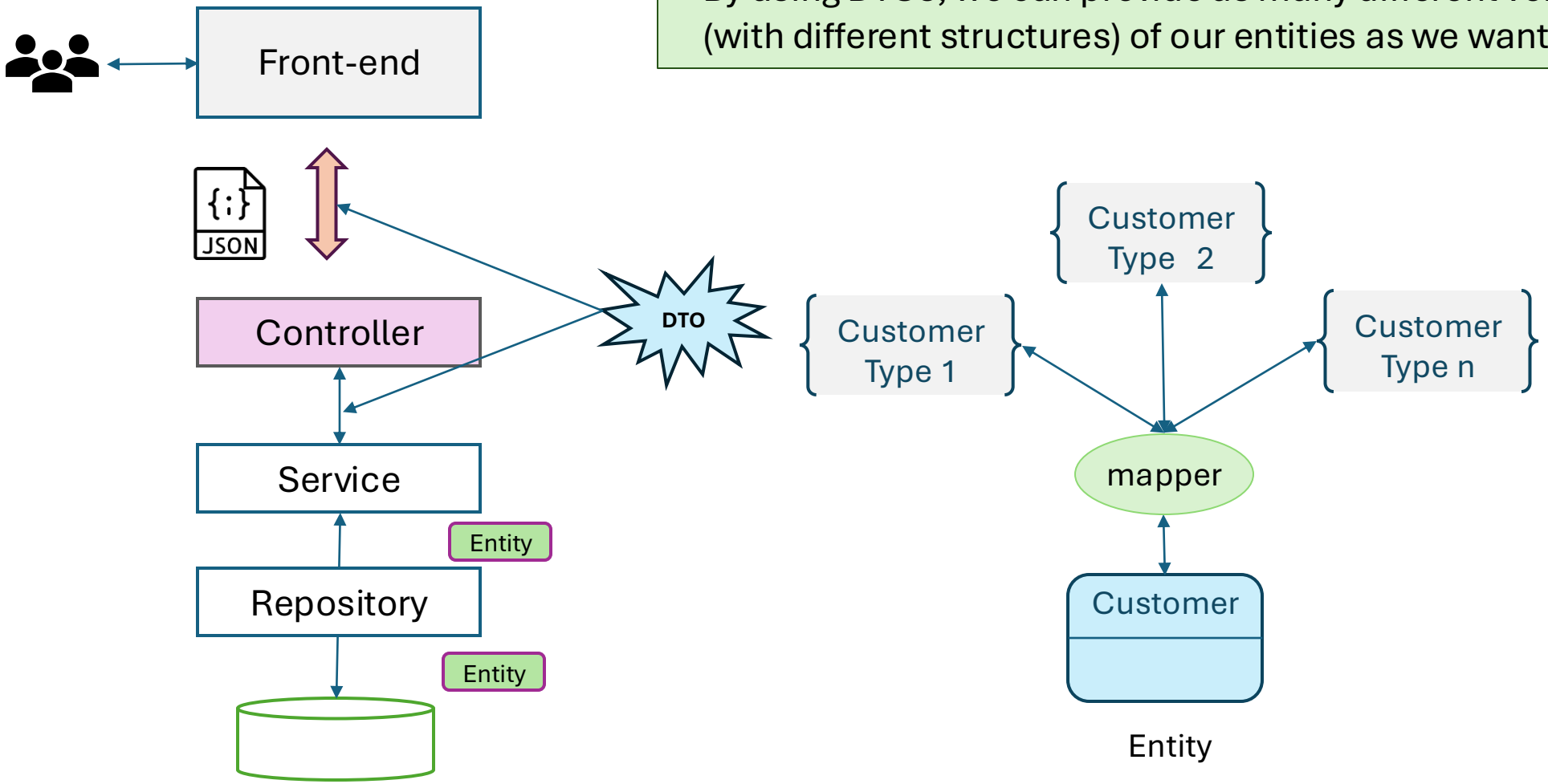
```
const { PrismaClient } = require("../generated/prisma/sakila");
const prisma = new PrismaClient();

module.exports = {
    findById: async function (id, includeCity = false) {
        return await prisma.country.findUnique({
            where: { id: id },
            include: includeCity ? {
                cities: {
                    select: {
                        id: true,
                        city: true,
                    }
                }
            } : false,
        });
    },
```

`include` Example (3)

