Full Stack Application Development

BITS Pilani

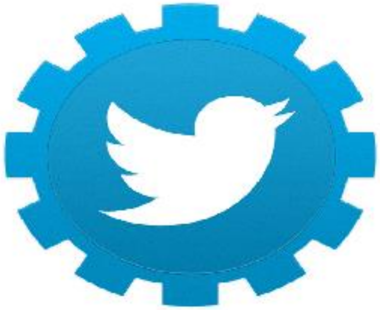
BITS Pilani
Pilani | Dubai | Goa | Hyderabad

API and its management

API

API stands for ‘Application Programming Interface'

- Technically,
  - ✓ an API describes how to connect a dataset or business process with some sort of consumer application or another business process
- We are probably familiar with a lot of the big names that use APIs all the time
- For example,
- whenever use Facebook account to join another site, login request is being routed via an API
- whenever use the Share functions of an application on your mobile device, those apps are using APIs to connect you to Twitter, Instagram, etc.

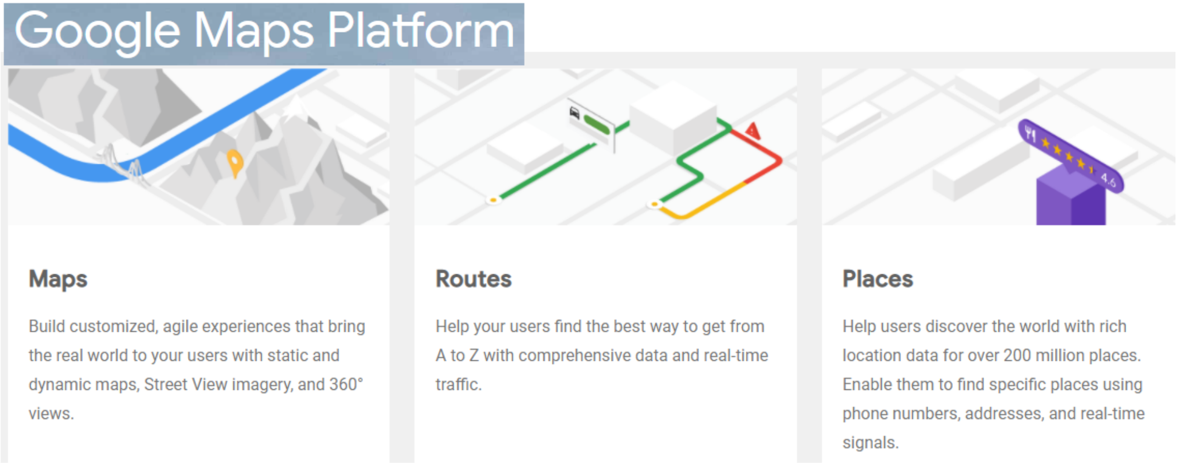


Mapping API

Google Maps

- When you search for an address, an API helps interact with a map database to identify the latitude and longitude, and other related data, for that address

- The API also makes it possible for a mapping interface to then display
  - ✓ the address on the map
  - ✓ any additional information such as the directions to that destination



Google Maps Platform

Maps

Build customized, agile experiences that bring the real world to your users with static and dynamic maps, Street View imagery, and 360° views.

Routes

Help your users find the best way to get from A to Z with comprehensive data and real-time traffic.

