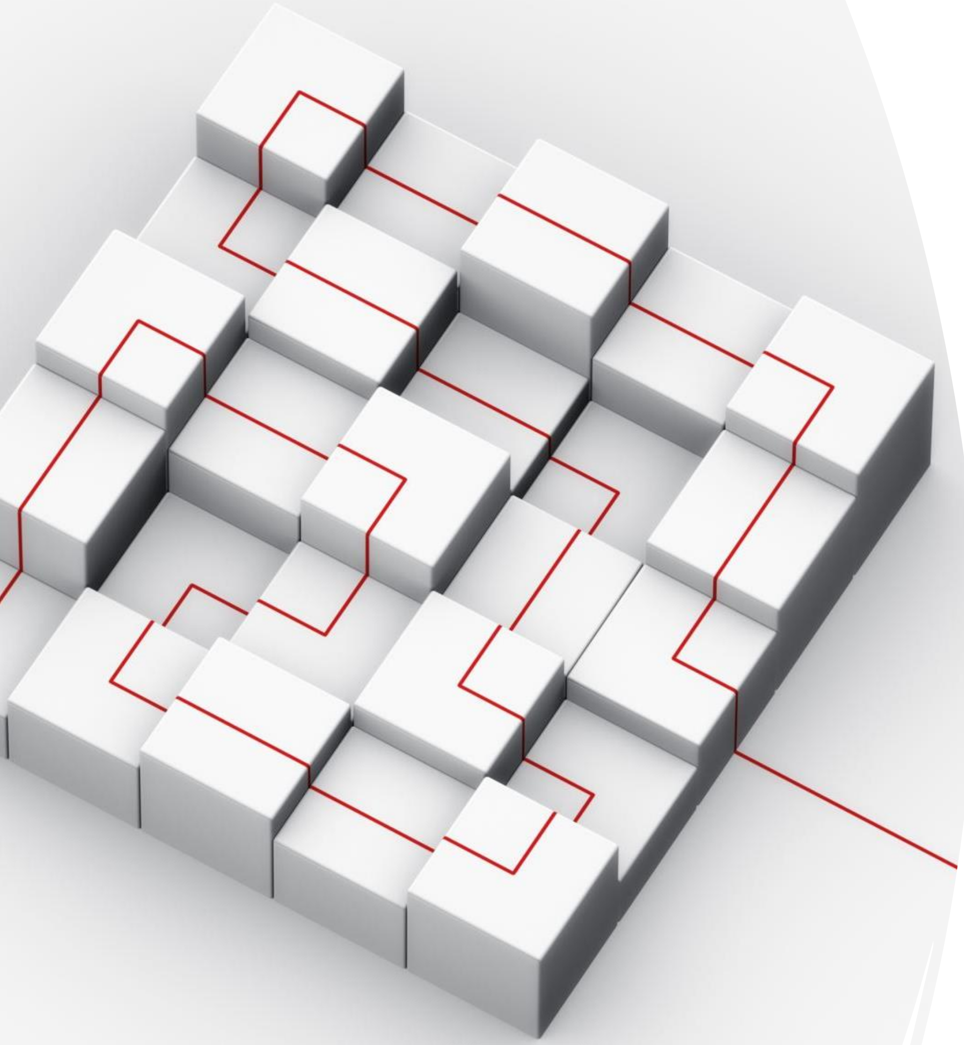
INT 161

Prisma

Basic Backend Development

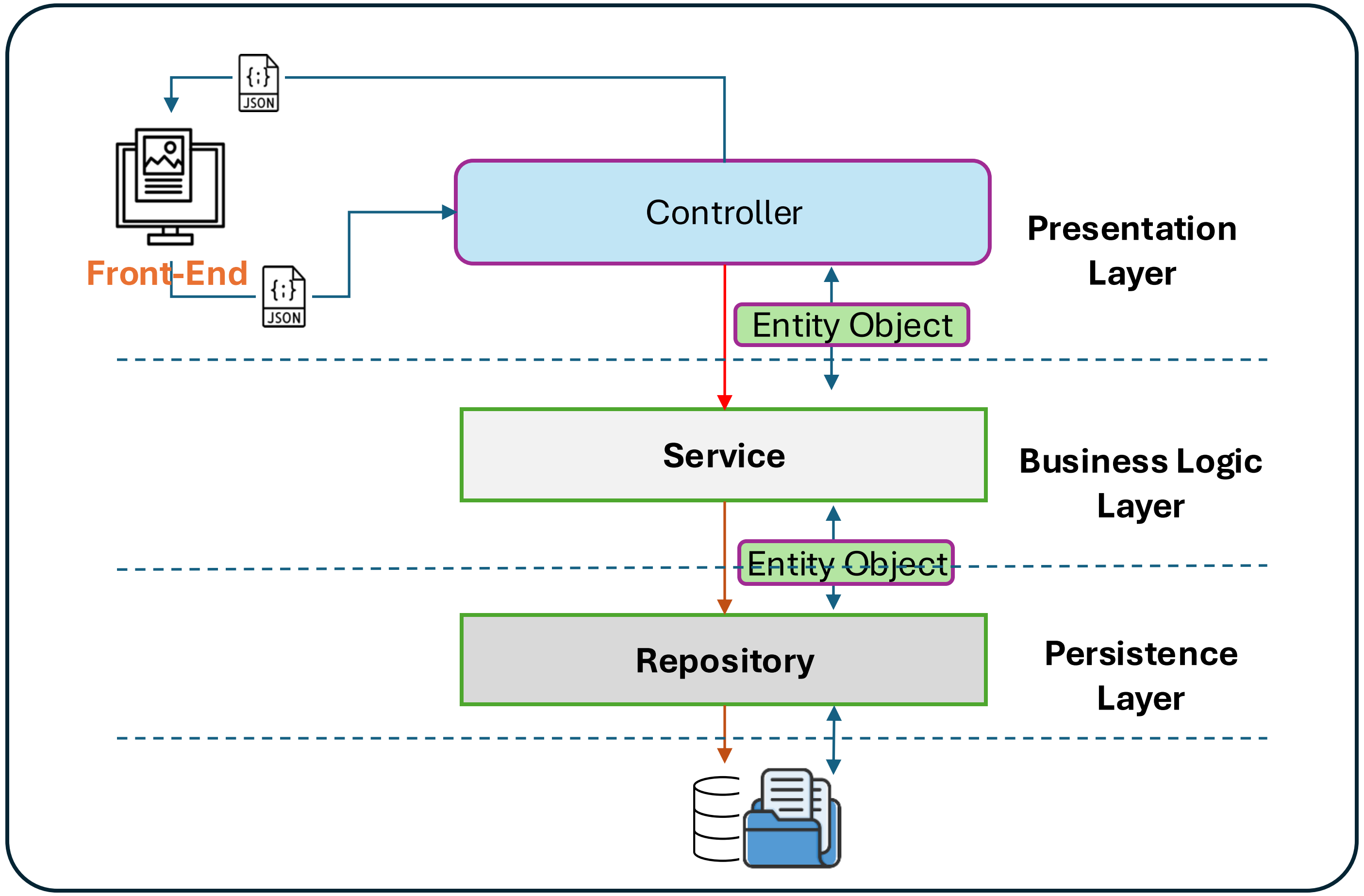
CREATE CRUD REST API with Prisma



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**.

Introduction to Prisma ORM

- Prisma ORM is an open-source next-generation ORM.
  - Prisma Client:
    - Auto-generated and type-safe query builder for Node.js & TypeScript
    - Prisma Client can be used in any Node.js (supported versions) or TypeScript backend application.
    - This can be a REST API, a GraphQL API, a gRPC API, or anything else that needs a database.
  - Prisma Migrate: Migration system
    - Prisma ORM's integrated database migration tool
  - Prisma Studio: GUI to view and edit data in your database.

https://www.prisma.io/docs/getting-started

How does Prisma ORM work?

