DAVID LIN

CSC309 Week 3

APIs, Next.js, Clean Architecture and Memes

HTTPS://WWW.CS.TORONTO.EDU/~KIANOOSH/COURSES/CSC309H5/

21 JANUARY, 2025

Breakdown

Welcome to third tutorial of CSC309!

APIs

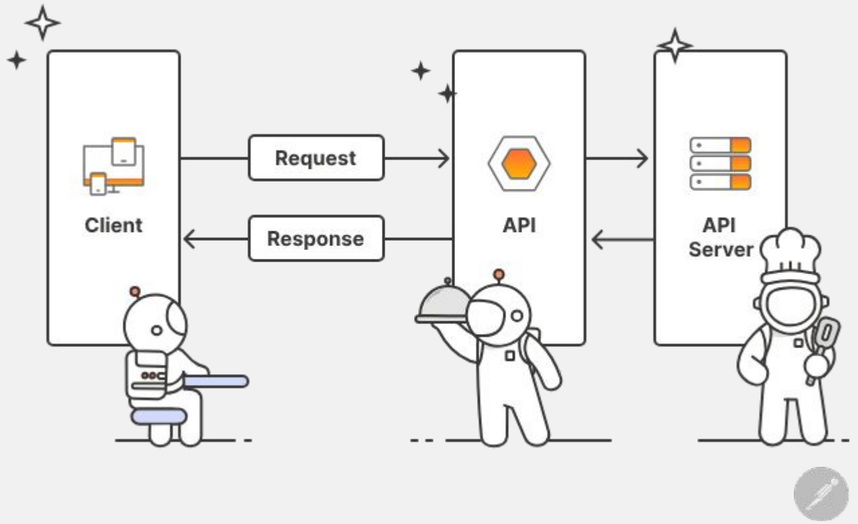
Next.js,

Clean Architecture

Case Study

What are APIs

- API: Application Programming Interface, duh
- A defined set of rules and specifications that allows one piece of code to interact with another
- Acts as a contract between different software components about how they can interact



Real Life Example

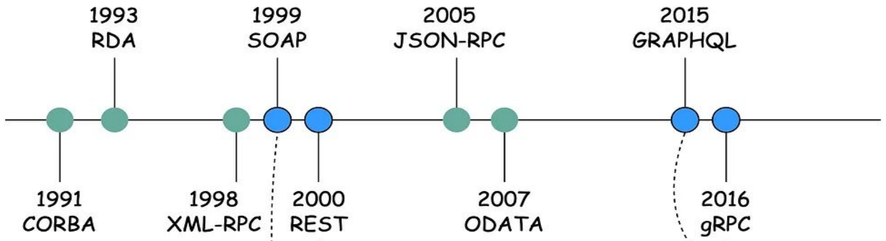
Think of it like a restaurant's menu and ordering system:

- The menu is the "interface" - it tells you what you can order (available operations)
- You don't need to know how the kitchen works (implementation is hidden)
- You have a standard way to request food (place order with waiter)
- The kitchen provides a predictable response (your food)

A Brief History of APIs

API Architectural Styles Comparison

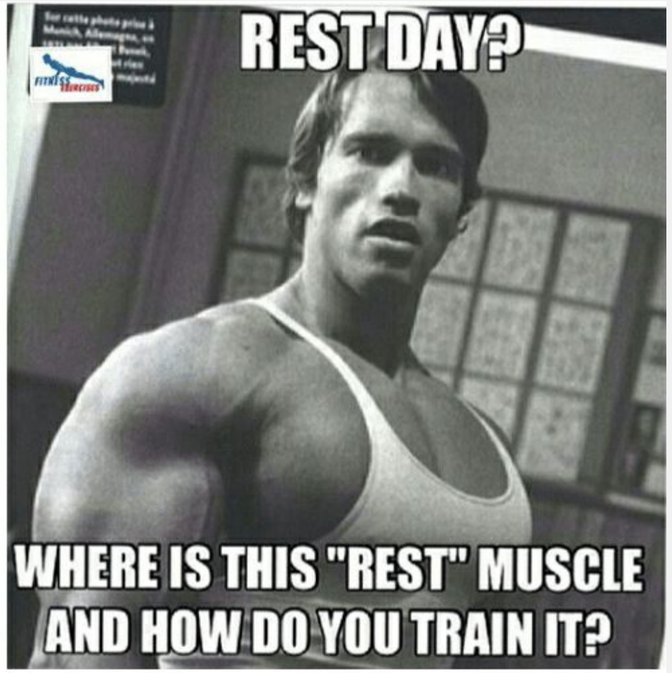
Source: altexsoft



	SOAP (Simple Object Access Protocol)	REST (REpresentational State Transfer)	GraphQL	RPC (Remote Procedure Call)
Organized in terms of	enveloped message structure	compliance with six architectural constraints	schema & type system	local procedure call
Format	XML only	XML, JSON, HTML, plain text	JSON	JSON, XML, Protobuf, Thrift, FlatBuffers
Learning curve	Difficult	Easy	Medium	Easy
Community	Small	Large	Growing	Large
Use cases	- payment gateways - identity management - CRM solutions - financial and telecommunication services - legacy system support	- public APIs - simple resource-driven apps	- mobile APIs - complex systems - micro-services	- command and action-oriented APIs - high performance communication in massive micro-services systems

REST

- Representational State Transfer, duh
- **Common interview question**
- Everything in REST revolves around "resources" (e.g., a user, a file, a product). Each resource is uniquely identified by a URL.



