Web Services Notes

Rest

- REST is an acronym name for **Representational State Transfer**
  - standardized way to provide data to other applications.
- it is used for building and communicating with web services.
- best way to transfer data across the applications
- mandates resources on the web are represented in
  - JSON,
  - HTML,
  - or XML.

API

- An API is an acronym for **Application Programming Interface**,
  - an interface that defines the interaction between different software components.
- Web APIs => what exactly request is made to the component.

IMP API METHODS

- GET
  - most common method for get some data from component.
  - returns some data from the API based on
    - the endpoint we hit
    - any parameter we pass.
- POST
  - creates the new records
  - updates the new created record in the database.
- PUT
  - takes the new records at the given URI.
  - If the record exists, update the record.
  - If record is not available, create a new record.
- PATCH
  - It takes one or more fields at the given URI.
  - used to update one or more data fields.
  - If the record exists, update the record.
  - If the record is not available, create a new record.
- DELETE
  - It deletes the records at the given URI.

REST FRAMEWORKS

- Rest frameworks, also known as RESTful frameworks or RESTful APIs,
    - tools and libraries that provide a set of conventions and functionalities to build and implement RESTful web services.
- REST stands for Representational State Transfer,
    - which is an architectural style for designing networked applications. It
    - is commonly used in web development to create APIs that
        - allow communication between client-side applications and server-side systems.

Key features of REST frameworks include:

- HTTP Methods:
    - RESTful APIs use standard HTTP methods like GET, POST, PUT, DELETE, etc., to perform operations on resources (data) exposed by the API.

- Resource-based URLs:
    - Resources are represented by URLs, making it easy for clients to access and manipulate them via HTTP methods.

- Statelessness:
    - RESTful APIs are stateless,
        - meaning that each request from a client to the server must contain all the information needed to understand and process the request.
    - The server does not store any client information between requests.

- Representations:
    - Resources can have multiple representations, such as JSON, XML, HTML, etc., allowing clients to choose the format they prefer.

Hypermedia as the Engine of Application State (HATEOAS): This principle suggests that the API responses should contain hyperlinks to related resources, allowing clients to navigate the API dynamically.

REST frameworks provide a structured way to implement these principles, handling HTTP request and response handling, routing, serialization, authentication, and other common tasks.

- Some popular REST frameworks include:
    - Django REST framework
    - Express.js
    - Spring Boot
    - ASP.NET Core

DJANGO REST FRAMEWORK

- Django Rest Framework (DRF) is a package built on the top of Django
    - create web APIs.
- provides the most extensive features of
    - Django,
    - Object Relational Mapper (ORM),
- which allows the interaction of databases in a Pythonic way.

- Hence the Python object can't be sent over the network, so we need to translate Django models into the other formats like JSON, XML, and vice-versa. This process is known as **serialization**, which the Django REST framework made super easy.

key components of the Django framework:

- Models:
    - handles the data layer.
    - developers to define the data models and relationships between them using Python classes.
    - These models are then translated into database tables, making it easy to work with the database without writing SQL queries directly.

- Views:
    - handle the business logic of the application.
    - receive requests from the user's browser, interact with the models to fetch data from the database, and then render templates to generate the HTML response to be sent back to the user.

- Templates:
    - responsible for generating the HTML presentation of the data.
    - They provide a way to separate the design from the business logic.
    - Django's template engine allows developers to embed Python code within the HTML templates, making it easy to dynamically generate content.

- URL Routing:
    - URL routing system that maps URLs to views.
    - The URLs are defined in a central URL configuration file, which makes it easy to organize and manage the different endpoints of the web application.

- Forms:
    - Django includes a powerful form handling system that simplifies the process of validating user input and processing form submissions.
    - This feature helps developers create and manage HTML forms in a more straightforward manner.

- Admin Interface:
    - automatic admin interface that allows developers to manage the application's data through a web-based interface without having to create a separate admin section manually.

- Middleware:
    - supports middleware, which allows developers to process requests and responses globally before they reach the view or after they leave the view.
    - This is useful for tasks like authentication, caching, and error handling.

- Security:
    - includes various built-in features to help developers protect against common web vulnerabilities, such as cross-site scripting (XSS), cross-site request forgery (CSRF), and SQL injection.

- Internationalization (i18n) and Localization (l10n):
    - Django provides tools to build applications that can be easily translated into multiple languages and localized for different regions.

Diff between rest and other frameworks

The main difference between REST frameworks and other frameworks lies in their focus and purpose. Let's look at the distinctions:

Aspect	REST Frameworks	Other Frameworks
Purpose	Specialized for building RESTful APIs	General-purpose for web app development
Architecture	Follows REST architectural style	May follow MVC or other patterns
Focus	API-centric	Application-centric
HTTP Methods	Relies on standard HTTP methods (GET, POST, PUT, DELETE)	May or may not use standard HTTP methods
Resource-Based URLs	Emphasizes resource-based URLs	URLs may or may not be resource-based
Statelessness	Stateless design	May or may not be stateless
Uniform Interface	Provides a uniform and consistent interface for API interactions	May have varied interfaces for different functionalities
CRUD Operations	Maps CRUD operations to HTTP methods	CRUD operations may not be standardized
Hypermedia (HATEOAS)	Follows HATEOAS principles	May or may not implement HATEOAS
Simplicity	Generally simpler and more opinionated	May be more flexible and customizable
Learning Curve	Lower learning curve	May have a steeper learning curve
Use Cases	Ideal for building APIs to expose data and services	Suitable for building complete web apps
Examples	Django REST framework, Express.js	Ruby on Rails, Laravel, Angular, React
	Flask, Spring Boot, etc.	

Structure:

- refers to the organization and design of the components that make up the web service
- includes how the code, data, and functionalities are organized and divided to create a cohesive and maintainable web service.
- A well-structured web service follows a clear architecture (e.g., RESTful, SOAP),
    - separates concerns effectively
    - follows best practices for code organization.
    - is easier to understand, modify, and scale,
    - crucial for maintaining and evolving a successful web service over time.

Schema:

- a schema refers to the definition of the data format used to exchange information between the client and the server.
- defines the structure and organization of the data that can be sent and received by the web service.
- For example, in a RESTful API, the schema may define the JSON or XML format used for representing resources and data.
- In a SOAP-based web service, the schema may be defined using XML Schema (XSD) to specify the structure of the XML messages.
- The schema ensures that data exchanged between the client and the server is
    - well-defined,
    - consistent,
    - can be parsed correctly by both sides.

Endpoints:

- endpoints are specific URLs that the web service exposes to clients for accessing its functionalities.
- Each endpoint corresponds to a specific operation or resource in the web service.
- For example, in a RESTful API, endpoints might include URLs like "https://api.example.com/products" for accessing a list of products or "https://api.example.com/orders/123" to retrieve details about a specific order.
- Clients (e.g., web browsers, mobile apps) interact with the web service by making HTTP requests to these endpoints.
- Each HTTP method (GET, POST, PUT, DELETE, etc.) typically maps to a specific action on the resource represented by the endpoint.

Network Calls:

- network calls refer to the communication that occurs between the client and the server over a network.
- client needs to interact with a web service, it initiates a network call by making an HTTP request to the appropriate endpoint.
- The HTTP request includes the necessary information for the server to understand the client's intent (e.g., HTTP method, request headers, request body).
- The server processes the request
- executes the appropriate actions
- sends an HTTP response back to the client.
- The HTTP response contains the
    - requested data
    - the result of the operation.
- Network calls are at the core of how web services allow clients to interact with and consume their functionalities over the internet.

Protocols:

- set of rules and conventions
    - define how different components of a web service communicate and exchange data with each other.
- play a crucial role in enabling
    - interoperability
    - Standardization
    - allowing web services to work seamlessly across
        - different platforms, devices, and programming languages
- Key protocols include:
    - HTTP
        - Fundamental for web communication, uses methods like GET, POST, etc.
    - SOAP
        - Uses XML for structured information exchange.

structure - organization of the web service's code and components
schema - data format used for communication
endpoints - URLs that clients use to access specific functionalities
network calls - requests and responses that facilitate communication between the client and the server.
Protocols - sets of rules and conventions

Node express javascript

- fast, minimalistic, and flexible web application framework for Node.js.
- provides a set of robust features and tools to build web applications and APIs with ease.
- Key points about Node.js Express:
    - Web Application Framework
        - Express simplifies the process of building web applications and APIs by providing essential functionalities for handling routes, middleware, and HTTP requests and responses.
    - Lightweight
        - Express is designed to be lightweight and unopinionated, allowing developers the freedom to structure their applications as they see fit.
    - Routing
        - Express enables easy definition of routes to handle different HTTP methods (GET, POST, PUT, DELETE, etc.) and URLs. It makes it effortless to respond to various client requests.
    - Middleware
        - Middleware functions are a critical aspect of Express. They can intercept and modify HTTP requests and responses, enabling tasks like authentication, logging, error handling, and more.
    - Template Engines
        - While not included by default, Express allows you to integrate template engines like EJS or Pug to render dynamic HTML pages on the server-side.
    - Robust Ecosyste
        - Express has a massive community and a vast ecosystem of third-party middleware, extensions, and plugins, providing additional features and functionalities to enhance development productivity.

request.js

```
// Importing the express module
const express = require('express');
const bodyParser = require('body-parser');
const url = require('url');
const querystring = require('querystring');

let app = express();
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
```

```
// Sending the response for '/' path
app.get('/' , (req,res)=>{
    let page = req.query.name;
    let limit = req.query.age;
    let data = {'name':page,"age":limit};
// Sending the data json text
res.json(data);
})

// Setting up the server at port 3000
app.listen(4000 , ()=>{
   console.log("server running");
});
```

Server.js

```
// Importing the express module
const express = require('express');
const app = express();

// Initializing the data with the following json
const data = {
   portal: "TutorialsPoint",
   tagLine: "SIMPLY LEARNING",
   location: "Hyderabad"
}

// Sending the response for '/' path
app.get('/' , (req,res)=>{

// Sending the data json text
res.json(data);
})

// Setting up the server at port 3000
app.listen(4000 , ()=>{
   console.log("server running");
});
```

https://www.tutorialspoint.com/nodejs/nodejs_express_framework.htm

https://www.tutorialspoint.com/express-js-express-json-function

GraphQL

- data query language and runtime
- developed by Facebook.
- provides a powerful and flexible approach to data fetching in web applications.
- In traditional RESTful APIs,
    - the server defines endpoints for specific resources,
    - clients can fetch data by making HTTP requests to these endpoints.
    - this can lead to overfetching, where clients receive more data than they need, or underfetching, where they need to make multiple requests to gather all the required data.
- With GraphQL, clients can specify exactly what data they need in their queries.
    - The client defines the structure of the response,
    - indicating the specific fields and relationships it wants.
    - allows clients to request a customized set of data
    - eliminating the problems of overfetching and underfetching
    - also reduces the number of requests required to fetch all the necessary data, as the server can respond with a single GraphQL query containing all the requested information.
- GraphQL can also be used for updating data on the server using mutations.
- Mutations allow clients to specify the changes they want to make to the data, and the server processes these mutations and returns a response.
- GraphQL has a strong typing system, which means that the data types of all the fields are explicitly defined.
    - This ensures that the responses are predictable and well-structured, making it easier for clients to understand and work with the data.