.env

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

- The Prisma schema
  - Every project that uses a tool from the Prisma ORM toolkit starts with a Prisma schema.
  - The Prisma schema allows developers to define their application models in an intuitive data modeling language.
  - It also contains the connection to a database and defines a generator:

```
datasource db {
provider = ”mysql"
  url = env("DATABASE_URL")
  output = "../generated/prisma"
}

generator client {
  provider = "prisma-client-js"
}
```

```
model User {
    id Int @id @default(autoincrement())
    email String @unique
    name String?
    posts Post[]
}
```

```
model Post {
        id Int @id @default(autoincrement())
        title String
        content String?
        published Boolean @default(false)
        author User? @relation(fields: [authorId], references: [id])
        authorId Int?
}
```

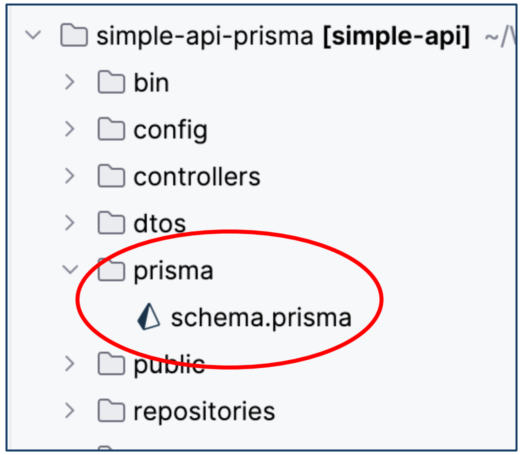
The Prisma schema data model

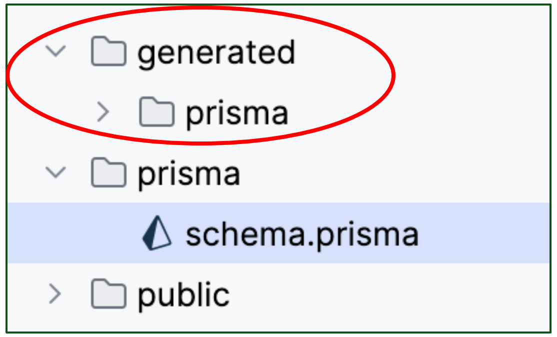
- The data model is a collection of models.
- A model has two major functions:
    - Represent a table in relational databases or a collection in MongoDB
    - Provide the foundation for the queries in the Prisma Client API
- Getting a data model
    - There are two major workflows for "getting" a data model into your Prisma schema:
        - Manually writing the data model and mapping it to the database with Prisma Migrate
        - Generating the data model by introspecting a database
    - Once the data model is defined, you can generate Prisma Client which will expose CRUD and more queries for the defined models.

Accessing your database with Prisma Client

- Generating Prisma Client
  - The first step when using Prisma Client is installing the @prisma/client and prisma npm packages:

```
> npm install prisma
> npm install @prisma/client
> npx prisma init
```





  - Then, you can run prisma generate (after defined schema data model) :

```
> npx prisma generate
```

Using Prisma Client to send queries to your database

- Once Prisma Client has been generated, you can import it in your code and send queries to your database.

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

```
findAll: async  function () {
    return await prisma.subject.findMany();
},
findById: async function (id) {
    return await prisma.subject.findUnique({
        where: { id: id }
    });
},
save: async function (subject) {
    return await prisma.subject.create({
        data: subject
    });
},
```

```
update: async function (subject) {
    return await prisma.subject.update({
        where: { id: subject.id },
        data: subject
    });
},
deleteById: async function (id) {
    return await prisma.subject.delete({
        where: { id: id }
    });
}
```