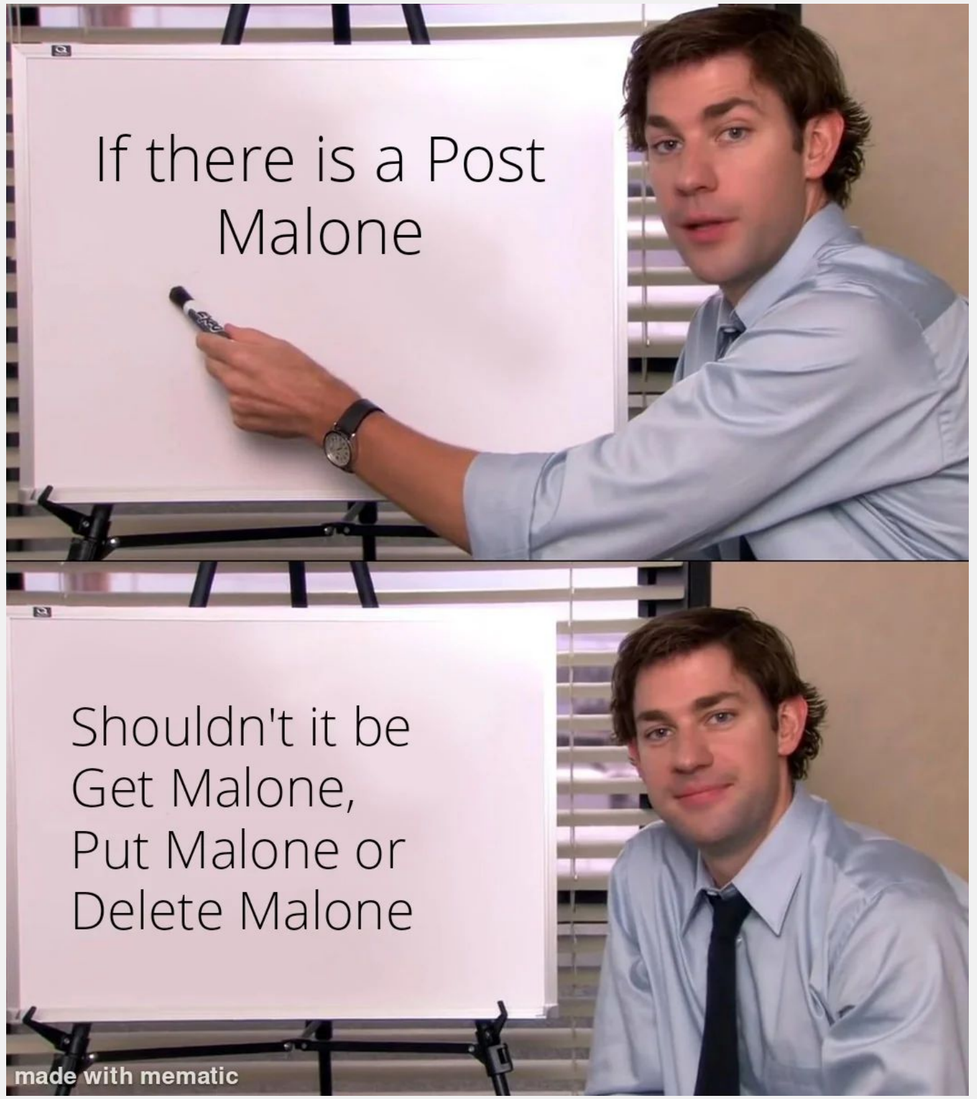
Rest Day... What!!!!

REST

- Instead of directly accessing a resource, systems send back and forth representations of it (e.g., JSON).

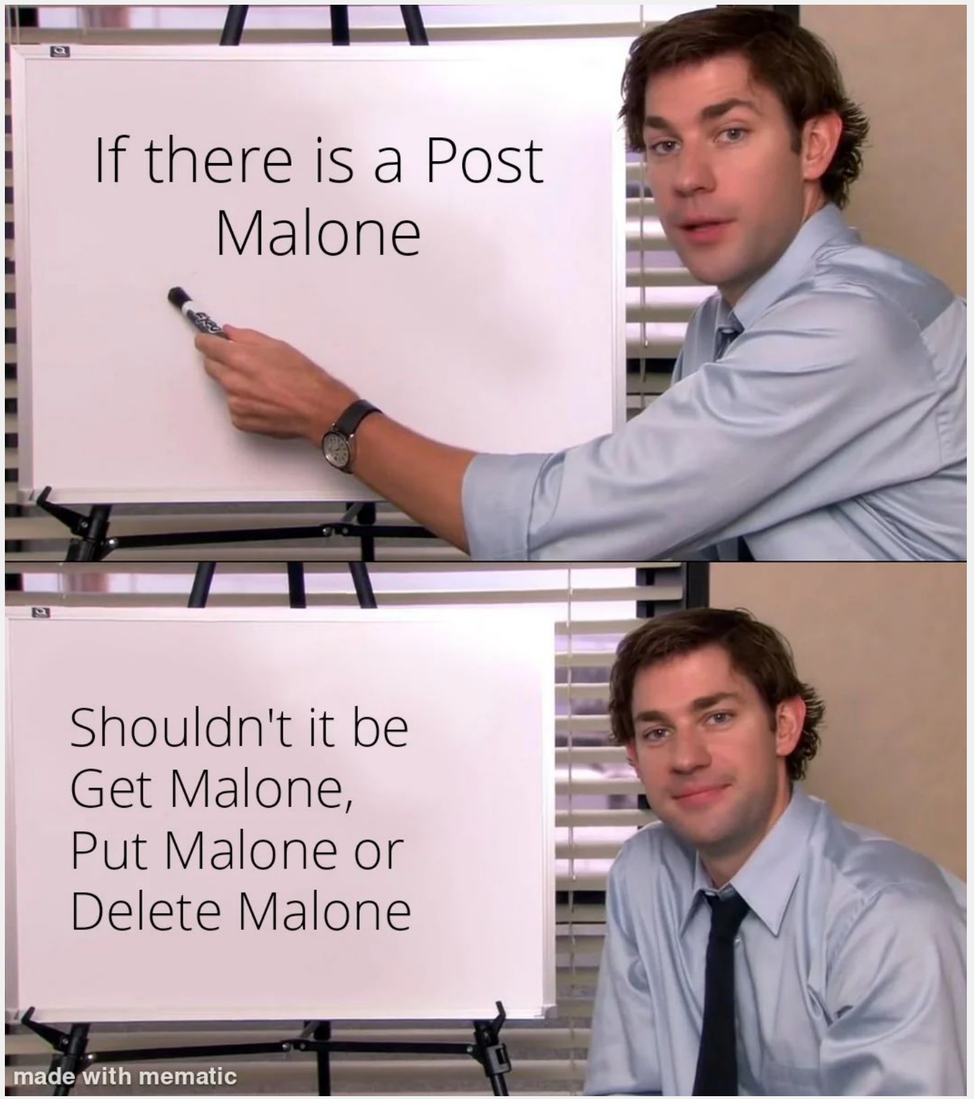
REST

- REST uses a small set of standardized HTTP methods (like GET, POST, PUT, DELETE) to perform actions on these resources.
- REAL INTERVIEW QUESTION: PUT VS PATCH