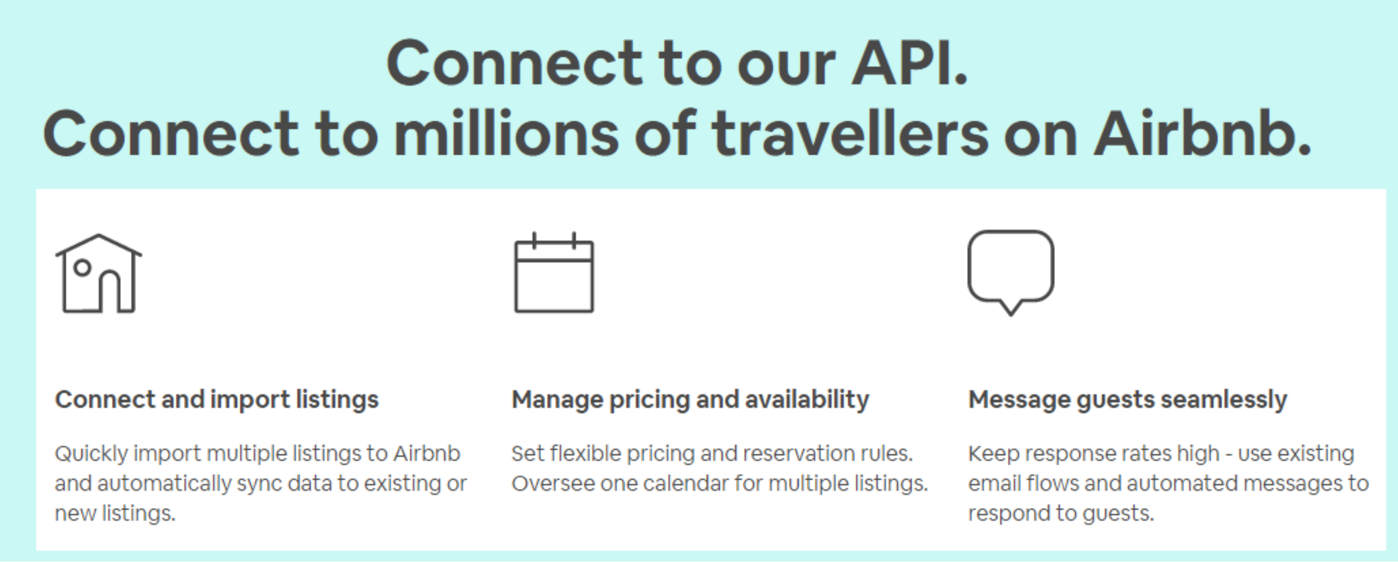
Places

Help users discover the world with rich location data for over 200 million places. Enable them to find specific places using phone numbers, addresses, and real-time signals.

Business APIs

Airbnb

- When you search for hotel online, API helps you
  - ✓ to interact with hotels database
  - ✓ filter out the entries based upon your specification
  - ✓ Do the booking



Connect to our API.
Connect to millions of travellers on Airbnb.

Connect and import listings

Quickly import multiple listings to Airbnb and automatically sync data to existing or new listings.

Manage pricing and availability

Set flexible pricing and reservation rules. Oversee one calendar for multiple listings.

Message guests seamlessly

Keep response rates high - use existing email flows and automated messages to respond to guests.

Payment APIs

- Payment APIs are APIs (Application Programming Interfaces) designed for managing payments.

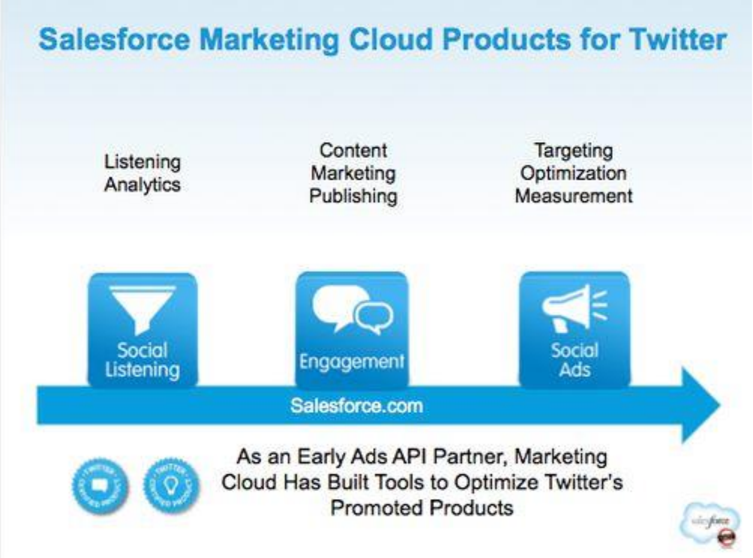
- They enable eCommerce sites to process:
- credit cards,
- track orders,
- and maintain customers lists.

- In many instances, they can help protect merchants from fraud and information breaches.