GraphQL is a query language and runtime for APIs that enables more efficient and flexible communication between clients (usually frontend applications) and servers (backend services). It was developed by Facebook and has gained popularity due to its ability to address many of the limitations of traditional REST APIs.

Here's an explanation of the key concepts in GraphQL: structure, schema, endpoints, and network calls.

Structure: GraphQL structures data in terms of types and fields. Each field on a type can be queried or mutated (for updates) directly by clients. The data structure is hierarchical and mirrors the way clients need the data. This contrasts with REST APIs where endpoints often return fixed data structures that might over-fetch or under-fetch data for specific use cases.

Schema: The schema is the contract between the client and the server, defining how data can be queried, what types of data can be retrieved, and how it can be modified. It is a crucial aspect of GraphQL. The schema is defined using the GraphQL Schema Definition Language (SDL), which outlines the types available, their fields, relationships, and the operations that can be performed.

Endpoints: Unlike REST APIs which often have multiple endpoints (URLs) for different resources, GraphQL typically has a single endpoint (URL) for all interactions. This simplifies the API surface and allows clients to request exactly the data they need in a single request, reducing over-fetching of data.

Network Calls: In GraphQL, clients make queries or mutations to the GraphQL server over HTTP. The client specifies the fields it needs, and the server responds with exactly those fields, avoiding over-fetching. The client constructs a query that resembles the structure of the data it wants. Network calls can be made using standard HTTP methods like POST or GET. The server processes the query, validates it against the schema, and returns the requested data in the shape requested.

To sum up the typical flow:

The client sends a GraphQL query to the server over HTTP.

The server parses and validates the query against the defined schema.

The server resolves the query, fetching the necessary data from the backend.

The server responds to the client with the requested data in the same shape as the query.

In addition to queries for fetching data, GraphQL also supports mutations for making modifications to the data on the server. This includes creating, updating, and deleting data.

Overall, GraphQL offers a more efficient and flexible approach to building APIs compared to traditional REST APIs. It allows clients to request just the data they need,

avoids multiple network requests, and provides a clear and predictable structure for API interactions.

Graph ql query mutation

GraphQL queries and mutations are two essential components of the GraphQL language, allowing clients to request and modify data on a server.

GraphQL Query:

- A GraphQL query is used to request data from the server.
- Clients can specify the exact data they need in the query,
- and the server responds with the requested data in the same shape as the query.
- The basic structure of a GraphQL query looks like this:

```
query {
        Field1
        Field2
         …
}
```

- In a query, clients can request specific fields and nested fields to retrieve the data they require. For example:

```
query {
        user(id: "123") {
        Name
        email
        posts {
                Title
                content
                 }
        }
}
```

In this example, the client is requesting the name and email fields of a user with ID "123" and also fetching the title and content fields of the user's posts.

GraphQL Mutation:

- A GraphQL mutation is used to modify data on the server.
- it allows clients to specify what changes they want to make,
    - Creating
    - Updating
    - deleting data.

- Mutations are similar in structure to queries but are defined using the mutation keyword. The basic structure of a GraphQL mutation looks like this:

```
mutation {
        createEntity(input: { field1: value1, field2: value2, ... }) {
                field
                Field2
                …
              }
    }
```

In a mutation, clients can specify the input data required to perform the operation. For example:

```
mutation {
  createPost(input: { title: "New Post", content: "This is a new post." }) {
    id
    title
    content
  }
}
```

In this example, the client is creating a new post by providing the title and content, and the server responds with the id, title, and content of the newly created post.

GraphQL queries and mutations provide a flexible and efficient way for clients to request and modify data on the server, making it a powerful language for building APIs that cater to specific data needs of modern web applications.

Diff between rest and graphql

Aspect	REST	GraphQL
Architectural Style	REST (Representational State Transfer)	GraphQL (Query Language for APIs)
Communication Protocol	Uses standard HTTP methods (GET, POST, PUT, DELETE, etc.)	Uses HTTP POST method with a single endpoint
Endpoint Structure	Typically uses multiple endpoints for different resources	Uses a single flexible endpoint for all queries
Data Fetching	Fetches fixed data structures (predefined by the server)	Allows clients to request specific data structures as needed
Overfetching/Underfetching	Can suffer from overfetching (getting more data than needed) or underfetching (not getting enough data in a single request)	Solves overfetching/underfetching issues by allowing clients to request precisely the data they need

Reg

Response Format	Typically returns JSON or XML	Returns JSON by default, but the client specifies the format needed
Versioning	May use URL versioning or custom headers	Versioning is not necessary as clients specify the needed fields
Caching	Uses standard HTTP caching mechanisms	Requires custom caching strategies
Server Complexity	The server determines the structure of the response data	The client specifies the structure of the response data
Learning Curve	Generally well-known and easy to understand	May have a steeper learning curve due to the unique query language
Flexibility	May lack flexibility in data fetching	Highly flexible, allowing clients to define data requirements
Ecosystem	Wide range of tools and libraries available	Growing ecosystem with fewer tools and libraries compared to REST

REST Framework:
A REST framework is designed to build APIs following the principles of the REST architectural style.
It simplifies the creation of RESTful APIs using standard HTTP methods and resource-based URLs.
Popular REST frameworks include Django REST framework, Express.js, and Flask.

Diff between REST and Other Frameworks:
REST frameworks are API-centric, while other frameworks focus on building complete web applications.
REST frameworks adhere to REST principles, while other frameworks may use different architectural patterns.
RESTful APIs have statelessness and a uniform interface, whereas other frameworks may not follow these constraints.

GraphQL Query Mutation:
GraphQL is a query language for APIs, allowing clients to request precise data from the server.
Queries are used to fetch data, and clients define the structure of the response they need.
Mutations are used to modify data, allowing clients to specify changes they want to make on the server.

Structure:
Structure refers to the organization and arrangement of code, data, and components in an application.
A well-structured application is easier to understand, maintain, and scale.

Schema:
In the context of databases, a schema defines the structure and organization of data within the database.
It specifies tables, fields, data types, and constraints to ensure consistent data storage and retrieval.

Endpoints:
Endpoints are specific URLs in a web API used by clients to access different functionalities or resources.
Each endpoint corresponds to a specific operation or action in the API.

Network Calls:
Network calls refer to data exchange over a network, usually using protocols like HTTP.
In web services, clients initiate network calls to interact with the server and receive responses.

Node Express JavaScript:
Node.js Express is a fast, flexible, and minimalist web application framework for Node.js.
It simplifies building web apps by handling routes, middleware, and HTTP requests and responses.

Diff between REST and GraphQL:
REST follows a fixed data structure, while GraphQL allows clients to request custom data structures.
REST uses multiple endpoints, while GraphQL uses a single flexible endpoint for all queries.
GraphQL reduces overfetching and underfetching issues seen in RESTful APIs.

Protocols:
Protocols are sets of rules and conventions defining how components in web services communicate.
Key protocols include HTTP (for RESTful APIs), SOAP, and GraphQL (for query language).

GraphQL
GraphQL is a query language for APIs, developed by Facebook in 2012.
It allows clients to request exactly the data they need from the server, reducing overfetching and underfetching issues.
Clients define the structure of the response they want, improving the efficiency of data retrieval.
GraphQL uses a single flexible endpoint, which accepts all queries and mutations.
It has a strongly typed schema, making the API self-documented and improving communication between frontend and backend teams.
GraphQL supports real-time subscriptions, enabling real-time updates to data.
Advantages include reduced data transfer, efficient batching of requests, versionless APIs, and a rich ecosystem of tools and libraries.

WEB SERVICES U2

REST Framework:

A REST framework is designed to build APIs following the principles of the REST architectural style.

It simplifies the creation of RESTful APIs using standard HTTP methods and resource-based URLs.

Popular REST frameworks include Django REST framework, Express.js, and Flask.

Diff between REST and Other Frameworks:

REST frameworks are API-centric, while other frameworks focus on building complete web applications.

REST frameworks adhere to REST principles, while other frameworks may use different architectural patterns.

RESTful APIs have statelessness and a uniform interface, whereas other frameworks may not follow these constraints.

REST ARCH

REST is not an architecture; rather, it is a set of constraints that creates a software architectural style, which can be used for building distributed applications.

Constraints-

- Client-server
- Stateless
- Cacheable
- Uniform interface
- Layered system
- Code on demand

Restful Web Services

- RESTful web services are a type of web service that follows the
    - Representational State Transfer (REST) architectural style.
- REST is a set of principles that guide the design of web services.
- RESTful web services are
    - typically stateless, each request is independent of any previous requests
    - designed to be easy to use and efficient.
    - typically implemented using the HTTP protocol.
    - HTTP is a stateless protocol,
        - RESTful web services use the HTTP methods GET, POST, PUT, and DELETE to perform different operations on resources.
- RESTful web services use URIs (Uniform Resource Identifiers) to identify resources. URIs are unique identifiers for resources on the web. RESTful web services also use media types to specify the format of the data that is being exchanged.

API

- An API is an acronym for Application Programming Interface,

    - an interface that defines the interaction between different software components.
- Web APIs => what exactly request is made to the component.