REST

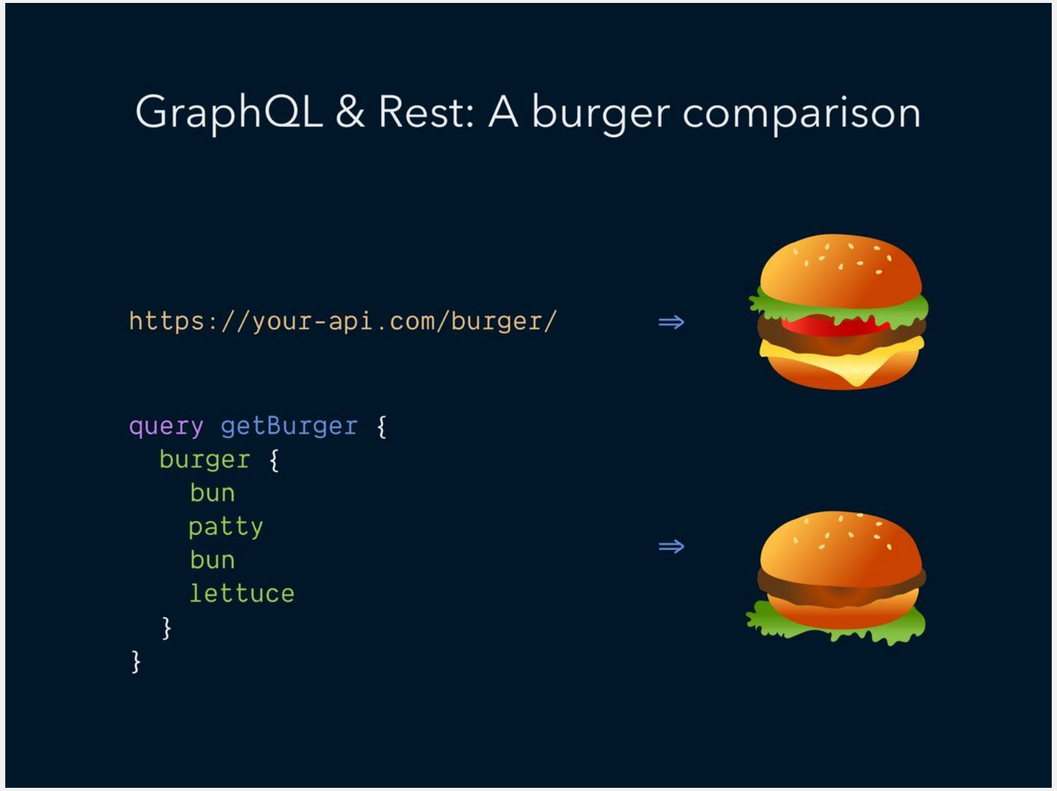
- Each interaction is independent. The server doesn’t remember what the client did previously—everything needed for a request is included in that request.