Source : rapidapi

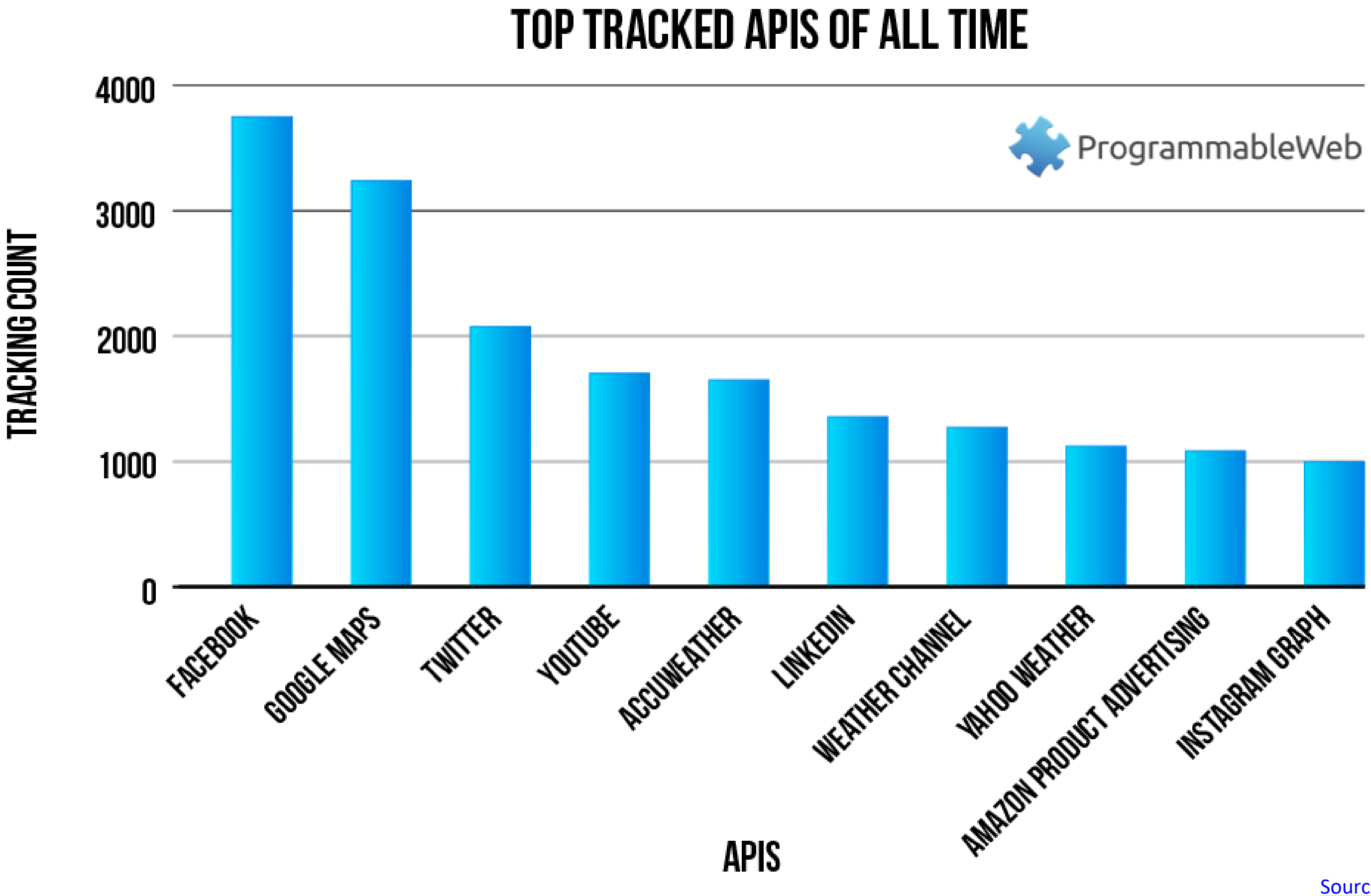
Marketing API

- Marketing APIs are a collection of API endpoints that can be used to help you advertise on social media platform like Facebook, Twitter etc.
- Amazon offers vendors and professional sellers the opportunity to advertise their products on Amazon



Source : pinterest

Popular APIs



Source: programmable web

The Components of APIs

What resources are shared and with whom?

- Shared assets / Resources
  - Are the currency of an API
  - Can be anything a company wants to share - data points, pieces of code, software, or services that a company owns and sees value in sharing

- API
  - Acts as a gateway to the server
  - Provides a point of entry for developers to use those assets to build their own software
  - Can acts like a filter for those assets - only reveal what you want them to reveal

- Audience
  - Immediate audience of an API is rarely an end user of an app
  - Typically developers creating software or an app around those assets

- Apps
  - Results in apps that are connected to data and services, allowing these apps to provide richer, more intelligent experiences for users
  - API-powered apps are also compatible with more devices and operating systems
  - Enable end users tremendous flexibility to access multiple apps seamlessly between devices, use social profiles to interact with third-party apps etc.

API

Benefits

- Acts as a doorway that people with the right key can get through
  - a gateway to the server and database that those with an API key can use to access whatever assets you choose to reveal
- Lets applications (and devices) seamlessly connect and communicate
  - With API one can create a seamless flow of data between apps and devices in real time
  - Enables developers to create apps for any format—a mobile app, a wearable, or a website
  - Allows apps to “talk to” one another - heart of how APIs create rich user experiences
- Let you build one app off another app
  - Allows you to write applications that use other applications as part of their core functionality
  - Developers get access to reusable code and technology
  - Other technology gets automatically leverages for your own apps
- Acts like a “universal plug”
  - Apps in Different languages does not matter
  - Everyone, no matter what machine, operating system, or mobile device they’re using—gets the same access
  - Standardizes access to app and its resources
- Acts as a filter
  - Security is a big concern with APIs
  - Gives controlled access to assets, with permissions and other measures that keep too much traffic

Types

Public APIs vs. Private APIs - Very Different Value Chains

- Public API
  - Twitter API, Facebook API, Google Maps API, and more
  - Granting Outside Access to Your Assets
  - Provide a set of instructions and standards for accessing the information and services being shared
  - Making it possible for external developers to build an application around those assets
  - Much more restricted in the assets they share, given they're sharing them publicly with developers around the web