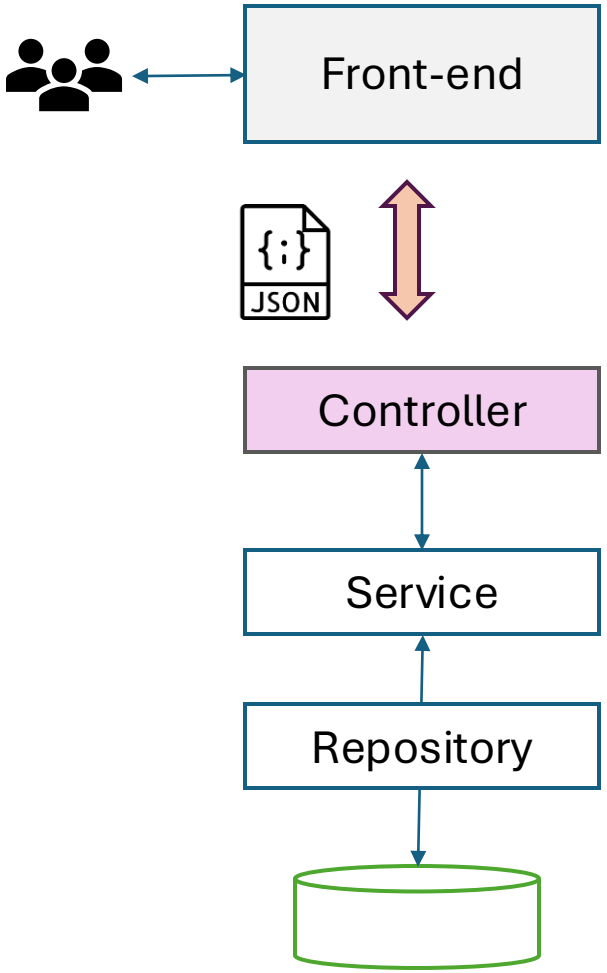
```
const {PrismaClient} = require("../generated/prisma/sakila");
const prisma = new PrismaClient();
module.exports = {
findById: async function (id) {
    return await prisma.customer.findUnique({
        where: {id: id},
        include: {
            address: {
                include: {
                    city: {
                        include: {
                            country: true
                        }
                    },
                }
            }
        }
    });
},
```

DTO: Data Transfer Object

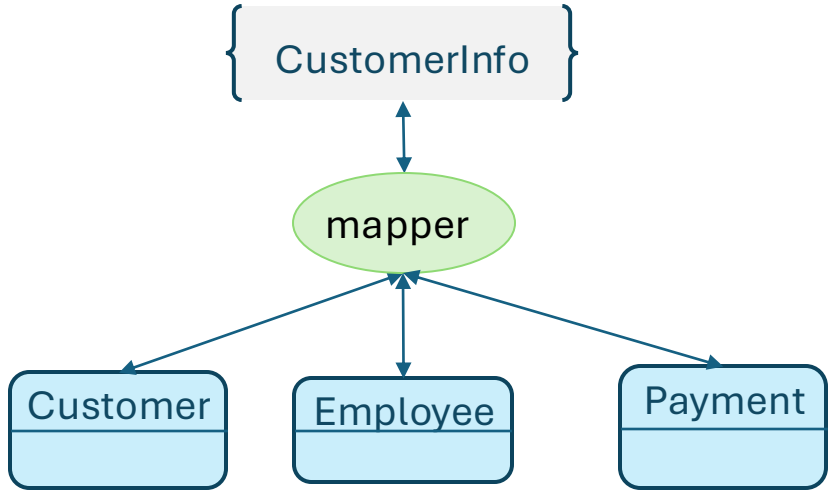
- The data is mapped from the domain models to the DTOs.
- By using DTOs, we can provide as many different versions (with different structures) of our entities as we want.



DTO: **D**ata **T**ransfer **O**bject



DTO is a design pattern conceived to reduce the number of calls when working with remote interfaces.



Another advantage of using DTOs on RESTful APIs is that they can help hiding implementation details of domain objects (aka. entities). Exposing entities through endpoints can become a security issue if we do not carefully handle what properties can be changed through what operations.