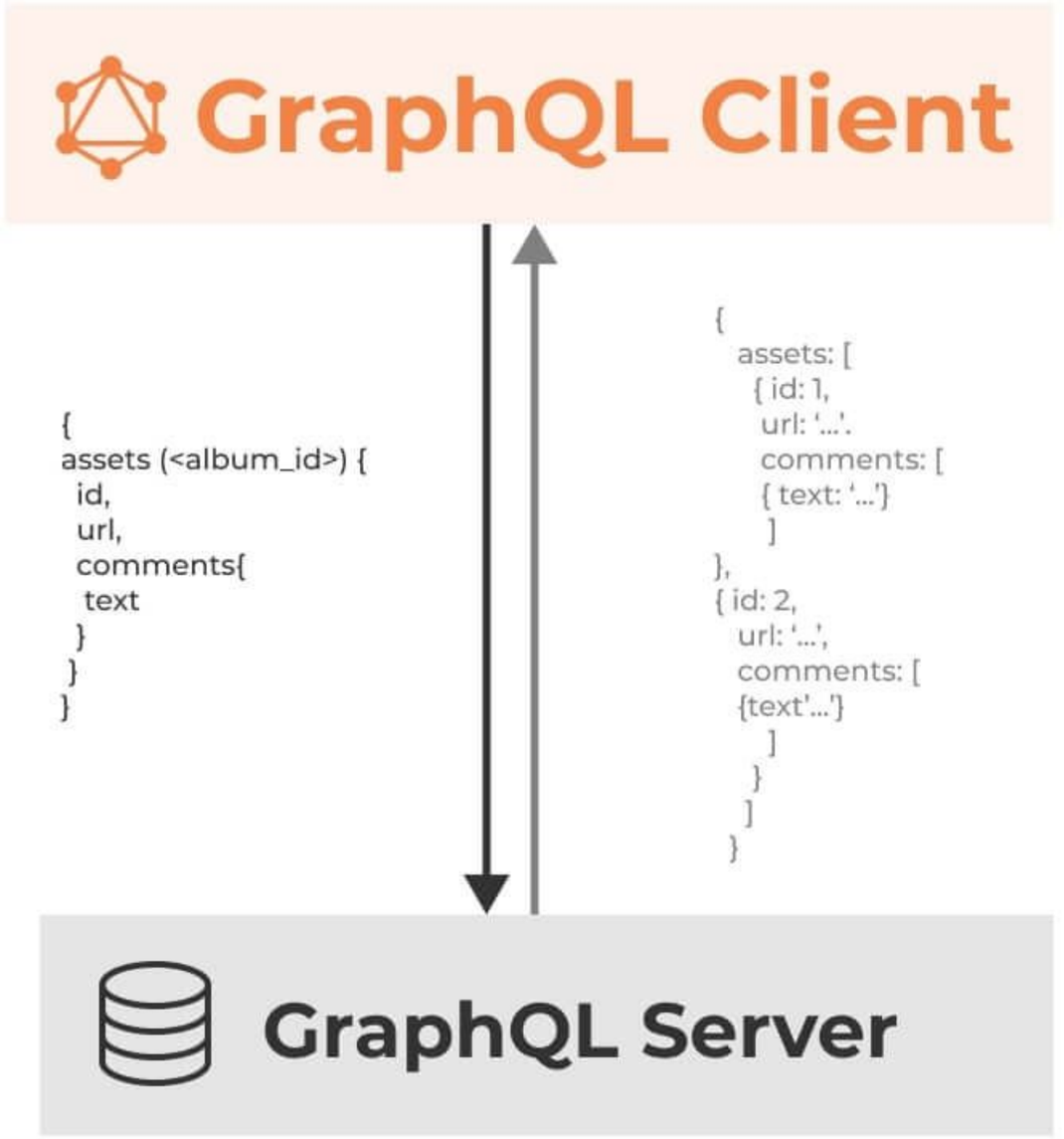


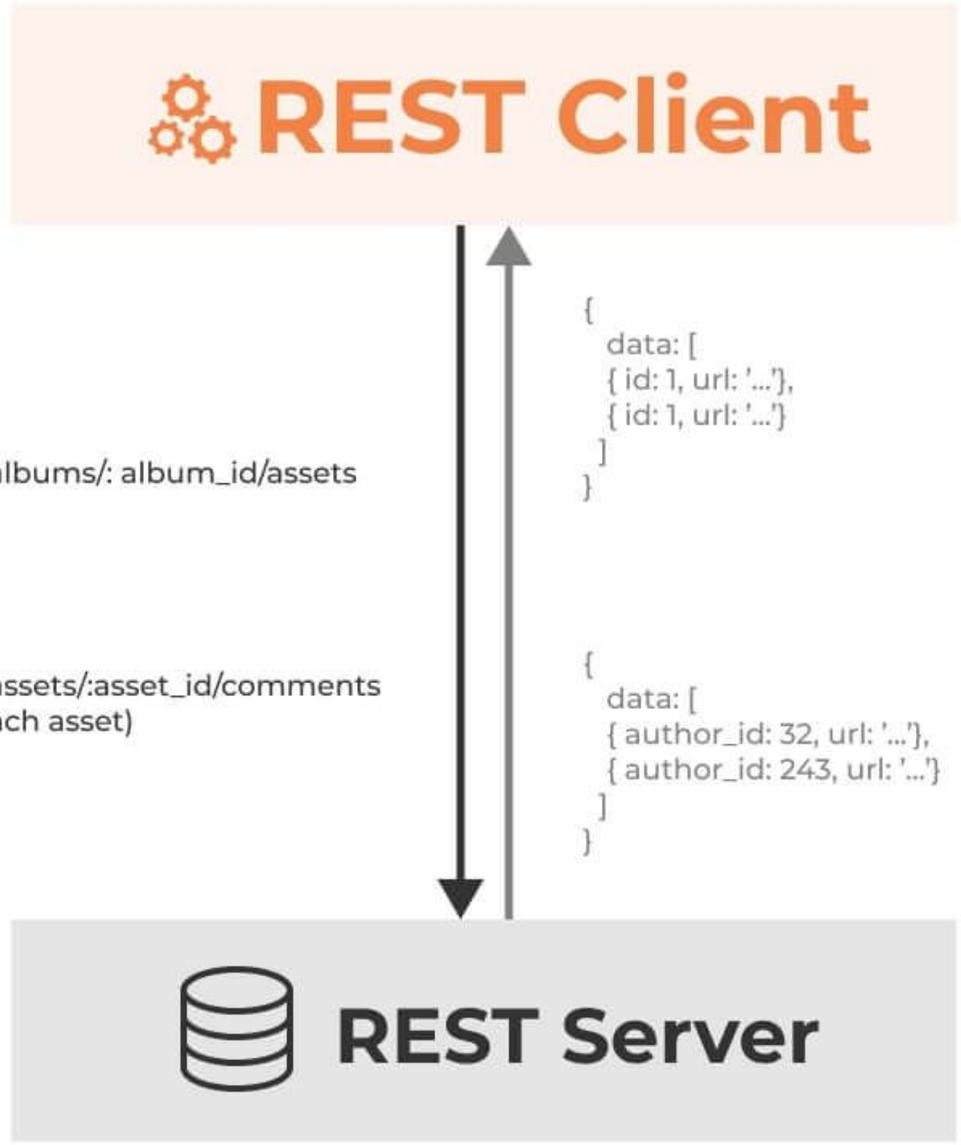
(Unfortunately A Stateful Image)

REST vs GraphQL

- REST
  - Multiple endpoints, each focusing on a resource
  - Easy to cache
  - Straightforward and widely adopted
- GraphQL
  - Single endpoint, usually **POST /graphql**
  - Flexible queries
  - Increased complexity on server-side setup




GraphQL Client
{
assets (<album_id>) {
id,
url,
comments{
text
}
}
}
{
assets: [
{ id: 1,
url: '...'.
comments: [
{ text: '...'}
]
},
{ id: 2,
url: '...',
comments: [
{text'...'}
]
}
]
}
GraphQL Server