- Private API
  - Self-Service Developer & Partner Portal API
  - Far more common (and possibly even more beneficial, from a business standpoint)
  - Give developers an easy way to plug right into back-end systems, data, and software
  - Letting engineering teams do their jobs in less time, with fewer resources
  - All about productivity, partnerships, and facilitating service-oriented architectures

REST

Representation State Transfer

- Most commonly known item in API space
- has become very common amongst web APIs
- First defined by Roy Fielding in his doctoral dissertation in the year 2000
- Architectural system defined by a set of constraints for web services based on
  - stateless design ethos
  - standardized approach to building web APIs

- Operations are usually defined using GET, POST, PUT, and other HTTP methodologies
- One of the chief properties of REST is the fact that it is hypermedia rich
- Supports a layered architecture, efficient caching, and high scalability

- All told, REST is a very efficient, effective, and powerful solution for the modern micro service API industry

gRPC

Backed by Google

- Actually a new take on an old approach known as RPC, or Remote Procedure Call
- RPC is a method for executing a procedure on a remote server
- RPC functions upon an idea of contracts, in which the negotiation is defined and constricted by the client-server relationship rather than the architecture itself
    - RPC gives much of the power (and responsibility) to the client for execution
    - offloading much of the handling and computation to the remote server hosting the resource

- RPC is very popular for IoT devices and other solutions requiring custom contracted communications for low-power devices
    - gRPC is a further evolution on the RPC concept, and adds a wide range of features

- The biggest feature added by gRPC is the concept of protobufs
    - Protobufs are language and platform neutral systems used to serialize data, meaning that these communications can be efficiently serialized and communicated in an effective manner
- gRPC has a very effective and powerful authentication system that utilizes SSL/TLS through Google's token-based system
- Open source, meaning that the system can be audited, iterated, forked, and more

REST vs gRPC vs GraphQL

When to use?

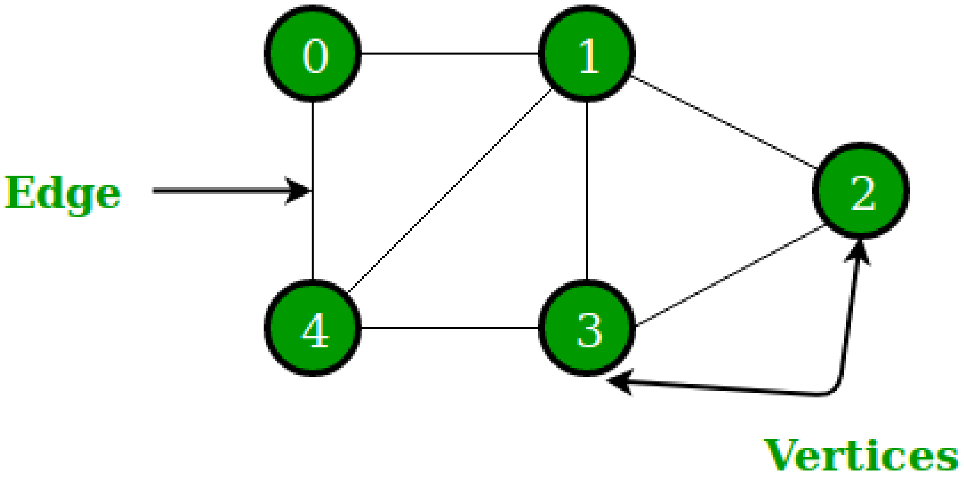
- REST
  - A stateless architecture for data transfer that is dependent on hypermedia
  - Tie together a wide range of resources that might be requested in a variety of formats for different purposes
  - Systems requiring rapid iteration and standardized HTTP verbiage will find REST best suited for their purposes
- gRPC
  - A nimble and lightweight system for requesting data
  - Best used when a system requires a set amount of data or processing routinely and requester is either low power or resource-jealous
  - IoT is a great example
- GraphQL
  - An approach wherein the user defines the expected data and format of that data
  - Useful in situations in which the requester needs the data in a specific format for a specific use

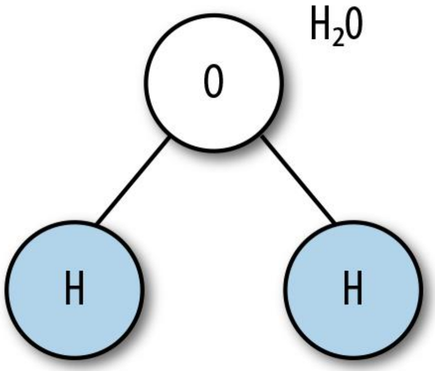
BITS Pilani
Pilani Campus

GraphQL (cont)