DTO: Example

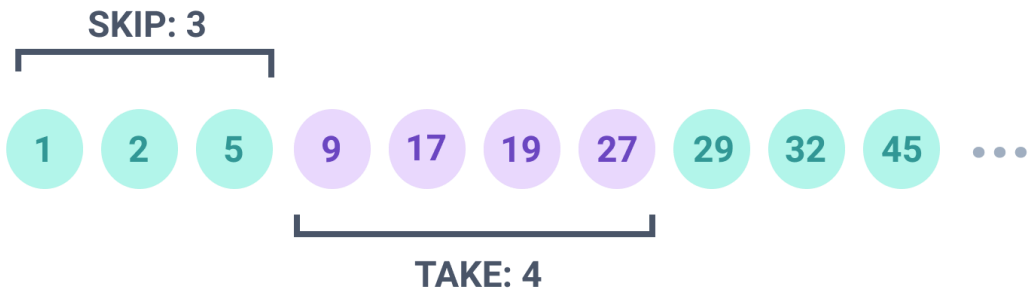
```
const uniqueOne = await service.getById(id);
res.json(new SimpleCustomerDto(uniqueOne));
```

```
class SimpleCustomerDto {
  constructor(customer = {}) {
    const { id, firstName, lastName, email, address, active } = customer;
    this.id = id ?? null;
    this.name = [firstName, lastName].filter(Boolean).join(" ") || null;
    this.email = email ?? null;
    this.active = active ?? false;
    if (address) {
      this.address = {
        id: address.id ? address.id : null,
        address: address.address ? `${address.address} ${address.address2}`.trim() : null,
        district: address.district ? address.district : '-',
        city: address.city && address.city.city ? address.city.city : null,
        postalCode: address.postalCode ? address.postalCode : '-',
        country: address.city && address.city.country ? address.city.country.country : null,
        phone: address.phone ? address.phone : '-',
      }
    }
  }
}
module.exports = SimpleCustomerDto;
```

Prisma Pagination

- Prisma Client supports both offset pagination and cursor-based pagination.
- Offset pagination
  - Offset pagination uses skip and take to skip a certain number of results and select a limited range.
  - The following query skips the first 3 Customer records and returns records 4 - 7:

```
const results = await prisma.customer.findMany({
  skip: 3,
  take: 4,
})
```



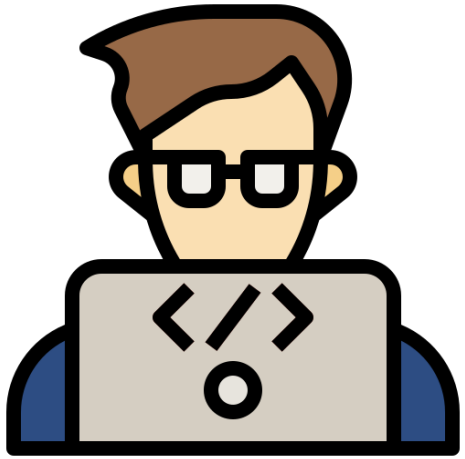
- To implement pages of results, you would just skip the number of pages multiplied by the number of results you show per page.

Offset pagination Example

```
const data = await
prisma.customer.findMany({
   skip: (page - 1) * pageSize,
   take: pageSize,
   orderBy: sortBy,
   include: {
      address: includeAddress ? {
         include: {
               city: {
                     include: {
                        country: true
                     }
               },
         }
      } : false,
   },
```

page	pageSize	skip	take
1	10	0	1-10
2	10	10	11-20
3	10	20	21-30
9	10	80	81-90

Practices



Country Repository: `country-repository.js`

```
const { PrismaClient } = require("../generated/prisma");
const prisma = new PrismaClient();

module.exports = {
    findAll: async  function () {
        return await prisma.country.findMany();
    },
    findById: async function (id, includeCity = false) {
        return await prisma.country.findUnique({
            where: { id: id },
            include: includeCity ? {
                cities: {
                    select: {
                        id: true,
                        city: true,
                    }
                }
            } : false,
        });
    },
}
```

Country Service: `country-service.js`

```
const repo = require('../repositories/country-repository');

module.exports = {
    getAll: async function () {
        const array = await repo.findAll();
        return array;
    },
    getById: async function (id, includeCity = false) {
        const uniqueOne = await repo.findById(id, includeCity);
        if (!uniqueOne) {
            const err = new Error(`Country not found for ID ${id}`);
            err.code = 'NOT_FOUND';
            err.status = 404;
            throw err;
        }
        return uniqueOne;
    },
}
```

Country Controller: `country-controller.js`

```
var service = require('../services/country-service');

module.exports = {
   list: async function (req, res) {
      try {
         const countries = await service.getAll();
         res.json(countries);
      } catch (e) {
         res.status(e.status).json(
            {code:e.code, mesage:e.message,
              status:e.status}
         );
      }
   },
```

```
   get: async function (req, res) {
      const id = Number(req.params.id);
      try {
         const {includeCity} = req.query;
         const country = await service.getById(id,includeCity);
         res.json(country);
      } catch (e) {
         res.status(e.status).json(
            {code:e.code, mesage:e.message, status:e.status}
         );
      }
   },
```