IMP API METHODS

- GET
    - most common method for get some data from component.
    - returns some data from the API based on
        - the endpoint we hit
        - any parameter we pass.
- POST
    - creates the new records
    - updates the new created record in the database.
- PUT
    - takes the new records at the given URI.
    - If the record exists, update the record.
    - If record is not available, create a new record.
- PATCH
    - It takes one or more fields at the given URI.
    - used to update one or more data fields.
    - If the record exists, update the record.
    - If the record is not available, create a new record.
- DELETE
    - It deletes the records at the given URI.

Hypermedia as the Engine of Application State (HATEOAS): This principle suggests that the API responses should contain hyperlinks to related resources, allowing clients to navigate the API dynamically.

REST frameworks provide a structured way to implement these principles, handling HTTP request and response handling, routing, serialization, authentication, and other common tasks.

Some of the advantages of RESTful web services include:

- Scalability: RESTful web services are scalable, meaning that they can handle a large number of requests without any problems.
- Flexibility: RESTful web services are flexible, meaning that they can be used to create a wide variety of applications.
- Performance: RESTful web services are performant, meaning that they can respond to requests quickly.
- Ease of use: RESTful web services are easy to use, meaning that they are easy to develop, consume, and debug.

Java Webservices Java RS

- Java web services using

    - JAX-RS (Java API for RESTful Web Services)
- is a popular way to create and consume RESTful web services with Java.
- JAX-RS provides a simple and intuitive way to
    - define and implement REST resources,
    - and it includes a number of features that make it easy to develop
        - robust and scalable web services.

To create a Java web service using JAX-RS, you will need to:

- Create a Java interface that defines the web service resources.
- Implement the Java interface.
- Use annotations to map the resources to HTTP methods and URIs.
- Deploy the web service to a server.

Code of 'GET' Method

```java
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/hello")
public class HelloWorldService {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String sayHello() {
        return "Hello, world!";
    }
}
```

Code to get response -

```java
import java.net.URL;
import java.net.URLConnection;
import java.io.InputStream;
import java.io.BufferedReader;

public class HelloWorldClient {

   public static void main(String[] args) throws Exception {
      URL url = new URL("http://localhost:8080/hello");
      URLConnection connection = url.openConnection();
      InputStream inputStream = connection.getInputStream();
      BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));
      String response = reader.readLine();
      System.out.println(response);
   }
}
```

The @WebService and @Method annotations are used to define and implement RESTful web services in Jakarta RESTful Web Services (JAX-RS).

@WebService

The @WebService annotation is used to define a web service class. The @WebService annotation can be used to specify the following properties of the web service:

- The name of the web service
- The target namespace of the web service
- The WSDL document of the web service
- The endpoint interface of the web service

```
@WebService(
   name = "MyWebService",
   targetNamespace = "http://example.com/webservice",
   wsdlLocation = "WEB-INF/wsdl/MyWebService.wsdl"
)
public class MyWebService {
   public String sayHello(String name) {
      return "Hello, " + name + "!";
   }
}
```

@Method

The @Method annotation is used to define a web service method. The @Method annotation can be used to specify the following properties of the web service method:

- The HTTP method of the web service method
- The path of the web service method

Asynchronous Behavior Of Javascript

JavaScript is a single-threaded language, which means that it can only execute one task at a time. However, JavaScript also supports asynchronous programming, which allows it to execute multiple tasks simultaneously without blocking the main thread. This is achieved by using callbacks, promises, and async/await.

Callbacks

Callbacks are functions that are passed to other functions as arguments. The other function then calls the callback function when it is finished executing. Callbacks are a common way to handle asynchronous operations in JavaScript.

```
function readFile(fileName, callback) {
  // Read the file asynchronously
  fs.readFile(fileName, (err, data) => {
    if (err) {
      // Handle the error
      callback(err);
    } else {
      // The file was read successfully
      callback(null, data);
    }
  });
}

// Read the file "myfile.txt" and pass in a callback function
readFile("myfile.txt", (err, data) => {
  if (err) {
    // Handle the error
    console.log(err);
  } else {
    // The file was read successfully
    console.log(data);
  }
});
```

Promises

Promises are objects that represent the eventual completion or failure of an asynchronous operation. They can be used to chain together multiple asynchronous operations, and to handle errors.

```
function readFile(fileName) {
  // Return a promise that resolves when the fil
  return new Promise((resolve, reject) => {
    fs.readFile(fileName, (err, data) => {
      if (err) {
        reject(err);
      } else {
        resolve(data);
      }
    });
  });
}

// Read the file "myfile.txt" and use the promis
readFile("myfile.txt")
  .then((data) => {
    // The file was read successfully
    console.log(data);
  })
  .catch((err) => {
    // Handle the error
    console.log(err);
  });
```

Async/await

Async/await is a newer way to handle asynchronous operations in JavaScript. It allows you to write asynchronous code in a more synchronous style, which can make it easier to read and understand.

```
async function readFile(fileName) {
  // Read the file asynchronously
  const data = await fs.readFile(fileName);

  // Return the data
  return data;
}

// Read the file "myfile.txt" and use async/
const data = await readFile("myfile.txt");

// The file was read successfully
console.log(data);
```

NPM

- npm (Node Package Manager) is a package manager for the JavaScript programming language.
- It is the default package manager for the JavaScript runtime environment Node.js. npm allows developers to install, manage, and share JavaScript packages.

- npm packages are typically hosted on the npm registry,
    - which is a public repository of JavaScript packages.
    - Developers can also publish their own packages to the npm registry.

- To install a JavaScript package using npm, you can run the following command:

npm install package-name

This will install the package and all of its dependencies.

- To uninstall a JavaScript package, you can run the following command:

npm uninstall package-name

- This will uninstall the package and all of its dependencies.

Here are some of the benefits of using npm:

- Package management:
    - npm makes it easy to install, manage, and update JavaScript packages.
- Discoverability:
    - npm provides a central repository of JavaScript packages, making it easy to find packages that meet your needs.
- Community:
    - npm has a large and active community, which means that there are many resources available to help you get started and troubleshoot problems.

Node js

- Node.js is an open-source, cross-platform JavaScript runtime environment
    - that executes JavaScript code outside of a browser.
- It is built on Chrome's V8 JavaScript engine
- uses an event-driven,
- non-blocking I/O model that makes it
    - lightweight and efficient.
- fast, minimalistic, and flexible web application framework for Node.js.
- provides a set of robust features and tools to build web applications and APIs with ease.
- Used for a wide variety of applications,
    - including web development,
    - server-side programming,
    - networking,
    - Real-time applications.
    - popular for building microservices and APIs.

Some of the best features of Node.js include:

- Asynchronous and event-driven: Node.js is asynchronous and event-driven, which means that it can handle multiple requests simultaneously without blocking the main thread. This makes it ideal for building real-time applications and applications that need to handle a high volume of concurrent requests.
- Fast: Node.js is built on Chrome's V8 JavaScript engine, which is one of the fastest JavaScript engines available. This makes Node.js applications very fast and responsive.

- Scalable: Node.js applications are highly scalable. This means that they can handle a large number of users and requests without becoming unresponsive.
- Lightweight: Node.js is very lightweight and efficient. This means that it has a small footprint and can run on low-powered devices.
- Cross-platform: Node.js is cross-platform, which means that it can run on Windows, macOS, and Linux.
- Large community and ecosystem: Node.js has a large and active community. There are also many libraries and frameworks available for Node.js, which makes it easy to build complex applications.

request.js

```
// Importing the express module
const express = require('express');
const bodyParser = require('body-parser');
const url = require('url');
const querystring = require('querystring');

let app = express();
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());


// Sending the response for '/' path
app.get('/' , (req,res)=>{
    let page = req.query.name;
    let limit = req.query.age;
    let data = {'name':page,"age":limit};
// Sending the data json text
res.json(data);
})

// Setting up the server at port 3000
app.listen(4000 , ()=>{
   console.log("server running");
});
```

Server.js

```
// Importing the express module
const express = require('express');
const app = express();
const data = {
   portal: "TutorialsPoint",
   tagLine: "SIMPLY LEARNING",
   location: "Hyderabad"
```

```
}

// Sending the response for '/' path
app.get('/' , (req,res)=>{

// Sending the data json text
res.json(data);
})

// Setting up the server at port 3000
app.listen(4000 , ()=>{
   console.log("server running");
});
```

Connection with database (Mysql)
To connect Node.js to a MySQL database, you will need to install a MySQL driver for Node.js. The most popular MySQL driver is mysql.

To install the mysql driver, run the following command:

```
npm install mysql
```

Once the driver is installed, you can create a connection to the MySQL database using the following code:

```
const mysql = require('mysql');

const connection = mysql.createConnection({
  host: 'localhost',
  port: 3306,
  user: 'root',
  password: 'password',
  database: 'mydb'
});
```

Steps of Api calls in node js

```
npm install express
const express = require('express');
const app = express();
app.get('/users', (req, res) => {
  // Get all users from the database
  const users = [{ name: 'John Doe' }, { name: 'Jane Doe' }];

  // Send the users to the client
```

```
  res.send(users);
});
app.post('/users', (req, res) => {
  // Get the user data from the request body


  const user = req.body;

  // Save the user to the database
  // ...

  // Send a success response to the client
  res.status(201).send('User created successfully');
});
app.listen(3000, () => {
  console.log('Server listening on port 3000');
});
```

DJANGO REST FRAMEWORK

- Django Rest Framework (DRF) is a package built on the top of Django
    - create web APIs.
- provides the most extensive features of
    - Django,
    - Object Relational Mapper (ORM),
- which allows the interaction of databases in a Pythonic way.

- Hence the Python object can't be sent over the network, so we need to translate Django models into the other formats like JSON, XML, and vice-versa. This process is known as serialization, which the Django REST framework made super easy.

key components of the Django framework:

- Models:
    - handles the data layer.
    - developers to define the data models and relationships between them using Python classes.
    - These models are then translated into database tables, making it easy to work with the database without writing SQL queries directly.

- Views:
    - handle the business logic of the application.

    - receive requests from the user's browser, interact with the models to fetch data from the database, and then render templates to generate the HTML response to be sent back to the user.

- Templates:
    - responsible for generating the HTML presentation of the data.
    - They provide a way to separate the design from the business logic.
    - Django's template engine allows developers to embed Python code within the HTML templates, making it easy to dynamically generate content.
- URL Routing:
    - URL routing system that maps URLs to views.
    - The URLs are defined in a central URL configuration file, which makes it easy to organize and manage the different endpoints of the web application.