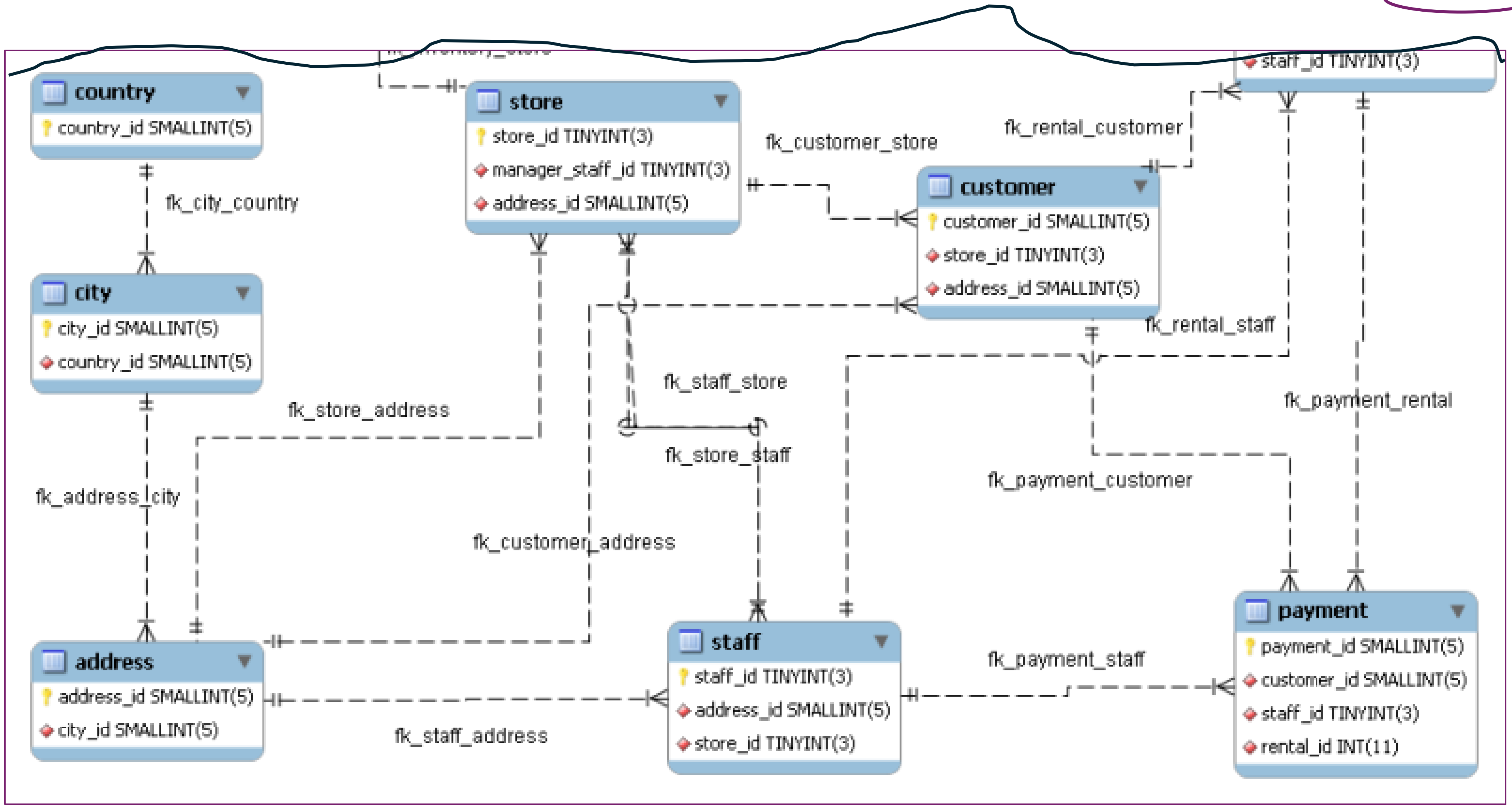
Prisma Workflow Summary

- Install Prisma
  - npm install prisma
  - npm install @prisma/client
  - npx prisma init
- Config Datasource
  - Edit .env → DATABASE_URL
- Modify Data Model (Prisma Schema)
  - prisma/schema.prisma
- Generate Client
  - npx prisma generate (When Data Model Changed)
- Using
  - const { PrismaClient } = require('../generated/prisma')
  - const prisma = new PrismaClient()
  - const users = await prisma.user.findMany()