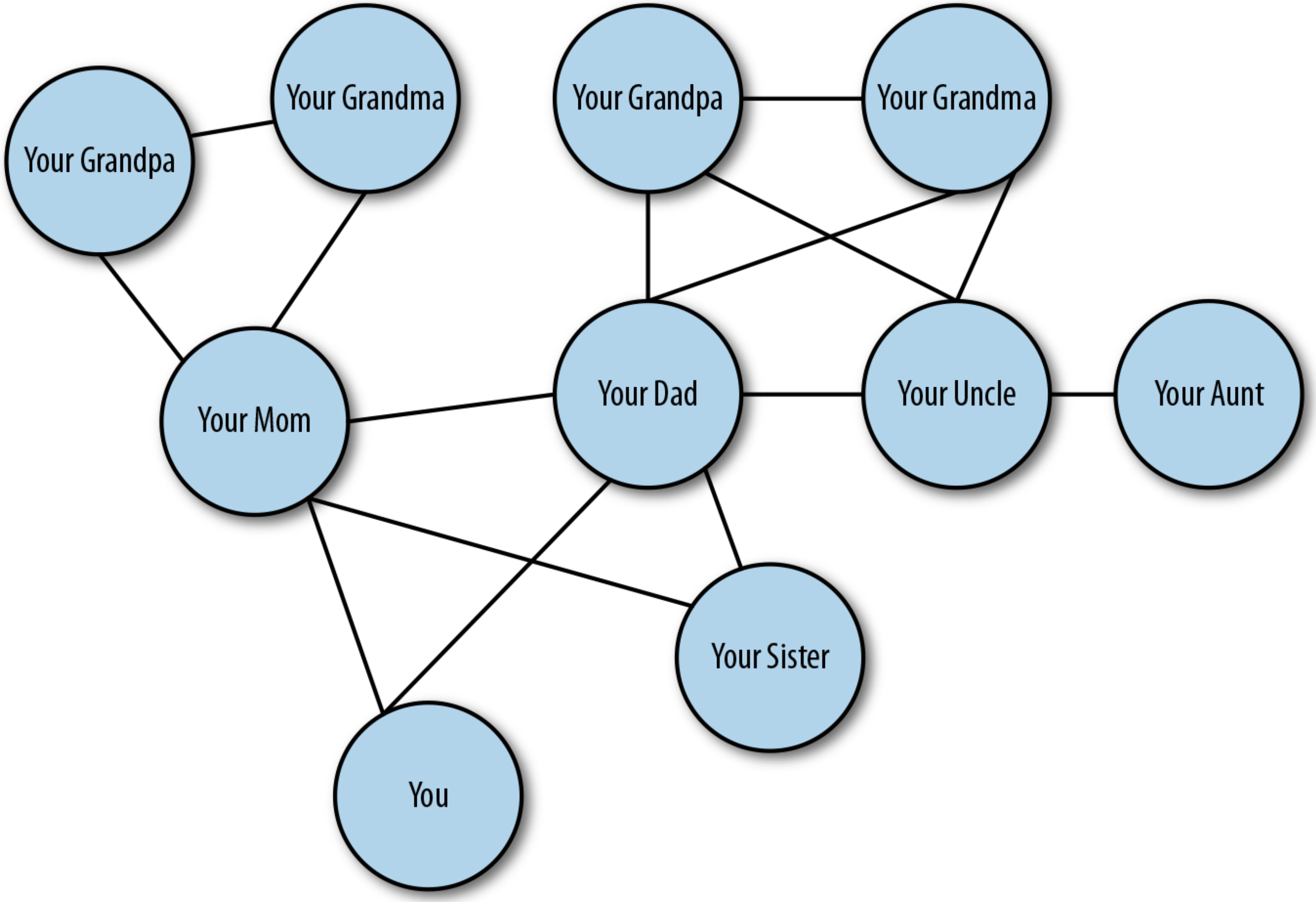
What is Graph?

A graph is a non-linear data structure, which consists of vertices(or nodes) connected by edges(or arcs) where edges may be directed or undirected.