- Forms:
    - Django includes a powerful form handling system that simplifies the process of validating user input and processing form submissions.
    - This feature helps developers create and manage HTML forms in a more straightforward manner.

- Admin Interface:
    - automatic admin interface that allows developers to manage the application's data through a web-based interface without having to create a separate admin section manually.

- Middleware:
    - supports middleware, which allows developers to process requests and responses globally before they reach the view or after they leave the view.
    - This is useful for tasks like authentication, caching, and error handling.
- Security:
    - includes various built-in features to help developers protect against common web vulnerabilities, such as cross-site scripting (XSS), cross-site request forgery (CSRF), and SQL injection.

- Internationalization (i18n) and Localization (l10n):
    - Django provides tools to build applications that can be easily translated into multiple languages and localized for different regions.

common annotations used in JAX-WS:

@WebService:

- The @WebService annotation is used to mark a Java class as a web service. It indicates that the class contains web service operations that can be accessed remotely using SOAP (Simple Object Access Protocol) or other web service protocols.

- You can use this annotation to specify properties of the web service, such as the service name, port name, target namespace, and endpoint interface.

@WebMethod:

- The @WebMethod annotation is used to explicitly mark a Java method within a web service class as a web service operation. It's typically used when you want to customize the behavior of individual methods.
- This annotation can be used to specify properties of the web service method, such as the operation name.

@WebParam:

- The @WebParam annotation is used to customize the names and attributes of the parameters of a web service method. It allows you to specify additional information about the parameters that will be used in the generated WSDL (Web Services Description Language).

@WebResult:

- The @WebResult annotation is used to customize the return value of a web service method. It allows you to specify additional information about the method's return type, including its name, partName, and targetNamespace.

@WebFault:

- The @WebFault annotation is used to define a custom exception class for a web service. It allows you to create custom fault elements in the WSDL for specific exceptions thrown by the web service methods.

Here's an example of how these annotations can be used in a JAX-WS web service:

java

Copy code

```
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebFault;

@WebService
public class MyWebService {
   @WebMethod(operationName = "sayHello")
   @WebResult(name = "greeting")
   public String greet(@WebParam(name = "name") String name) {
      if (name == null || name.isEmpty()) {
         throw new IllegalArgumentException("Name cannot be empty");
      }
      return "Hello, " + name + "!";
   }
}
import javax.xml.ws.WebFault;

@WebFault(name = "CustomException", targetNamespace = "http://example.com/exceptions")
```

```
public class CustomException extends Exception {
    private String faultInfo;

    public CustomException(String message, String faultInfo) {
        super(message);
        this.faultInfo = faultInfo;
    }

    public String getFaultInfo() {
        return faultInfo;
    }
}
```

In this example, we use the @WebService, @WebMethod, @WebResult, @WebParam, and custom @WebFault annotations to define a web service class and its methods while customizing the behavior and structure of the web service. These annotations help define the contract and operation details of the web service.

JAX-RS annotations, followed by a single code example that demonstrates the use of these annotations in a single Java class:

@Path: Specifies the base URI path for a resource class or method.
@GET, @POST, @PUT, @DELETE, etc.: Annotate methods to indicate the HTTP method they handle.
@Produces: Specifies the media type of the response (e.g., JSON, XML).
@Consumes: Specifies the media type of the request.
@QueryParam, @PathParam, @FormParam: Used for handling query parameters, path parameters, and form parameters in a request.

```
import javax.ws.rs.*;
import javax.ws.rs.core.MediaType;

@Path("/books")
public class BookResource {

    // Example 1: Using @GET to handle HTTP GET requests
    @GET
    @Path("/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Book getBook(@PathParam("id") int id) {
```

```
        // Handle GET request to retrieve a book by ID and return it in JSON format
    }

    // Example 2: Using @POST to handle HTTP POST requests
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response createBook(Book book) {
        // Handle POST request to create a new book from JSON data
        return Response.status(Response.Status.CREATED).build();
    }

    // Example 3: Using @PUT to handle HTTP PUT requests
    @PUT
    @Path("/{id}")
    @Consumes(MediaType.APPLICATION_JSON)
    public Response updateBook(@PathParam("id") int id, Book book) {
        // Handle PUT request to update an existing book by ID using JSON data
        return Response.status(Response.Status.OK).build();
    }

    // Example 4: Using @DELETE to handle HTTP DELETE requests
    @DELETE
    @Path("/{id}")
    public Response deleteBook(@PathParam("id") int id) {
        // Handle DELETE request to delete a book by ID
        return Response.status(Response.Status.NO_CONTENT).build();
    }

    // Example 5: Using @QueryParam to handle query parameters
    @GET
    @Path("/search")
    @Produces(MediaType.APPLICATION_JSON)
    public List<Book> searchBooks(@QueryParam("title") String title) {
        // Handle GET request with a query parameter to search for books by title
    }
}

DJANGO CODE
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=100)
    published_date = models.DateField()
```

```
from rest_framework.decorators import api_view
from rest_framework.response import Response
from .models import Book
from .serializers import BookSerializer

@api_view(['GET', 'POST'])
def book_list(request):
    if request.method == 'GET':
        books = Book.objects.all()
        serializer = BookSerializer(books, many=True)
        return Response(serializer.data)
    elif request.method == 'POST':
        serializer = BookSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=201)
        return Response(serializer.errors, status=400)

@api_view(['GET', 'PUT', 'DELETE'])
def book_detail(request, pk):
    try:
        book = Book.objects.get(pk=pk)
    except Book.DoesNotExist:
        return Response(status=404)

    if request.method == 'GET':
        serializer = BookSerializer(book)
        return Response(serializer.data)
    elif request.method == 'PUT':
        serializer = BookSerializer(book, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=400)
    elif request.method == 'DELETE':
        book.delete()
        return Response(status=204)

from rest_framework import serializers
from .models import Book

class BookSerializer(serializers.ModelSerializer):
```

```
class Meta:
    model = Book
    fields = '__all__'
```

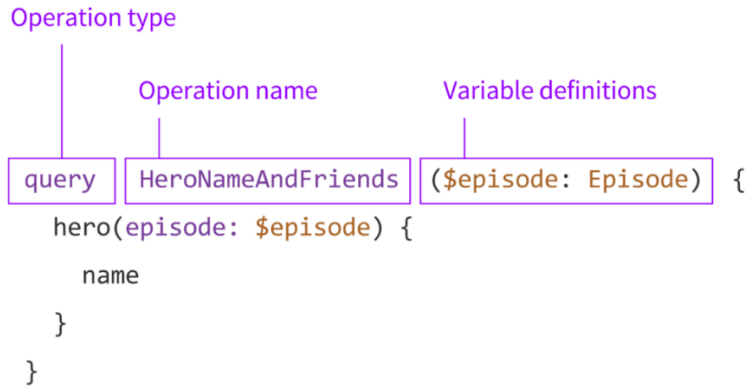
```
import javax.ws.rs.*;
import javax.ws.rs.core.MediaType;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;

@Path("/api")
public class UserService {

   private static final String DB_URL = "jdbc:mysql://localhost:3306/your_db";
   private static final String DB_USER = "your_username";
   private static final String DB_PASSWORD = "your_password";
```

```
    @POST
    @Path("/user")
    @Consumes(MediaType.APPLICATION_JSON)
    public Response addUser(User user) {
        try {
            Connection conn = DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD);
            String insertUserQuery = "INSERT INTO users (username, email) VALUES (?, ?)";
            PreparedStatement preparedStatement = conn.prepareStatement(insertUserQuery);
            preparedStatement.setString(1, user.getUsername());
            preparedStatement.setString(2, user.getEmail());
            preparedStatement.executeUpdate();
            conn.close();
            return Response.status(Response.Status.CREATED).build();
        } catch (Exception e) {
            return Response.status(Response.Status.INTERNAL_SERVER_ERROR).entity("Error: " + e.getMessage()).build();
        }
    }

    @POST
    @Path("/account")
    @Consumes(MediaType.APPLICATION_JSON)
    public Response addAccount(Account account) {
        try {
            Connection conn = DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD);
            String insertAccountQuery = "INSERT INTO accounts (account_number, balance) VALUES (?, ?)";
            PreparedStatement preparedStatement = conn.prepareStatement(insertAccountQuery);
            preparedStatement.setString(1, account.getAccountNumber());




            preparedStatement.setDouble(2, account.getBalance());
            preparedStatement.executeUpdate();
            conn.close();
            return Response.status(Response.Status.CREATED).build();
        } catch (Exception e) {
            return Response.status(Response.Status.INTERNAL_SERVER_ERROR).entity("Error: " + e.getMessage()).build();
        }
    }
}
```

WEB SERVICES U3

GraphQL

- Query lang
    - Query from not a particular type of dta store, but any num of sources
    - Requests - queries + mutations
- Server side runtime - typically over http



- GraphQL is a new API standard invented and developed by Facebook.
- It is an open-source server-side technology, now maintained by a large community of companies and individuals from all over the world.
- It is also an execution engine that works as a data query language and used to fetch declarative data.
- It was developed to optimise RESTful API calls and provides a flexible, robust, and more efficient alternative to REST.
- GraphQL is data query and manipulation language for your API and a server-side runtime for executing queries when you define a type system for your data.
- Unlike the REST APIs, a GraphQL server provides only a single endpoint and responds with the precise data that a client asked for.

Runtime

The runtime part points to the fact that a GraphQL API runs on a server.

In a GraphQL API, we shape our data within a strictly typed schema, telling it how to resolve data when asked for. To GraphQL, where data comes from doesn't matter. It could be from a database, micro-service, or even an underlying RESTful API.

Advantages

GraphQL has a number of advantages over other API technologies, such as REST:

- Efficiency: GraphQL queries can be very efficient, as clients can request exactly the data they need, and no more. This can lead to significant reductions in network traffic and bandwidth usage.
- Flexibility: GraphQL queries can be used to fetch data from multiple resources in a single request, and the results can be nested and filtered in a variety of ways. This makes it easy to build complex and dynamic Urls.