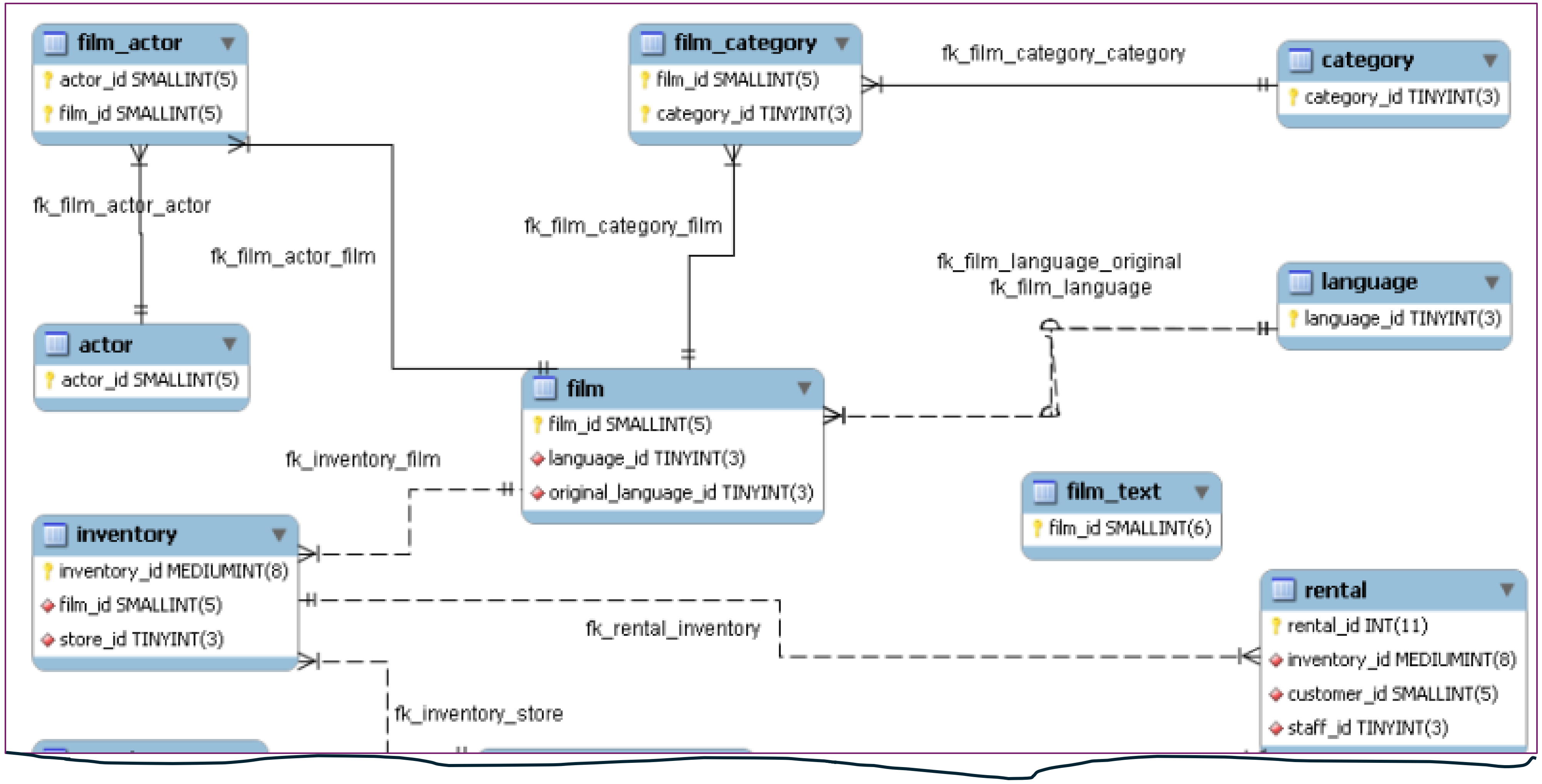
```
datasource db {
provider = ”mysql"
  url = env("DATABASE_URL")
  output = "../generated/prisma"
}
```