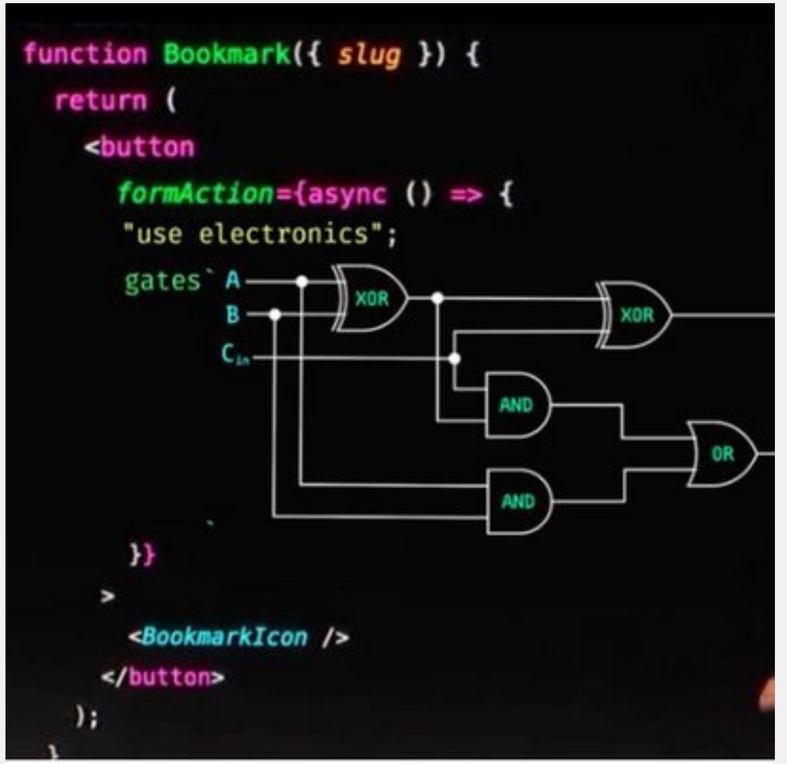

REST Client
GET /albums/: album_id/assets
GET /assets/:asset_id/comments
(for each asset)
{
data: [
{ id: 1, url: '...'},
{ id: 1, url: '...'}
]
}
{
data: [
{ author_id: 32, url: '...'},
{ author_id: 243, url: '...'}
]
}
REST Server

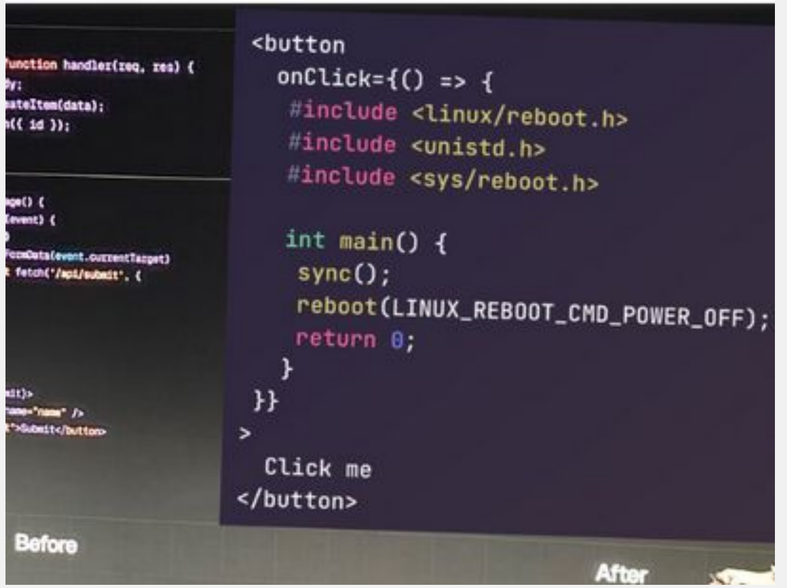
Demo

- REST API
  - https://github.com/r-spacex/SpaceX-API/blob/master/docs/README.md
- GraphQL
  - https://studio.apollographql.com/public/SpaceX-pxxbxen/variant/current/explorer

Next.js

and its Family

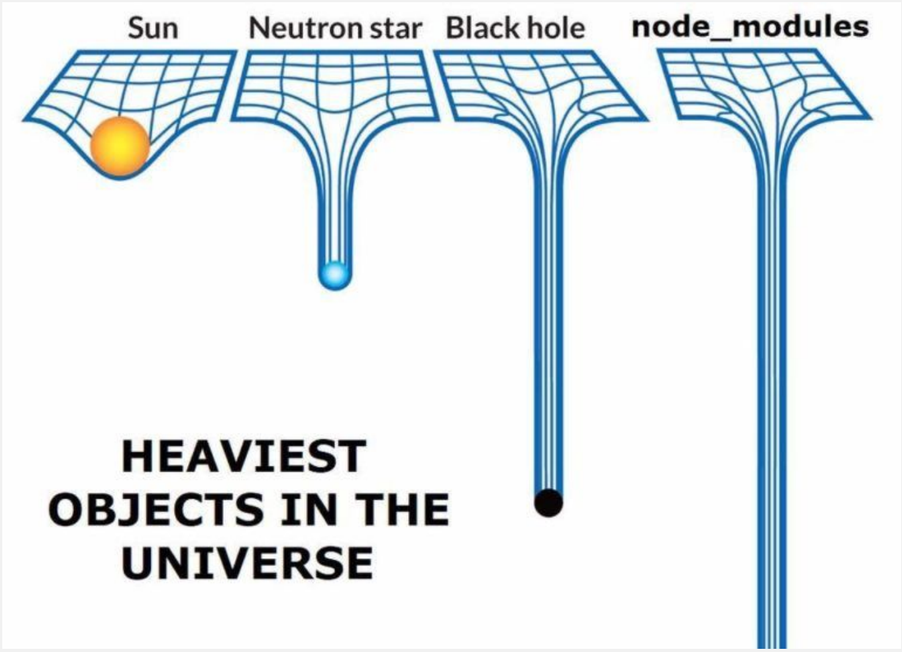




JS Naming Be Like...

Node.js

- A runtime environment for executing JavaScript outside the browser
- Built on Google’s V8 JavaScript engine
- Features:
  - Non-blocking, event-driven architecture
  - Package management through npm (Node Package Manager)
- Example
  - $ node # Start Node.js console
  - $ node <filename>.js # Execute a JavaScript file