- Developer experience: GraphQL provides a number of tools and features that make it easy for developers to build and consume APIs. For example, GraphQL schemas are self-documenting, and there are a number of powerful developer tools available, such as GraphQL Explorer and Apollo Client.
- No over-fetching or under-fetching: With GraphQL, clients can request exactly the data they need, no more and no less. This avoids the problem of over-fetching, which can waste bandwidth and slow down applications.
- Hierarchical data structure: GraphQL returns data in a hierarchical structure, which makes it easy to parse and consume.
- Strongly typed: GraphQL is strongly typed, which means that the schema defines the types of data that can be returned. This helps to prevent errors and makes it easier to maintain code.
- Real-time updates: GraphQL supports real-time updates, so clients can be notified of changes to the data as they happen.
- Platform agnostic: GraphQL can be used with a variety of client and server platforms.

Graphql vs Rest

What is it?	REST is a set of rules that defines structured data exchange between a client and a server.	GraphQL is a query language, architecture style, and set of tools to create and manipulate APIs.
Best suited for	REST is good for simple data sources where resources are well defined.	GraphQL is good for large, complex, and interrelated data sources.
Data access	REST has multiple endpoints in the form of URLs to define resources.	GraphQL has a single URL endpoint.
Data returned	REST returns data in a fixed structure defined by the server.	GraphQL returns data in a flexible structure defined by the client.
How data is structured and defined	REST data is weakly typed. So the client must decide how to interpret the formatted data when it is returned.	GraphQL data is strongly typed. So the client receives data in predetermined and mutually understood formats.
Error checking	With REST, the client must check if the returned data is valid.	With GraphQL, invalid requests are typically rejected by schema structure. This results in an autogenerated error

Structure of GraphQL
The inner structure of GraphQL is composed of two main components:

Schema:

- The schema is a document that defines the data types and relationships that are available in the API.
- It is written in a special language called Schema Definition Language (SDL).

```
const { buildSchema } = require('graphql');

const schema = buildSchema(`
  type User {
    id: ID!
    name: String!
    email: String!
  }

  type Query {
    user(id: ID!): User
    users: [User!]!
  }

  type Mutation {
    createUser(name: String!, email: String!): User!
  }
`);
```

Resolvers:

- Resolvers are functions that are responsible for fetching and returning data based on the schema.
- They are typically implemented in a programming language such as JavaScript, Python, or Java.

- When a client sends a GraphQL query, the server first validates the query against the schema.
- If the query is valid, the server then executes the resolvers for the fields in the query.
- The resolvers return the data that is requested by the query, and the server then returns the data to the client.

```
const posts = [
  { id: 1, title: "First Post", content: "This is the content of the first post.", authorId: 101 },
  // ... more posts
];

const authors = [
```

```
  { id: 101, name: "John Doe", email: "john@example.com" },
  // ... more authors
];

const resolvers = {
  Query: {
    post: (parent, args) => {
      // Find the post by ID
      return posts.find(post => post.id === args.id);
    },
  },
  Post: {
    author: (parent) => {
      // Find the author of the post by authorId
      return authors.find(author => author.id === parent.authorId);
    },
  },
};
```

GraphQL works in a tree-like structure by using a nested query syntax. Nested queries allow clients to request data from multiple resources in a single request, and the results can be nested and filtered in a variety of ways

SDL

The Schema Definition Language (SDL) is a human-readable language used to define GraphQL schemas. It is a declarative language, which means that it describes the desired state of the schema rather than how to achieve it.

The SDL is composed of a number of different constructs, such as:

Types: Types define the different kinds of data that can be represented in the GraphQL schema.
Fields: Fields define the different properties that can be associated with a type.
Directives: Directives are optional modifiers that can be used to change the behavior of a type or field.
Here is an example of a simple GraphQL schema written in SDL:

```
type User {
  id: ID!
  name: String!
  email: String!
}
type Query {
  user(id: ID!): User
  users: [User!]!
}
```

This schema defines two types: User and Query. The User type has three fields: id, name, and email. The Query type has two fields: user and users.

Here are some of the benefits of using SDL:

- Human-readable: SDL is a human-readable language, which makes it easy to understand and maintain GraphQL schemas.
- Declarative: SDL is a declarative language, which means that it describes the desired state of the schema rather than how to achieve it. This makes it easier to write and modify schemas.
- Powerful: SDL is a powerful language that can be used to define complex GraphQL schemas.

Mutation

Mutations are used to modify data in the GraphQL API. They are typically defined in the Mutation type of the schema.

Here is an example of a mutation that can be used to create a new user:

```
type Mutation {
  createUser(name: String!, email: String!): User!
}
```

This mutation takes two arguments: name and email. It returns a new User object with the specified name and email address.

Query

Queries are used to fetch data from the GraphQL API. They are typically defined in the Query type of the schema.

Here is an example of a query that can be used to fetch all users:

```
type Query {
  users: [User!]!
}
```

This query returns a list of all users in the database.

Resolvers

Resolvers are functions that are responsible for fetching and returning data based on the schema. They are typically implemented in a programming language such as JavaScript, Python, or Java.

Here is an example of a resolver for the user field in the Query type:

```
const resolvers = {
  Query: {
    user: async (parent, args, context) => {
      const user = await context.db.users.findOne(args.id);
```

```
    return user;
   },
  },
};
```

This resolver fetches the user with the specified ID from the database and returns it.

Connecting to mongodb

To connect Node.js and MongoDB, you will need to:

- Install the MongoDB Node.js driver:

npm install mongodb

- Create a MongoDB connection string. This string will contain the hostname, port, and database name of your MongoDB server.

For example, if your MongoDB server is running on localhost on port 27017 and you want to connect to the database named my_database, your connection string would be:
mongodb://localhost:27017/my_database

- Create a new Node.js file and import the MongoDB driver:

const MongoClient = require('mongodb').MongoClient;

- Connect to MongoDB using the MongoClient class:

```
const MongoClient = require('mongodb').MongoClient;

const connectionString = 'mongodb://localhost:27017/my_database';

const client = new MongoClient(connectionString);

client.connect(function(err, db) {
  if (err) {
    console.log('Error connecting to MongoDB:', err);
    return;
  }

  console.log('Connected to MongoDB successfully!');

  const collection = db.collection('users');

  const document = {
    name: 'John Doe',
    email: 'john.doe@example.com',
  };

  collection.insertOne(document, function(err, result) {
    if (err) {
      console.log('Error inserting document into MongoDB:', err);
      return;
    }

    console.log('Document inserted into MongoDB successfully!');

    db.close();
  });
});
```

Apollo Client:
Apollo Client is a widely used JavaScript library for managing state and making requests to a GraphQL API from a client-side application. It's designed to work with various frontend frameworks like React, Angular, and Vue. Here are some key points about Apollo Client:

- State Management: Apollo Client provides a client-side state management system that allows developers to manage local application data and cache, making it easier to keep the UI in sync with data changes.
- Query and Mutation Execution: It simplifies the process of sending GraphQL queries and mutations to a server by providing a set of intuitive APIs for sending and caching requests. Apollo Client also supports real-time subscriptions.
- Optimistic UI: Apollo Client allows you to perform optimistic updates, where UI changes are made optimistically (before the server response) and rolled back if needed. This can provide a smoother user experience.
- Local State Management: In addition to managing data from a remote server, Apollo Client can also manage local application state, which is particularly useful when working with complex, data-driven applications.
- Powerful DevTools: Apollo Client offers developer tools that make it easier to inspect the GraphQL queries, monitor network activity, and debug issues.
- Ecosystem: There are various extensions and integrations available, such as Apollo Link, which allows you to customize and extend the client's behavior, and Apollo Server for mocking server responses during development.

Apollo Server:
Apollo Server is a community-driven, open-source GraphQL server implementation. It allows developers to build GraphQL APIs using JavaScript, Node.js, and other compatible languages. Here are some key points about Apollo Server:

- Schema-First Development: Apollo Server encourages a schema-first approach, where you define your GraphQL schema upfront, specifying types, queries, mutations, and subscriptions using the GraphQL Schema Definition Language (SDL).
- Resolvers: Apollo Server provides a straightforward way to define resolvers for your GraphQL operations. Resolvers are functions responsible for fetching data or executing logic in response to queries or mutations.
- Data Sources: It supports various data sources, including databases, REST APIs, and custom data connectors. You can use resolvers to connect to these data sources and fetch the required data.
- Real-Time Data: Apollo Server supports GraphQL subscriptions, enabling real-time updates and data synchronization between the server and clients using WebSockets or other transport mechanisms.
- Performance and Caching: Apollo Server provides features for caching and performance optimization, helping reduce the number of redundant database queries and making your API more efficient.
- Authentication and Authorization: You can integrate authentication and authorization logic into your resolvers to control access to specific parts of your API.

- Middleware and Plugins: Apollo Server allows you to add middleware and plugins to extend and customize the behavior of your server. This is useful for tasks like logging, error handling, and more.

Both Apollo Client and Apollo Server are widely adopted in the GraphQL community and offer comprehensive tools and features for building and consuming GraphQL APIs. They can be used together for end-to-end GraphQL application development.

NOSql vs SQL

Feature	SQL Databases	NoSQL Databases
Data Structure	Relational tables with fixed schemas	Flexible, schema-less or dynamic schemas
Query Language	SQL (Structured Query Language)	Varies by database (e.g., MongoDB Query Language)
Scalability	Typically vertically scalable (by adding more resources to a single server)	Horizontally scalable (by adding more servers to a distributed system)
Data Integrity	ACID (Atomicity, Consistency, Isolation, Durability) transactions for strong data integrity	BASE (Basically Available, Soft state, Eventually consistent) for eventual consistency and high availability
Complex Queries	Well-suited for complex joins and aggregations	May not support complex queries as efficiently, but optimized for simple queries on large datasets

Reliability	Typically high reliability, often used for critical systems	Reliability varies by NoSQL database and configuration
Use Cases	Traditional business applications, e-commerce, finance	Big data, real-time applications, content management, and situations where data structures evolve rapidly
Joins	Supports complex table joins and relational data modeling	Usually limited or less efficient for complex joins
Transactions	Supports ACID transactions for strong data consistency	Limited support for transactions; focus on partition tolerance and high availability
Horizontal Scaling	More challenging and often requires sharding	Designed for horizontal scaling from the outset
Examples	MySQL, PostgreSQL, Oracle, SQL Server, etc.	MongoDB, Cassandra, Redis, CouchDB, etc.