Country Router: `country-route.js`

```
var express = require('express');
var router = express.Router();
const controller = require('../controllers/country-controller');

router.get('/', controller.list);
router.get('/:id', controller.get);

module.exports = router;
```

** Add Coutry Router to App.js ** (map to '/countries')

Customer Repository(1/2): `customer-repository.js`

```
const {PrismaClient} = require("../generated/prisma");
const prisma = new PrismaClient();
module.exports = {
   findAll: async function (includeAddress = false, {page = 1, pageSize = 10}) {
      const totalItems = await prisma.customer.count();
      const data = await prisma.customer.findMany( {
            skip: (page - 1) * pageSize,
            take: pageSize,
            include: {
               address: includeAddress ? {
                  include: {
                     city: {
                        include: { country: true }
                     },
                  }
               } : false,
            },
      });
```

```
      return {
         data: data,
         pageInfo: {
            page: page,
            pageSize: pageSize,
            totalItems: totalItems,
            totalPages: Math.ceil(totalItems / pageSize),
         }
      }
   },
```

Customer Repository(2/2): customer-repository.js

```
findById: async function (cid) {
    return await prisma.customer.findUnique({
        where: {id: cid},
        include: {
            address: {
                include: {
                    city: {
                        include: {
                            country: true
                        }
                    },
                }
            }
        }
    });
},
```

Customer DTO: `simple-customer-dto.js`

```
class SimpleCustomerDto {
  constructor(customer = {}) {
    const { id, firstName, lastName, email, address, active } = customer;
    this.id = id ?? null;
    this.name = [firstName, lastName].filter(Boolean).join(" ") || null;
    this.email = email ?? null;
    this.active = active ?? false;
    if (address) {
      this.address = {
        id: address.id ? address.id : null,
        address: address.address ? `${address.address} ${address.address2}`.trim() : null,
        district: address.district ? address.district : '-',
        city: address.city && address.city.city ? address.city.city : null,
        postalCode: address.postalCode ? address.postalCode : '-',
        country: address.city && address.city.country ? address.city.country.country : null,
        phone: address.phone ? address.phone : '-',
      }
    }
  }
}
module.exports = SimpleCustomerDto;
```

Customer Service : `customer-service.js`

```
const repo = require('../repositories/customer-repository');
module.exports = {
   getAll: async function (includeAddress = false, pageRequest = {}) {
      const array = await repo.findAll(includeAddress, pageRequest);
      return array;
   },
   getById: async function (id) {
      const uniqueOne = await repo.findById(id);
      if (!uniqueOne) {
         const err = new Error(`City not found for ID ${id}`);
         err.code = 'NOT_FOUND';
         err.status = 404;
         throw err;
      }
      return uniqueOne;
   },
}
```

Customer Controller(1/2): `customer-controller.js`

```
const service = require('../services/customer-service');
const SimpleCustomerDto = require('../dtos/simple-customer-dto');
module.exports = {
   list: async function (req, res) {
      try {
         const includeAddress = req.query.includeAddress || false;
         const {page,pageSize} = req.query;
         pageRequest = {
            page: Number(page) || 1,
            pageSize: Number(pageSize) || 10
         };
         const pageCustomer = await service.getAll(includeAddress, pageRequest);
         simpleCustomers = pageCustomer.data.map(customer => new SimpleCustomerDto(customer));
         pageCustomer.data = simpleCustomers;
         res.json(pageCustomer);
      } catch (e) {
          res.status(e.status).json({code:e.code, mesage:e.message, status:e.status});
      }
   },
```

Customer Controller(2/2): customer-controller.js

```
get: async function (req, res) {
    const id = Number(req.params.id);
    try {
        const uniqueOne = await service.getById(id);
        res.json(new SimpleCustomerDto(uniqueOne));
    } catch (e) {
        res.status(e.status).json(
           {code:e.code, mesage:e.message, status:e.status});
    }
},
```

Customer Router : customer-route.js

```
var express = require('express');
var router = express.Router();
const controller = require('../controllers/customer-
controller');

router.get('/', controller.list);
router.get('/:id', controller.get);



module.exports = router;
```

** Add Customer Router to App.js ** (map to **'/customers'**)