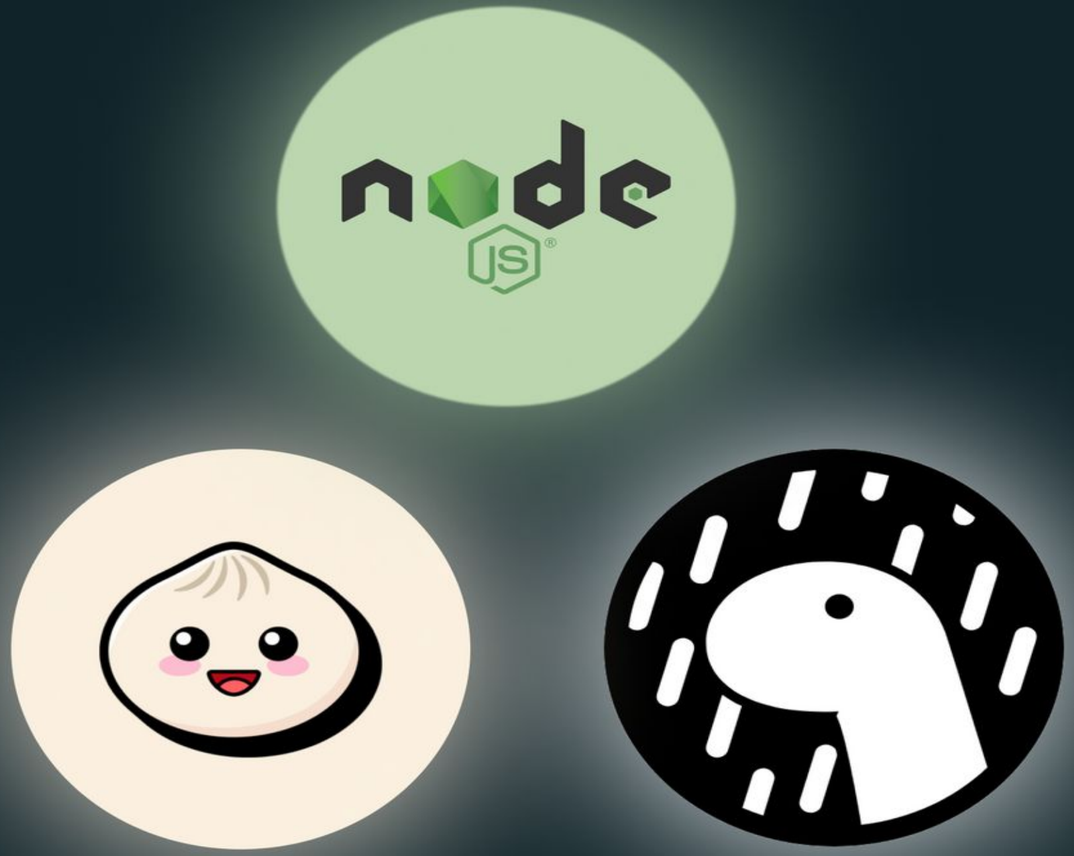


NPM

- npm (Node Package Manager) is used to manage libraries and tools for Node.js projects
- npm install <package-name> to add a package to package.json
- Workflow
  - Updating: Push package.json
  - Cloned Project: Run **npm install** to install all dependencies in package.json to node_modules

NPM Files

- node_modules
  - Directory where all installed packages and their dependencies are stored
- package.json
  - Contains metadata about the project, such as project name, version, and description
- package-lock.json
  - Ensures consistent installs by locking exact versions of dependencies
- NEVER PUSH NODE_MODULES


node
JS

Next.js

- A framework built on top of React and Node.js for server-side rendering and easy routing
- Gives you sweet features like SSR (Server-Side Rendering) and static site generation

Next.js & Node.js

1. User Request → HTTP Request sent to the server
2. Next.js→ Determines how to handle the request
    a. Static Site Generation (SSG)
    b. Server-Side Rendering (SSR)
    c. API Routes
3. Node.js Runtime → Used under the hood by Next.js to execute server-side JavaScript
4. Response to User → Sends formatted response

Request Handling - Headers

- Accessing Request Headers
  - Headers provide metadata about the request.
- Common use cases
  - Content type validation
  - Authorization tokens

app/api/route.ts — TypeScript

```ts
import { headers } from 'next/headers'

export async function GET(request: Request) {
  const headersList = await headers()
  const referer = headersList.get('referer')

  return new Response('Hello, Next.js!', {
    status: 200,
    headers: { referer: referer },
  })
}
```

Request Handling - Params

- What are Query Parameters?
  - Parameters passed in the URL after ?
  - Example: /api/items?search=book&page=2

app/api/search/route.ts TypeScript

```ts
import { type NextRequest } from 'next/server'

export function GET(request: NextRequest) {
  const searchParams = request.nextUrl.searchParams
  const query = searchParams.get('query')
  // query is "hello" for /api/search?query=hello
}
```

Request Handling - Dynamic

- From Next.js Documentation
- A Dynamic Segment can be created by wrapping a folder's name in square brackets: [folderName]. For example, [id] or [slug]
- Catch-all by adding an ellipsis inside the brackets [...folderName]
  - Example: **app/shop/[...slug]/page.js** will match **/shop/clothes**, but also **/shop/clothes/tops**, **/shop/clothes/tops/t-shirts**, and so on.

app/items/[slug]/route.ts — TypeScript

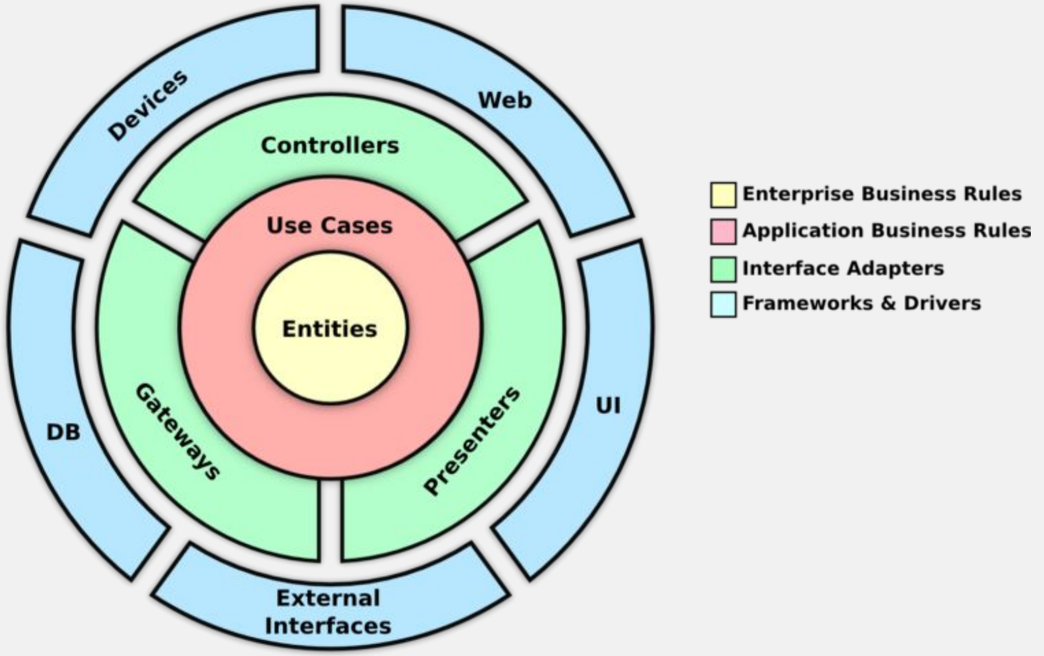
```ts
export async function GET(
  request: Request,
  { params }: { params: Promise<{ slug: string }> }
) {
  const slug = (await params).slug // 'a', 'b', or 'c'
}
```