MongoDB is a popular open-source NoSQL database management system that is designed for flexibility, scalability, and ease of development. It falls under the category of document-oriented databases and is known for its ability to handle unstructured or semi-structured data, making it a popular choice for a wide range of applications. Here are some key aspects of MongoDB:

Document-Oriented Database: MongoDB stores data in a format known as BSON (Binary JSON), which is a binary-encoded serialization of JSON-like documents. Each document is a self-contained unit that can contain various data types, including strings, numbers, arrays, and subdocuments.

Schema Flexibility: MongoDB is schema-less, which means you can store documents with different structures in the same collection. This flexibility is particularly useful in applications where data models are constantly evolving.

Collections and Documents: Data in MongoDB is organized into collections, which are analogous to tables in relational databases. Each collection contains one or more documents, which are analogous to rows or records in SQL databases.

Rich Query Language: MongoDB provides a powerful and expressive query language for querying data, including support for filtering, sorting, and aggregation operations.

Indexes: MongoDB supports the creation of indexes to improve query performance. Indexes can be created on any field within a document.

Horizontal Scalability: MongoDB is designed to scale horizontally, which means it can handle large volumes of data and high traffic by distributing data across multiple servers. This makes it suitable for applications that require high availability and scalability.

Replication: MongoDB supports replica sets, which provide data redundancy and fault tolerance. Replica sets consist of primary and secondary nodes, with automatic failover in case the primary node goes down.

Sharding: MongoDB offers sharding, a technique for distributing data across multiple servers to balance the load and ensure efficient storage and retrieval of data.

Geospatial Queries: MongoDB has built-in support for geospatial queries, making it suitable for applications that require location-based data and services.

Aggregation Framework: MongoDB includes a flexible and powerful aggregation framework for performing complex data transformations and calculations.

Official Drivers and Ecosystem: MongoDB provides official drivers and libraries for various programming languages, making it accessible and widely used in different development environments.

Community and Enterprise Editions: MongoDB is available in both open-source community editions and commercial enterprise editions. The enterprise edition offers additional features, such as advanced security and support.

Document Validation: MongoDB allows you to define validation rules for documents to ensure data integrity and consistency.

MongoDB is commonly used in a wide range of applications, including content management systems, e-commerce platforms, social networks, IoT applications, and real-time analytics. Its ease of use, scalability, and flexibility make it a valuable tool for developers working with diverse data needs.

WEB SERVICE U4

WCF - Windows Communication Foundation

- Framework - building + developing service - oriented apps in microsoft
- Create and expose services + interact w them using protocols -> HTTP, TCP, named pipes
- Send data as async messages -> from one service endpoint to another
- E.p ->
    - Continuously available service hosted by IIS (Internet Info Services)
    - Service hosted in an app
    - Can be of a client of a service -> req data from service endpoint
- Messages sent - simple as XML or complex - stream of data
    - Secure service to process business transnc
    - Services supplies curr data -> others, traffic report etc
    - Chat service - exchange data in real time
    - Dashboard app - polls one or more services for data, present in logical presentation
    - Expose workflow implementation -> use wcf as a service
- Creating such apps was possible before BUT w wcf -> dev of endpoints easier
- Conclusion -> wcf offers manageable approach to creating web services + web service clients

Grpc as alternative

- Modern rpc (remote program calls) framework
- Popular alt to wcf
- Bult on top of http/2
- Advantages:
    - Performance: more efficient, esp: long running connections
    - Scalability: scale to large num of clients and servers
    - Security: variety; TLS and auth
    - Cross-platform: platform neutral, used w variety of prog langs

FEATURES OF WCF

1. Service Orientation:
    - Create service oriented apps
    - SOArch - reliance of web services to send and receive data
    - Loosely coupled -> any client created on any platform can connect to any service as long as essential contracts are met
    - not hard coded'
2. Interoperability:
    - Implements modern ind standards
3. Multiple Message patterns
    - Exchanged in one of several patterns
    - Most common - request/reply
    - One way message -> single endpoint sends msg without any expectation of a reply

    - Duplex exchange - two eps estb connection, send data back and forth
4. Service Metadata
    - Supports publishing metadata using ind standard formats - WSDL, XML Schema
    - M.d -> used to auto generate + config clients for accessing wcf services
    - Published over http or https
5. Data contracts
    - Wcf built using .net, code friendly methods of supplying contracts to enforce
    - Universal type -> data contract
    - Handle data by creating classes -> rep data entity w props
    - Comprehensive sys -> working w data in this easy manner
    - Create class to rep data, service automatically gens metadata user must comply w
6. Security
    - Msgs can be encrypted to protect privac
    - Users - auth
    - Ssl or ws-secureconversation
7. Multiple Transports and Encoding
    - Msgs sent on any several transport protocols + encodigns
    - Most common - soap msgs -> HTTP
    - Or tcp, named pipes, msmq
    - Encoded as text/binary format
    - Binary - MTOM standard
    - None are good enough -> create own transport and encoding
8. Reliable and queued messages
    - Using reliable sessions implemented over WS-reliable messaging and using msmq
9. Durable Messages
    - Durable msg - one that is never lost due to disruption in comm
    - Always saved to db
    - If disruption, db allows to resume exchange
10. Transactions
    - 3 models -
        - WS-AtomicTransactions
        - APIs in System.Transactions namespace
        - Microsoft Distribute upd Transaction Coordinator
11. AJAX and Rest Support
    - REST -> Web 2.0 technology
    - WCF can be configured to process "plain" XML data that is not wrapped in a SOAP envelope
    - WCF can also be extended to support specific XML formats,
        - ATOM (a popular RSS standard),
        - even non-XML formats, such as JavaScript Object Notation (JSON).

12. Extensibility
    - Number of points
    - If extra capability -> nm of entry points, customize behaviour of service

Fundamentals

Key concepts + how they fit together

- WCF infra-> runtime + set of APIs
    - APIs -> creating sys - send msgs between services & clients
- Create apps to comm w apps on same comp sys
- Or sys resides in another company; accessed over the internet

1. Messaging and endpoints
    a. Wcf based on notion of **message based comm**
    b. Anything that can be a message - http req etc - can be rep in uniform way
    c. Enables unified api across diff transport mechanisms
    d. Model distinguishes between clients and services
        i. CLIENTS => apps that initiate comm,
        ii. SERVICES => apps wait for client comm, respond acc.
    e. Single app - can be both - duplex services, peer to peer networking
    f. Messages - sent between endpoints
    g. Endpoints : places where msgs are sent or received or both
        i. Define ALL info neede for msg exchange
        ii. Desc in standard way where msgs
            1. should be sent,
            2. How they shold be sent
            3. What msgs look like
    h. Services - exxpose one or more app endpoints, client gens e.p compatible w that of service
    i. Service - expose metadata - clients can process to gen appr. WCF clients and comm stacks
2. Communication Protocols
    a. Req ele of comm stack - transport protocol
        i. Msgs can be sent over intranets and internet using common transports - http tcp etc
        ii. Other -> support comm w Msg queuing apps and nodes on peet networking mesh
        iii. More can be added by using built in extension points
    b. Another req ele -> encoding
        i. Specifies how any given msg is formatted
        ii. Wcf provides:

1. Text encoding, interoparable
2. Message Transmission Optimization Mechanism (MTOM) encoding - efficiently send unstructured binary data to and from a service
3. Binary encoding - efficient transfer

iii. More can be added using built in extension points

3. Message Patterns
    a. Several patterns
    b. Request - reply, one way, duplex
    c. Diff transport -> diff messaging patterns - affect types of interactions they support
    d. Wcf apis + runtime -> help to send messages securely and reliably

WCF Terms

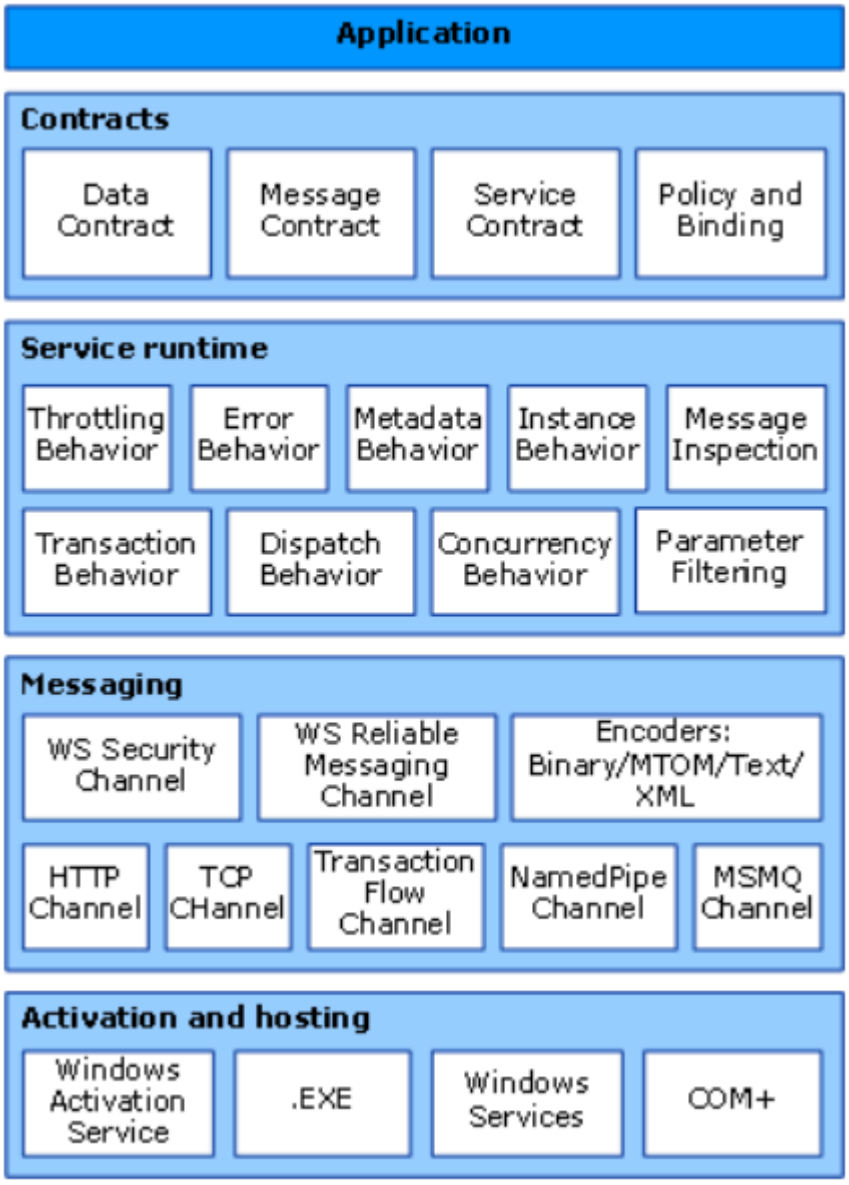
1. Message
    a. Self contained
    b. Unit of data
    c. Consist of several parts like body,header etc
2. Service
    a. A construct that exposes one or more endpoints
    b. Each endpoints- exposes one or more service operations
3. Endpoint
    a. Construct at which msgs are sent or received or both
    b. Comprises -
        i. Location - address - where msgs can be sent
        ii. Specification of comm mechanism - desc how msgs should be sent
        iii. Defn for a set of msgs - can be sent or received or both - desc what msg can be sent - service contract
4. Application endpoint
    a. E.p exposed by app
    b. Corresponds to service contract implemented by app
5. Infrastructure endpoint
    a. Exposed by infrastructure - facilitate functionality
        i. Needed or provided by service - does not relate to service contract
6. Address
    a. Where msgs are received
    b. Specified as a URi -uniform resource identifier
    c. Uri schema part -> names transport mechanism to use to reach the addr -> http and tcp