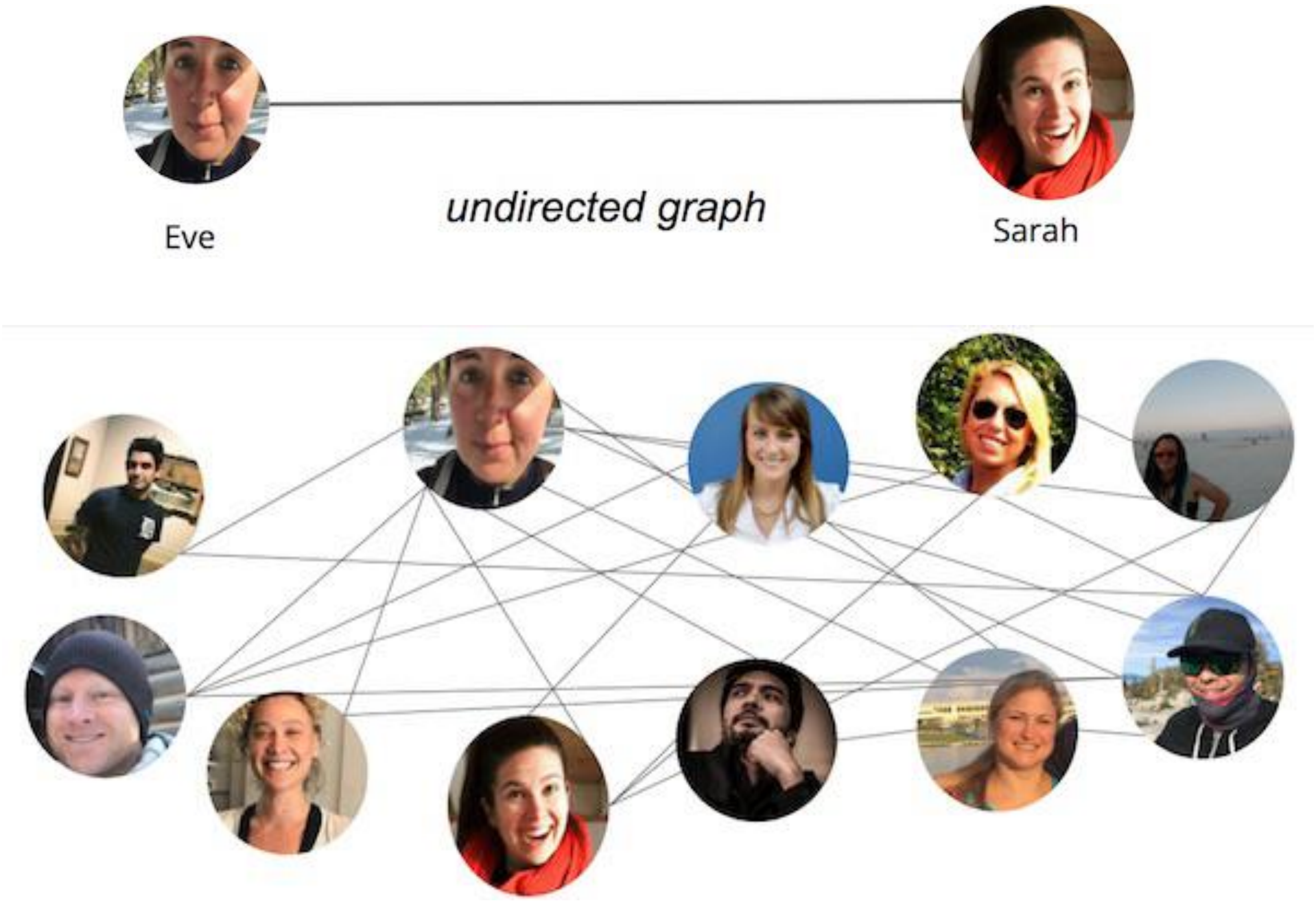




Graph Example



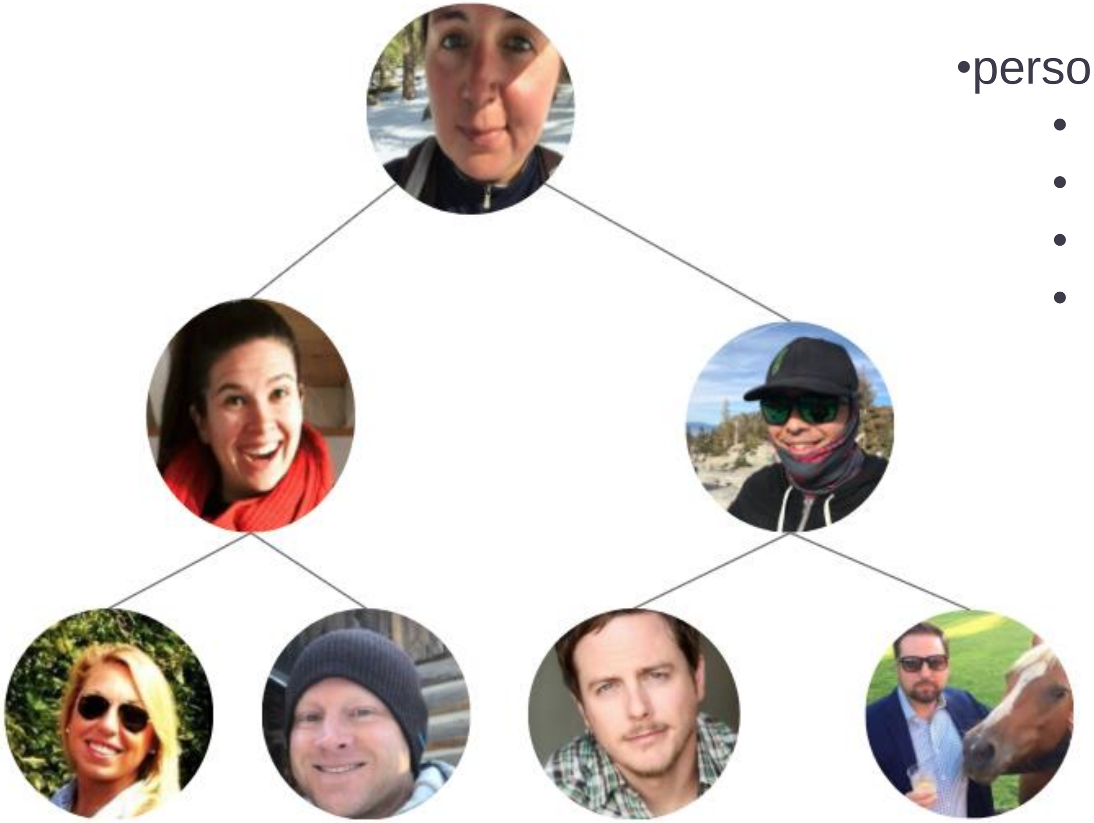
Facebook



Applications

- **Google maps** uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.
- In **Facebook**, users are considered to be the vertices and if they are friends then there is an edge running between them. Facebook's Friend suggestion algorithm uses graph theory. Facebook is an example of **undirected graph**.
- In **World Wide Web**, web pages are considered to be the vertices. There is an edge from a page u to other page v if there is a link of page v on page u. This is an example of **Directed graph**. It was the basic idea behind Google Page Ranking Algorithm.

Friends Tree



- person
    - name
    - location
    - birthday
    - friends
        - friend name
        - friend location
        - friend birthday

Simple Examples

```
{
  "data": {
    "books": [
      {
        "title": "The Lord of the Rings",
        "author": "J. R. R. Tolkien"
      },
      {
        "title": "Pride and Prejudice",
        "author": "Jane Austen"
      }
    ]
  }
}
```

Question-1

Write a query to fetch a list of books and for each book, it retrieve the title and author.

Answer-1

```
{
  books {
    title
    author
  }
}
```

Question-2

Say you only want books by a specific author.

```
query GetBooksByAuthor($author: String!) {
  books(author: $author) {
    title
    author
  }
}

# With variables:
{
  "author": "J. R. R. Tolkien"
}
```

Nested Query and Fetching of Data

```
query GetBookDetails($title: String!) {
  book(title: $title) {
    title
    author {
      name
    }
  }
}

# With variables:
{
  "title": "The Hitchhiker's Guide to the Galaxy"
}
```

We have a query named GetBookDetails with a variable $title.
We fetch the book and request nested data for the author field, specifying that we only want the name.

Case Study
Design a GraphQL schema

1. As a user, I want to search for a list of businesses by category, location, and name.
2. As a user, I want to view details for each business (name, description, address, photos, etc.).
3. As a user, I want to view reviews for each business, including a summary for each business, and rank my search by favorably reviewed businesses.
4. As a user, I want to create a review for a business.
5. As a user, I want to connect my friends and users who have tastes that I like, so I can follow my friends’ reviews.
6. As a user, I want to receive personalized recommendations based on reviews I have previously written and my social network.