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

Sakila

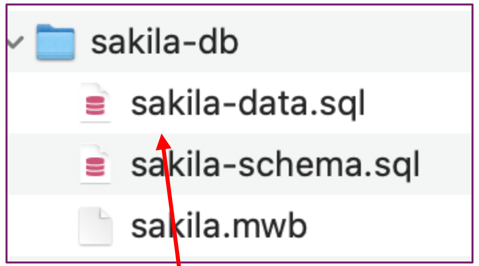


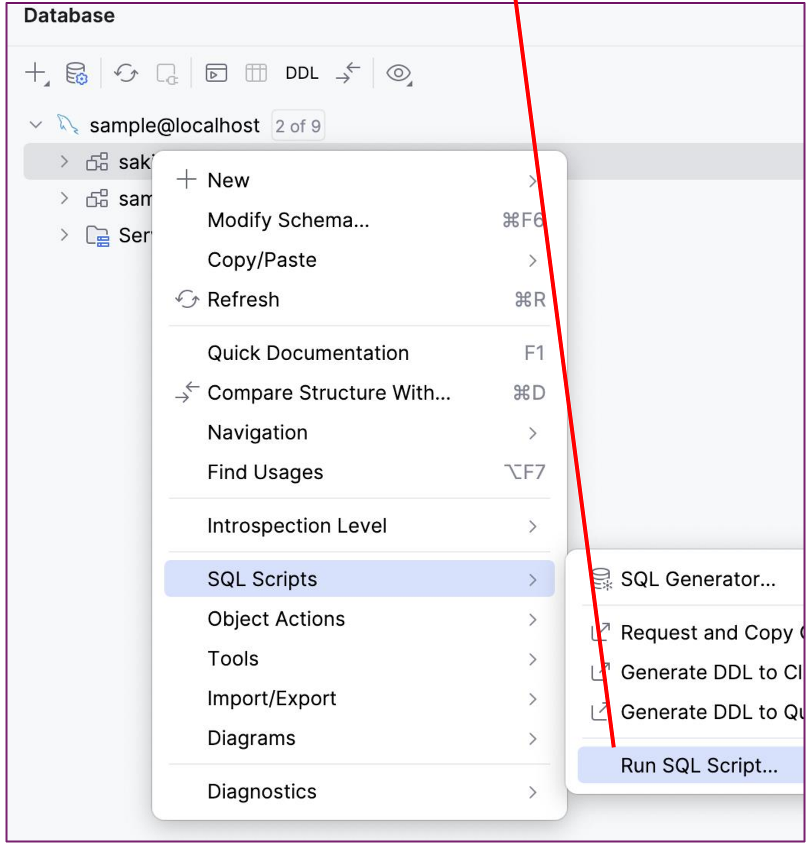
https://dev.mysql.com/doc/sakila/en/sakila-structure.html



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"
```

POC (1/2)

- Duplicate Project
  - Copy simple-api-db → simple-api-prisma
- Install Prisma
  - Open Project simple-api-prisma
  - Run CLI

```
> npm install prisma
> npm install @prisma/client
> npx prisma init
```

- Edit .env
- Modify Schema `(./prisma/schema.prisma)`

```
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)

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

POC (2/2)

- Generate Client

```
> npx prisma generate
```

- Using Prisma ORM
  - Create test-prisma.js

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

async function main() {
   const customers = await prisma.customer.findMany();
   console.log(customers);
}
main();
```

  - Run test-prisma.js

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}

RESTful API Characteristics

- Stateless Communication
- Scalability
- Cacheability
- Client-Server Separation
- Uniform Interface

Rules of REST API

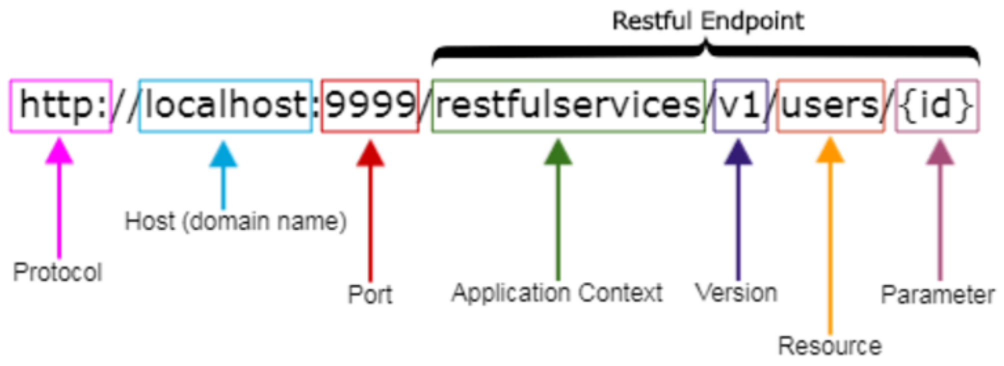
- There are certain rules which should be kept in mind while creating REST **API endpoints**.
  - REST is based on the resource or **noun** instead of action or verb based. It means that a URI of a REST API should always end with a noun. Example: **/api/users** is a good example.
  - HTTP verbs are used 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

Country Repository : `country-repository.js`

```
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
         }
      });
   },
```