    d. Uri hierarchical part -> unique location
    e. E.p address - create unique ep add for each ep or share across eps
7. Binding
    a. Defines how ep communicated to world
    b. Constructed of a set of components called binding eles -> stack on top of the other -> create comm infrastructure
    c. Defines transport and encoding
    d. Can contain eles, -> details -> security mechanisms, message patterns
8. Binding ele
    a. Reps particular piece of binding
        i. Transport
        ii. Encoding
        iii. Implementationof infra level protocol
        iv. Any other comp of comm stack
9. Behaviour
    a. Controls various run time aspects
    b. Grouped acc to scope
        i. Common - affect all eps globally
        ii. Service - only service related aspects
        iii. Endpoint - only ep
        iv. Operation - only particular operations
10. System provided bindings
    a. Number of these
    b. Collections of binding eles optimized for specific scenarios
    c. Save time - present only applicable opts
11. Configuration versus coding
    a. Control of app can be w config or code or combination
    b. Config - allow someone other than dev to set client and service params after code is written; no need to recompile; add eps, bindings, behaviours
    c. Coding - allow dev to restrict strict control over all comps of service or client + any settings done through the config can be inspected + if needed, overridden
12. Service Operation
    a. Procedure - implements functionality for an op
    b. Op is exposed to clients as methods
    c. Can return val and take argos or no argos or no return
13. Service Contract
    a. Ties together multiple related operations into a single functional unit.
    b. define service-level settings,

        i. as the namespace of the service,
        ii. a corresponding callback contract,
        iii. and other such settings.
    c. contract is defined by creating an interface in the programming language of your choice and applying the ServiceContractAttribute attribute to the interface.
    d. The actual service code results by implementing the interface.
14. Operation contract
    a. defines the parameters and return type of an operation
    b. interface that defines the service contract -> signify an operation contract by applying the OperationContractAttribute attribute to each method definition that is part of the contract.
15. Message contract
    a. Desc format of msg
16. Fault contract
    a. Assc w service operation -> denote eros that can be returned to the caller
    b. Op can have zero or more faults assc w it
17. Data contract
    a. Desc in metadata of data types a service uses
    b. enable s others to interoperate w service
18. Hosting
    a. Service must be hosted in some process
    b. Host - app that controls the lifetime of the service
    c. Can be self hosted or managed by existing process
19. Self hosted service
    a. Runs within a process app dev created
    b. Dev 0 controls lifetime, sets props, opens and closes service
20. Hosting process
    a. App designed to host services
    b. IIS, WAS , Windows services
    c. Host controls all that ^^^
21. Instancing
    a. Service has instancing models
    b. Three types:
        i. Single - one clr object services all client
        ii. Per call - new clr obj is created to handle each call
        iii. Per session - set of clr are created for each sesh
    c. choic e - app requirements + expected usage pattern of service
22. Client app

    a. Prog -> exchanges messages w one or more endpoints
    b. Begins by creating instance of wcf client and calling methods
23. Channel
    a. Concrete implementation of a binding ele
    b. Binding - reps config
    c. Channel - implementation assc w that config
    d. Stack on top of one other to create channel stack
24. WCF client
    a. Client app construct - exposes service ops as methods
25. Metadata
    a. Chars of service that extends entity needs to understand to comm
    b. The metadata exposed by the service includes XML schema documents, which define the data contract of the service, and WSDL documents, which describe the methods of the service.
26. Security
    a. includes confidentiality (encryption of messages to prevent eavesdropping)
    b. integrity (the means for detection of tampering with the message)
    c. authentication (the means for validation of servers and clients)
    d. authorization (the control of access to resources).
27. Transport sec mode
    a. Specifies that confidentiality, integrity, and authentication are provided by the transport layer mechanisms (such as HTTPS)
    b. Sus to man in the middle
28. Message sec mode
    a. security is provided by implementing one or more of the security specifications, such as the specification named Web Services Security: SOAP Message Security
    b. message contains the necessary mechanisms to provide security during its transit, and to enable the receivers to detect tampering and to decrypt the messages.
29. Transport w message credential sec mode
    a. Specifies use of trans layer to provide cia of msgs
    b. Each ,sg can contain multiple credentials (claims) -> req by receivers of msg
30. WS-*
    a. Shorthand for growing set of Web Service specifications
        i. Ws sec
        ii. Ws reliable msging

WCF ARCH



Contracts and Descriptions:

- Data Contract: Defines the parameters of messages using XML Schema (XSD) documents.
- Message Contract: Specifies message parts using SOAP protocols, providing fine-grained control over message structure.

- Service Contract: Defines method signatures distributed as interfaces in supported programming languages (e.g., C# or Visual Basic).
- Policies and Bindings: Stipulate conditions required for communicating with a service, including transport, encoding, and security requirements.

Service Runtime:

- Throttling: Controls message processing rates based on preset limits.
- Error Behavior: Specifies actions when internal errors occur, considering security implications.
- Metadata Behavior: Governs the availability and accessibility of service metadata.
- Instance Behavior: Defines how many service instances can run (e.g., singleton for one instance).
- Transaction Behavior: Enables the rollback of transacted operations in case of failure.
- Dispatch Behavior: Controls how messages are processed by the WCF infrastructure.
- Extensibility: Allows customization of runtime processes, including message inspection and parameter filtering.

Messaging:

- Channels: Components that process messages, with transport channels for reading and writing messages from the network, and protocol channels for implementing message processing protocols.
- Transport Channels: Deal with network communication and use encoders to convert messages to and from network-compatible representations (e.g., HTTP, TCP).
- Protocol Channels: Implement message processing protocols (e.g., WS-Security, WS-Reliability).

Specific Messaging Components:

- WS-Security: Enables security at the message layer.
- WS-Reliable Messaging: Ensures guaranteed message delivery.
- Encoders: Provide various encodings for message content.
- HTTP Channel: Uses HTTP for message delivery.
- TCP Channel: Specifies the use of the TCP protocol.
- Transaction Flow Channel: Manages transacted message patterns.
- Named Pipe Channel: Enables interprocess communication.
- MSMQ Channel: Facilitates interoperation with MSMQ applications.

Hosting and Activation:

- Self-hosted Service: Run as an executable program.

- Hosted Services: Managed by external agents such as IIS or Windows Activation Service (WAS).

- Windows Service: Automatically run as a Windows service.
- COM+ Components: Can also be hosted as WCF services.

MVC ARCH

- MVC is an architectural/design pattern that separates an application into three main components: Model, View, and Controller.
- It isolates business logic, UI logic, and input logic, making applications more scalable and extensible.
- Originally used for GUIs, it's now a standard web development framework for building web and mobile applications.

Components of MVC:

Model:

- The Model component in MVC represents the application's data and business logic. It encompasses various aspects:
    - Data Handling: The Model is responsible for managing data, which can include data storage, retrieval, and manipulation. This can involve interactions with databases, external APIs, or any data source.
    - Business Logic: It encapsulates the core business rules and operations of the application. This logic can range from complex algorithms to simple data validations.
    - Data Transfer: The Model can handle data transfer between the View and the Controller, ensuring that the appropriate data is available for rendering in the View or for processing in the Controller.
    - Independence: One of the key features of the Model is its independence from the View and Controller. It operates without direct knowledge of the user interface or user interactions.

View:

- The View component is responsible for the presentation and user interface of the application. It involves several important aspects:
    - User Interface Elements: The View defines the elements that users see and interact with, such as web pages, forms, buttons, images, and any visual components.
    - Data Presentation: It takes data from the Model (via the Controller) and presents it to users in a format they can understand. This includes rendering data as HTML for web applications, graphical elements, and textual content.

    - User Interaction: The View may handle user interactions, such as accepting user input and passing it to the Controller for further processing.
    - Independence: Similar to the Model, the View should remain independent of the business logic and data processing. It focuses on the presentation and user experience.

Controller:

- The Controller is the component responsible for managing user interactions, routing requests, and coordinating actions. It has several key functions:
    - Request Handling: The Controller receives user requests and initiates actions based on those requests. For example, it determines which Model methods to invoke and which View to render.
    - Business Logic Coordination: It coordinates the interactions between the Model and the View. The Controller retrieves data from the Model, processes it, and passes the results to the View for presentation.
    - User Input Handling: The Controller manages user input, validating and processing it before interacting with the Model. It enforces the business rules and handles errors.
    - Independence: The Controller remains independent of the specifics of the user interface. It's concerned with managing the flow of the application and ensuring the right Model and View components are connected.

How MVC Works:

- Example: A user requests a list of students in a class.
- Controller sends the request to the student model.
- Model retrieves data from the database and returns it to the controller.
- If successful, the controller instructs the student view to render the list as HTML.
- Controller sends the HTML response to the user.
- If an error occurs, the controller handles it by instructing the error view to render an error message.

Advantages of MVC:

- Code maintainability and extensibility.
- Separation allows individual testing of components.
- Simultaneous development of components.
- Reduces complexity by dividing the application.
- Supports Test-Driven Development (TDD).
- Suitable for large web development teams.

