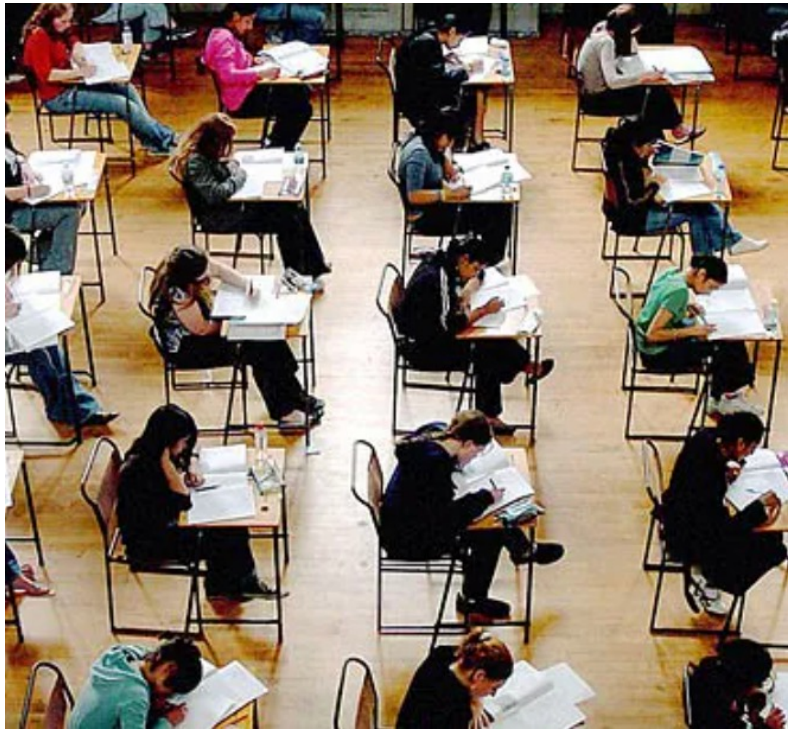

CRUD

CSC309

Kianoosh Abbasi

UNIVERSITY OF TORONTO

Fall 2024

So far

- Next.js API handlers
- Async programming and event loop
- MVC and model design
- Prisma ORM

Recap: Prisma

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

- Install via

```
npm i prisma @prisma/client @prisma/studio
```

- Create a file named `schema.prisma`

- Prisma generates JS classes from its schema file
  - And syncs it with the database schema
  - More on that later in the course

Recap: sync with database

- The schema file does not automatically impact anything!

- To generate the relevant JS classes:
  - Run `npx prisma generate`

- To sync the schema with the database:
  - Run `npx prisma migrate dev`

- More on these commands later in the course!

Recap: example model

```prisma
model Product {
  id          Int          @id @default(autoincrement()) // Primary key with auto-increment
  name        String       // Required string field
  description String?      // Optional string field
  price       Decimal      @default(0.00) // Decimal field with default value
  sku         String       @unique // Unique constraint
  inStock     Boolean      @default(true) // Boolean field with default value
  quantity    Int          @default(0) // Integer field with default value
  createdAt   DateTime     @default(now()) // DateTime field with default value
  updatedAt   DateTime     @updatedAt // Auto-update DateTime field

  categoryId  Int          // Foreign key for category relation
  category    Category     @relation(fields: [categoryId], references: [id])

  Store       Store?       @relation(fields: [storeId], references: [id])
  storeId     String?

  Transaction Transaction[]
}
```

CRUD

- Stands for Create, Read, Update, and Delete

- Runtime database operations are one of these:
  - Data Access Level and Data Manipulation Level queries

Prisma and Next.js

- Define a client instance (typically in a separate file)

```
import { PrismaClient } from '@prisma/client’
export const prisma = new PrismaClient()
```

- Import the instance in the API handlers

Database operations

- Async functions in the form of `prisma.<model_name>.<operation_name>`
  - Examples:
    `prisma.product.findMany(…)`
    `prisma.user.create(…)`
    `prisma.category.update(…)`

- Translated into SQL (or NoSQL) queries by the ORM
  - Results poured into JS objects by the ORM

Select queries

- The `where` object specifies filters
- `include` specifies related objects to fetch
  - Translate to JOIN
- Alternative methods
  `findFirst`
  `findUnique`
  `findUniqueOrThrow`

```
const products = await prisma.product.findMany({
  where: {
    name: {
      contains: req.query.productName,
    },
    price: {
      lt: 3000,
    },
    isAvailable: true,
  },
  include: {
    store: true,
  },
})
```

Insert queries

- The `data` object specifies column values
- Related fields
  - Use `create`, `connect`, or `connectOrCreate`
- Returns the created object
  - Could be used in API response

```
await prisma.store.create({
  data: {
    name: req.body.name,
    description: req.body.description,
    address,
    owner: {
      create: {
        username: generateRandomString(10),
        firstName: 'store owner',
        lastName: 'store owner last name',
      },
    },
    sector: {
      connect: {
        id: sectorId,
      },
    },
  },
})
```

Update and delete queries

```
await prisma.user.update({
  where: { id: 'test-user' },
  data: { password: hashPassword(newPassword) },
})
```

Side note: never store raw passwords in the database

```
await prisma.product.delete({
  where: {
    id: productId,
    storeId: req.store.id,
  },
})
```

Notes

- `update` and `delete` throw an exception if the `where` clause does not match to exactly one record
  - Alternative: `updateMany` and `deleteMany`

- You can define conditions on related objects in the where clause as well

- Use `select` and `include` to customize which fields should be present in the returned object(s)
  - `select`: only selects the specified columns and/or related objects
  - `include`: includes specified related objects on top of existing columns

Exercise: CRUD APIs

Validation

- Request data should be thoroughly validated
  - Required fields must exist
  - Field types (e.g., number, string) must match
  - Specific formats (e.g., email, phone number) must be checked

- NEVER trust the user or frontend
  - In fact, they should be distrusted
    - Malicious clients

Validation

- Return a 400 response for malicious/invalid data
  - Run the validations before executing the database queries
- Running a query with unclean data is very dangerous
  - Security vulnerability: might be stored somehow
  - Bad experience: might crash and cause a 500 response with unhelpful errors
- Remember: users can send anything!
  - JavaScript does not do any type enforcement or validation!

Next session

- Authentication and authorization
- Tokens and sessions
- Detailed discussion about migrations