Route	Example URL	`params`
`app/items/[slug]/route.js`	`/items/a`	`Promise<{ slug: 'a' }>`
`app/items/[slug]/route.js`	`/items/b`	`Promise<{ slug: 'b' }>`
`app/items/[slug]/route.js`	`/items/c`	`Promise<{ slug: 'c' }>`

Clean Architecture in Next.js

Clean Architecture in Next.js

- Separation of concerns for scalable and maintainable code
- Decoupling business logic from frameworks and tools
- Layers: Entities, **Service/Use Cases, Controllers**, Frameworks & Drivers
- All the source code dependencies should point inwards



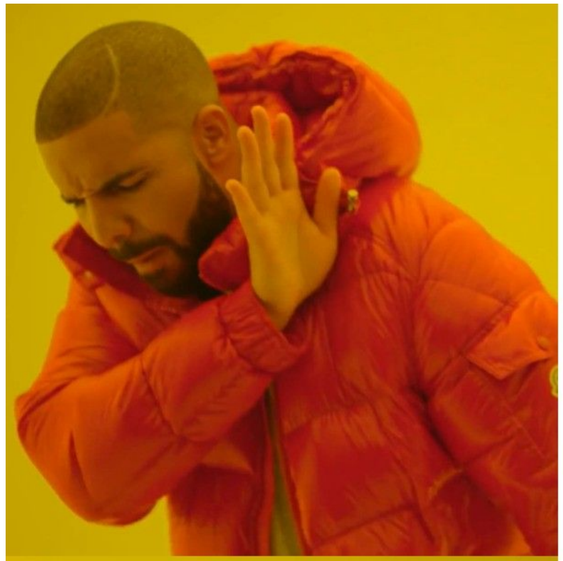
```
import { prisma } from "../../../infrastructure/prismaClient";
import bcrypt from "bcrypt";
import jwt from "jsonwebtoken";

export async function POST(req: Request) {
  try {
    // Parse request body
    const { title, content, userEmail, userPassword } = await req.json();
    if (!title || !content || !userEmail || !userPassword) {
      return NextResponse.json({ error: "Missing required fields" }, { status: 400 });
    }

    // Fetch user from database
    const user = await prisma.user.findUnique({
      where: { email: userEmail },
    });

    if (!user) {
      return NextResponse.json({ error: "User not found" }, { status: 404 });
    }

    // Validate user password
    const isPasswordValid = await bcrypt.compare(userPassword, user.password);
    if (!isPasswordValid) {
      return NextResponse.json({ error: "Invalid password" }, { status: 401 });
    }

    // Generate JWT token
    const token = jwt.sign({ userId: user.id }, process.env.JWT_SECRET || "secret", {
      expiresIn: "1h",
    });

    // Save post to database
    const newPost = await prisma.post.create({
```

Entities

- Entities represent core business objects and validation logic
- Custom errors for better error handling

```
export class Post {
  constructor(
    public id: string,
    public title: string,
    public content: string
  ) {}
}
```

Service Layer



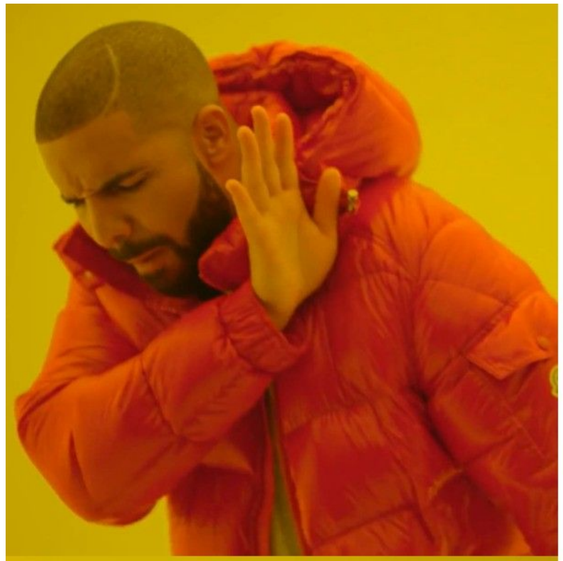
- Handle HTTP requests, headers, or responses
- Include plain database queries
  - Use ORM such as Prisma or repository instead, don’t worry about this for now
- Framework-specific dependencies like NextResponse

- Encapsulate business logic
- Interact with domain entities
- Data transformation and validation before interacting with the domain
- Reusable across controllers

Service Layer Example

```ts
// services/PostService.ts
import { Post } from "../domain/Post";

export class PostService {
  async getPosts(): Promise<Post[]> {
    // Simulate fetching from DB or API
    return [
      new Post("1", "Clean Architecture", "Learn about Clean Architecture."),
      new Post("2", "Service Layers", "Simplify your code."),
    ];
  }
}
```

Controller Layer



- Contain any business logic
- Query the database
- Include reusable workflows—these belong in the service layer

- Adaptor between services and frameworks (e.g., Next.js routes)
- Handle HTTP-specific tasks such as
  - Parsing incoming requests
  - Formatting outgoing responses
  - Managing headers and status codes
- Call the service layer to execute business logic