Disadvantages of MVC:

- May be challenging to read, change, test, and reuse.
- Less suitable for small applications.
- Inefficiency in data access from the view.
- Introduces complexity and abstraction layers.

Popular MVC Frameworks:

- Ruby on Rails, Django, Spring MVC, Laravel, and others.

Notes:

- MVC is an architectural pattern dividing applications into Model, View, and Controller.
- Controller handles requests and business logic, while View manages the UI.
- Model deals with data and interacts with databases.
- MVC supports modularity, testing, and scalability but can be complex.
- Popular MVC frameworks include Ruby on Rails, Django, and Laravel.

Quality of Service

Quality of Service (QoS) in the context of web services refers to the measures and strategies employed to ensure that web services operate at a certain level of performance, reliability, and availability. QoS for web services involves managing various aspects of their operation to meet service-level agreements (SLAs) and to provide a consistent and satisfactory user experience.

QoS Attribute	Description
Scalability	The throughput improvement rate in a given time interval
Latency	The time taken to start servicing a requested service
Throughput	A service request processing rate
Response time	The time taken between the end of service or demand of a service and the beginning of a response.
Capacity	The number of parallel requests a service allows
Availability	A service is operating time percentage
Reliability	The time for continuity of expected service and for transition to accurate state
Accuracy	The service error rate in a specified time interval.
Robustness	The service flexibility level to inappropriate access and invocation services
Stability	The change rate of service
Cost	The total cost to utilize the service
Security	Specifies the methodologies to protect data
Reliable messaging	Determines if the service offers mechanisms to guarantee reliable message delivery
Integrity	Determines the transactional properties support to the services
Interoperability	Defines if the service is acquiescent with interoperability profiles

C# CODE

using c# webservices add following data

user

account

Model

```
public class User
{
    public int UserId { get; set; }
    public string UserName { get; set; }
    // Add other user properties here
}

public class Account
{
    public int AccountId { get; set; }
    public string AccountName { get; set; }
    public decimal Balance { get; set; }
    // Add other account properties here
}
```

Controller

```
[Route("api/users")]
public class UserController : ApiController
{
    // Implement user-related methods here
}
```

```
[Route("api/accounts")]
public class AccountController : ApiController
{
    // Implement account-related methods here
}
Action
public class UserController : ApiController
{
    private List<User> users = new List<User>();

    [HttpPost]
    public IHttpActionResult AddUser(User user)
    {
        // You can validate and add the user to your data store here
        users.Add(user);
        return Ok(user);
    }

    // Implement other user-related actions as needed
}

public class AccountController : ApiController
{
    private List<Account> accounts = new List<Account>();

    [HttpPost]
    public IHttpActionResult AddAccount(Account account)
    {
```

```
        // You can validate and add the account to your data store here
        accounts.Add(account);
        return Ok(account);
    }

    // Implement other account-related actions as needed
}
```

With Db Connectivity

DB Structure

```
CREATE TABLE Users (
    UserId INT PRIMARY KEY IDENTITY,
    UserName NVARCHAR(50) NOT NULL,
    Email NVARCHAR(100) NOT NULL
);

CREATE TABLE Accounts (
    AccountId INT PRIMARY KEY IDENTITY,
    UserId INT,
    AccountName NVARCHAR(50) NOT NULL,
    Balance DECIMAL(18, 2) NOT NULL,
    FOREIGN KEY (UserId) REFERENCES Users(UserId)
);
```

APIs

```
[Route("api/users")]
[ApiController]
public class UsersController : ControllerBase
{
    private readonly YourDbContext _context;

    public UsersController(YourDbContext context)
    {
        _context = context;
    }

    [HttpPost]
    public IActionResult AddUser(User user)
    {
        _context.Users.Add(user);
        _context.SaveChanges();
        return Ok();
    }
}

[Route("api/accounts")]
[ApiController]
public class AccountsController : ControllerBase
{
    private readonly YourDbContext _context;

    public AccountsController(YourDbContext context)
```

```
    {
        _context = context;
    }

    [HttpPost]
    public IActionResult AddAccount(Account account)
    {
        _context.Accounts.Add(account);
        _context.SaveChanges();
        return Ok();
    }
}
```

WCF C# CODE

Here, I'll provide a simple example of creating a WCF service and consuming it in a C# application. In this example, I'll create a basic "Calculator" service that has two methods for addition and subtraction.

Step 1: Create a WCF Service

Open Visual Studio.

Create a new project: "WCFService" of type "WCF Service Application."

In the IService.cs file, define the service contract and its methods.

csharp

Copy code

```
using System.ServiceModel;

[ServiceContract]
public interface ICalculator
{
    [OperationContract]
    double Add(double num1, double num2);

    [OperationContract]
    double Subtract(double num1, double num2);
}
```

Implement the service in the Service1.svc.cs file:

csharp

Copy code

```
public class CalculatorService : ICalculator
{
    public double Add(double num1, double num2)
    {
        return num1 + num2;
    }

    public double Subtract(double num1, double num2)
    {
        return num1 - num2;
    }
}
```

Step 2: Host the WCF Service

Open the "Service1.svc" file.

In the code-behind, add the ServiceHost for the service:

csharp

Copy code

```
<% @ServiceHost Language="C#" Debug="true" Service="WCFService.CalculatorService"
CodeBehind="CalculatorService.svc.cs" %>
```

Step 3: Create a Client Application to Consume the Service

Create a new C# Console Application in Visual Studio: "WCFClient."

Right-click the project, and choose "Add Service Reference."

Enter the URL of your hosted WCF service (e.g., http://localhost:12345/Service1.svc).

Click "Go," and then "OK" to generate the proxy classes.

Step 4: Consume the WCF Service in the Client Application

Now, you can use the generated client proxy to consume the WCF service in your C# client application.

csharp

Copy code

```
using System;

namespace WCFClient
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create a proxy for the service
            CalculatorServiceClient client = new CalculatorServiceClient();

            // Call the service methods
```

```
            double num1 = 10;
            double num2 = 5;
            double resultAdd = client.Add(num1, num2);
            double resultSubtract = client.Subtract(num1, num2);

            // Display the results
            Console.WriteLine($"Addition Result: {resultAdd}");
            Console.WriteLine($"Subtraction Result: {resultSubtract}");

            // Close the client
            client.Close();
        }
    }
}
```

Make sure to replace CalculatorServiceClient with the actual name generated in your service reference.

Step 5: Host and Run the Service

- Build and run the WCF service in Visual Studio.
- Run the client application, and it will communicate with the hosted service to perform addition and subtraction.

This is a basic example to get you started with WCF. In practice, you may need to handle exceptions, security, and various configurations based on your specific requirements.

Firebase is a popular cloud-based platform provided by Google for building and running web and mobile applications. It offers a wide range of services and tools that help developers quickly and easily develop high-quality applications. Firebase is often used for building real-time, scalable, and serverless applications. Below, I'll detail some of the core services and features offered by Firebase:

Authentication:

- Firebase Authentication provides a secure way to authenticate users in your application. It supports various authentication methods, including email/password, social authentication (e.g., Google, Facebook, Twitter), phone number verification, and custom authentication.
- It handles user registration, login, and management, helping you secure your app's resources and data.

Realtime Database:

- Firebase Realtime Database is a NoSQL, JSON-based database that allows you to build real-time applications. It supports real-time data synchronization, meaning changes made on one client are immediately reflected on all other connected clients.

- The database is often used for chat applications, live collaboration tools, and other apps that require real-time updates.

Firestore:

- Firestore is Firebase's newer, more scalable NoSQL database that offers powerful querying and automatic scaling. It's designed for building complex, scalable applications.
- Firestore supports offline data access, real-time synchronization, and is ideal for applications that require complex data modeling and querying.

Storage:

- Firebase Storage provides a cloud-based storage solution for storing and serving user-generated content, such as images, videos, and files.
- It's a cost-effective way to manage user-generated content in your applications while offering secure access controls.

Cloud Functions:

- Firebase Cloud Functions allows you to write serverless functions that can respond to various Firebase and HTTP events. You can use it for tasks like sending notifications, data processing, and integrating with third-party services.
    - It integrates seamlessly with other Firebase services.

Hosting:

- Firebase Hosting is a web hosting service that allows you to deploy your web applications and static assets quickly and securely. It offers features like automatic SSL, CDN delivery, and easy rollback of deployments.
- It's an ideal solution for hosting single-page web apps, progressive web apps (PWAs), and static websites.

Authentication and Authorization:

- Firebase provides security rules and access control, allowing you to control who has access to your data and resources. You can define fine-grained rules for data read and write operations.
- Firebase Authentication and Authorization can work together to secure your application's backend.

Machine Learning (ML) Kit:

- Firebase ML Kit allows you to add machine learning capabilities to your app with minimal effort. It offers features like text recognition, image labeling, face detection, and more.
- It's used to build applications with intelligent features that can understand and process data.

Cloud Messaging:

- Firebase Cloud Messaging (FCM) is a messaging platform for sending notifications to iOS, Android, and web applications. It supports targeted messaging, topic-based messaging, and real-time delivery.

Performance Monitoring and Crash Reporting:

- Firebase Performance Monitoring allows you to monitor the performance of your app, identify bottlenecks, and optimize user experiences.

- Firebase Crashlytics provides detailed crash reporting, helping you quickly identify and resolve issues in your application.

Analytics:

- Firebase Analytics provides insights into user behavior and app usage. It allows you to track user engagement, retention, and conversion rates.
- It helps you make data-driven decisions to improve your application.

Remote Config:

- Firebase Remote Config allows you to remotely configure your app without requiring an app update. You can personalize user experiences, perform A/B testing, and change app behavior on the fly.

A/B Testing:

- Firebase A/B Testing enables you to run experiments to determine which app variants perform best. You can test different features, UI elements, and user flows.

Dynamic Links:

- Firebase Dynamic Links are smart URLs that allow you to send users to the right place within your app. They work across iOS, Android, and the web and provide a seamless user experience.

App Distribution:

- Firebase App Distribution simplifies the process of distributing pre-release versions of your app to testers and stakeholders. You can gather feedback and track app distribution.

Firebase offers a comprehensive suite of tools for building, deploying, and maintaining modern web and mobile applications. It's known for its ease of use, scalability, and the ability to get apps to market quickly. Firebase can be used for a wide range of applications, from simple single-page websites to complex, real-time, data-driven mobile applications.