```
save: async function (newData) {
      return await prisma.country.create({
         data: newData
      });
   },
   update: async function (newData) {
      return await prisma.country.update({
         where: { id: newData.id },
         data: newData
      });
   },
   deleteById: async function (id) {
      return await prisma.country.delete({
         where: { id: id }
      });
   }
}
```

Country Service : `country-service.js (1/2)`

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

function validateBody(body) {
   if (!body || Object.keys(body).length === 0) {
      const err = new Error('Bad Request: empty body');
      err.status = 400;
      throw err;
   }
   const { country } = body;
   if (!country || typeof country !== 'string') {
      const err = new Error(
            'Bad Request: country is required');
      err.status = 400;
      throw err;
   }
}
```

```
module.exports = {
   getAll: async function () {
      const array = await repo.findAll();
      return array;
   },
   add: async function (newData) {
      validateBody(newData);
      const created = await repo.save(newData);
      return created;
   },
```

Country Service : `country-service.js (2/2)`

```
getById: async function (id, includeCities = false) {
    const uniqueOne = await repo.findById(id, includeCities);
    if (! uniqueOne) {
        const err = new Error(`Country not found for ID ${id}`);
        err.code = 'NOT_FOUND';
        err.status = 404;
        throw err;
    }
    return uniqueOne;
},
update: async function (id, updateData) {
    validateBody(updateData);
    updateData.id = id;
    const updated = await repo.update(updateData);
    return updated;
},
```

```
    remove: async function (id) {
        const removed = await repo.deleteById(id);
        return removed;
    }
}
```

Country Controller : `country-controller.js (1/3)`

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

function error(req, error, message, statusCode) {
    return {
        error: error,
        statusCode: statusCode,
        message: message,
        path: req.originalUrl,
        timestamp: new Date().toLocaleString()
    };
}
```

```
function sendError(req, res, e) {
    let status = e.status || 500;
    if (e.code === 'P2002') {
        status = 400;
        e.code = 'BAD_REQUEST';
    } else if (e.code === 'P2025') {
        status = 404;
        e.code = 'NOT_FOUND';
    }
    const index = e.message.indexOf('(\n');
    let message = e.message;
    if (index > 0) {
        message = e.message.substring(index + 2);
    }
    res.status(status).json(error(req, e.code,
message, status));
}
```

Country Controller : `country-controller.js (2/3)`

```
function validateId(req, res) {
   const idStr = (req.params.id || '').toString().trim();
   if (idStr === "") {
      return res.status(400).json(error(req, "Bad Request",
         "Bad Request: empty id", 400));
   }
   const id = Number(idStr);
   if (isNaN(id)) {
      return res.status(400).json(error(req, "Bad Request",
         "Bad Request: id must be a number", 400));
   }
   return id;
}
```

```
module.exports = {

   list: async function (req, res) {
      try {
         const countries = await service.getAll();
         res.json(countries);
      } catch (e) {
         sendError(req, res, e);
      }
   },
   get: async function (req, res) {
      const id = validateId(req, res);
      try {
         const country = await service.getById(id);
         res.json(country);
      } catch (e) {
         sendError(req, res, e);
      }
   },
```

Country Controller : `country-controller.js (3/3)`

```
    create: async function (req, res) {
      try {
        const created = await service.add(req.body);
        res.status(201).json(created);
      } catch (e) {
        sendError(req, res, e);
      }
   },
   update: async function (req, res) {
      const id = validateId(req, res);
      try {
        const updated = await service.update(id,
req.body);
        res.json(updated);
      } catch (e) {
        sendError(req, res, e);
      }
   },
```

```
   remove: async function (req, res) {
      const id = validateId(req, res);
      try {
        await service.remove(id);
        res.status(204).send();
      } catch (e) {
        sendError(req, res, e);
      }
   },
   getCities: async function (req, res) {
      const id = validateId(req, res);

      try {
        const country = await service.getById(id,true);
        res.json(country.cities);
      } catch (e) {
        sendError(req, res, e);
      }
}
```

Country Router : `country-route.js`

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

router.get('/', controller.list);
router.get('/:id', controller.get);
//router.post('/:id', controller.addCity);
router.get('/:id/cities', controller.getCities);
router.post('/', controller.create);
router.put('/:id', controller.update);
router.delete('/:id', controller.remove);

module.exports = router;
```