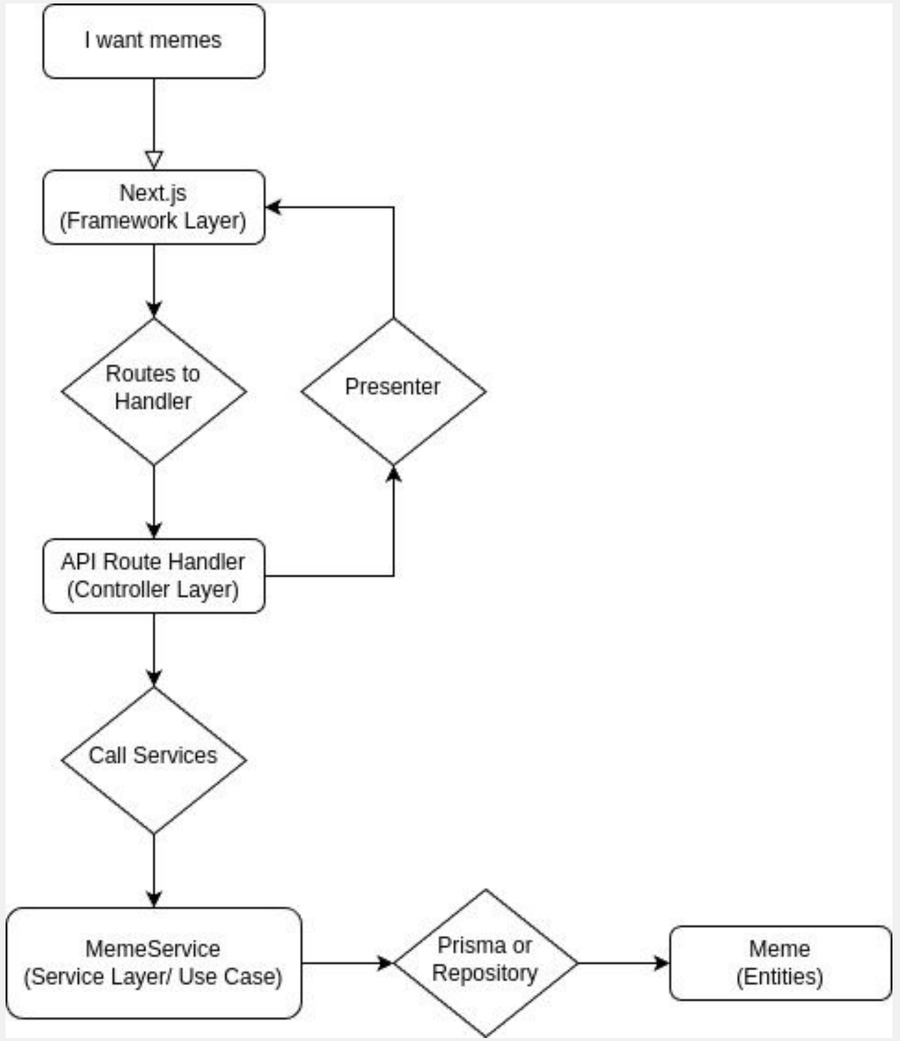
Controller Layer Example

```ts
// src/app/api/posts/route.ts
import { NextResponse } from "next/server";
import { PostService } from "../../../services/PostService";

const postService = new PostService();

export async function GET() {
  try {
    const posts = await postService.getPosts();
    return NextResponse.json(posts);
  } catch (error) {
    // For example
    return errorService(error);
  }
}
```

Framework/Infra

- It should only implement the technical details of interacting with external systems
- Business rules should remain in the Service Layer or Domain Layer
- The Infrastructure Layer provides services or repositories that the Service Layer consumes, not the other way around
- The Service Layer does not know the implementation details. Instead, the dependency is passed to it

Request Flow in Next.js

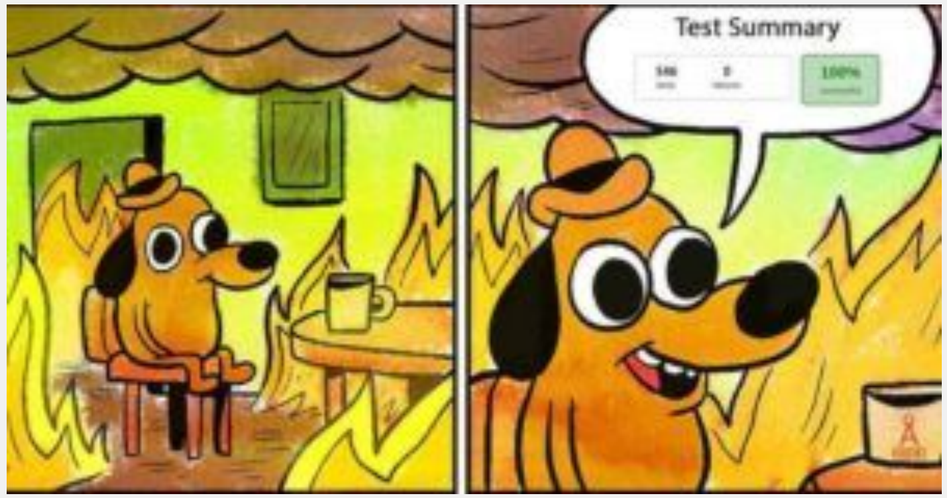


So GPT What Does It Mean?

Layer	What It Should Do	What It Shouldn't Do
Domain	Define entities and business rules.	Depend on frameworks or manage state.
Service	Encapsulate business workflows and coordinate logic.	Handle HTTP, headers, or database queries.
Adapter (Controller)	Handle HTTP requests and responses.	Contain business logic or interact directly with ORM.
Framework/Infra	Handle routing, database access, and external integrations.	Implement business rules or validations.

Testability

- **Independent Layers**: You can test each layer (like business logic or database access) separately.
- **Mocks and Stubs**: Easily replace real dependencies (e.g., databases) with fake ones in tests.
- **Isolated Logic**: Business rules don’t depend on external tools, so they’re easy to test.



DAVID LIN

Thank You

CSC309 Week 3

15 JANUARY, 2025

DAVID LIN

Please join the Zoom for polls

Meeting Code: **Password:**