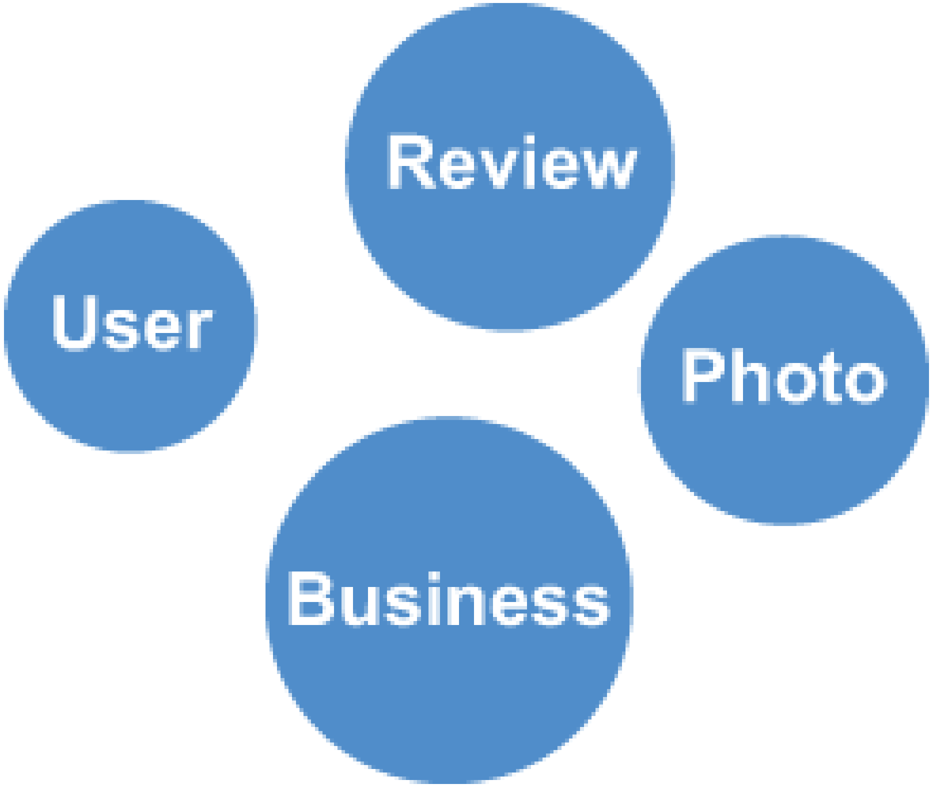
Question

What are the entities here?

Find out the entities

1. As a user, I want to search for a list of businesses by category, location, and name.
2. As a user, I want to view details for each business (name, description, address, photos, etc.).
3. As a user, I want to view reviews for each business, including a summary for each business, and rank my search by favorably reviewed businesses.
4. As a user, I want to create a review for a business.
5. As a user, I want to connect my friends and users who have tastes that I like, so I can follow my friends’ reviews.
6. As a user, I want to receive personalized recommendations based on reviews I have previously written and my social network.

Entities becomes node

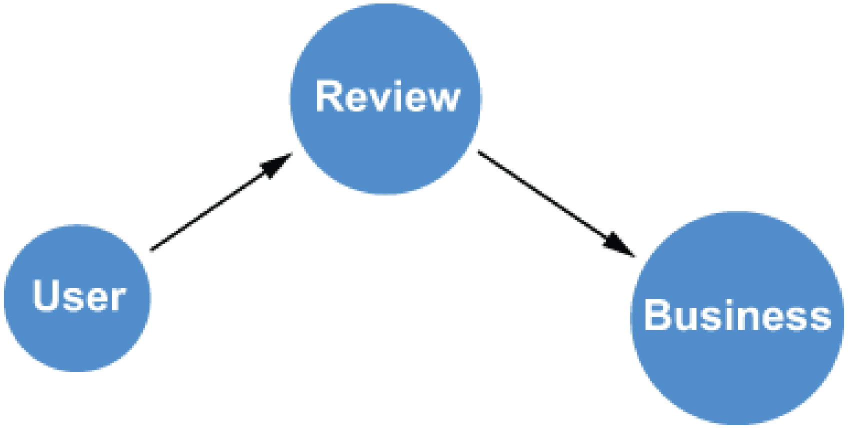


Question:

How are these entities connected?

Relationship between the entities

1. Users write reviews.
2. Reviews are connected to a business.
3. Users upload photos.
4. Photos are tagged to businesses.



Graphs in GraphQL

GraphQL models our business domain as a graph. With GraphQL, we define this graph model by creating a GraphQL schema, which we do by writing GraphQL type definitions.

In the schema, we define types of nodes, the fields available on each node, and how they are connected by relationships.

The most common way of creating a GraphQL schema is by using the GraphQL schema definition language (SDL).

GraphQL Type Definition

```
type Business {
  businessId: ID!
  name: String
  address: String
  avgStars: Float
  photos: [Photo!]!
  reviews: [Review!]!
}

type User {
  userId: ID!
  name: String
  photos: [Photo!]!
  reviews: [Review!]!
}
```

❶ Each type of object or entity in our graph becomes a GraphQL type

❷ Each type should have some field that uniquely identifies that object.

❸ Fields can be references to other types—in this case, a one-to-many relationship.

GraphQL Type definition

```
type Photo {
  business: Business!
  user: User!
  photoId: ID!
  url: String
}

type Review {
  reviewId: ID!
  stars: Float
  text: String
  user: User!
  business: Business!
                    ❹ Connection references can also represent one-to-one relationships
}
```

Type definition

Each type should have some field that uniquely identifies that object. **ID is a special GraphQL scalar** used to represent this unique field. Internally, we treat ID fields as strings.

The exclamation ! indicates this field is required; we cannot have a User object in our GraphQL API without a value for the userId field.

The brackets [] here indicate this is a one-to-many relationship; one User can create zero or more reviews and a Review can be written by only one User.

To represent **one-to-one relationships**, we simply leave off the brackets, indicating this is not an array field.

Query Fields

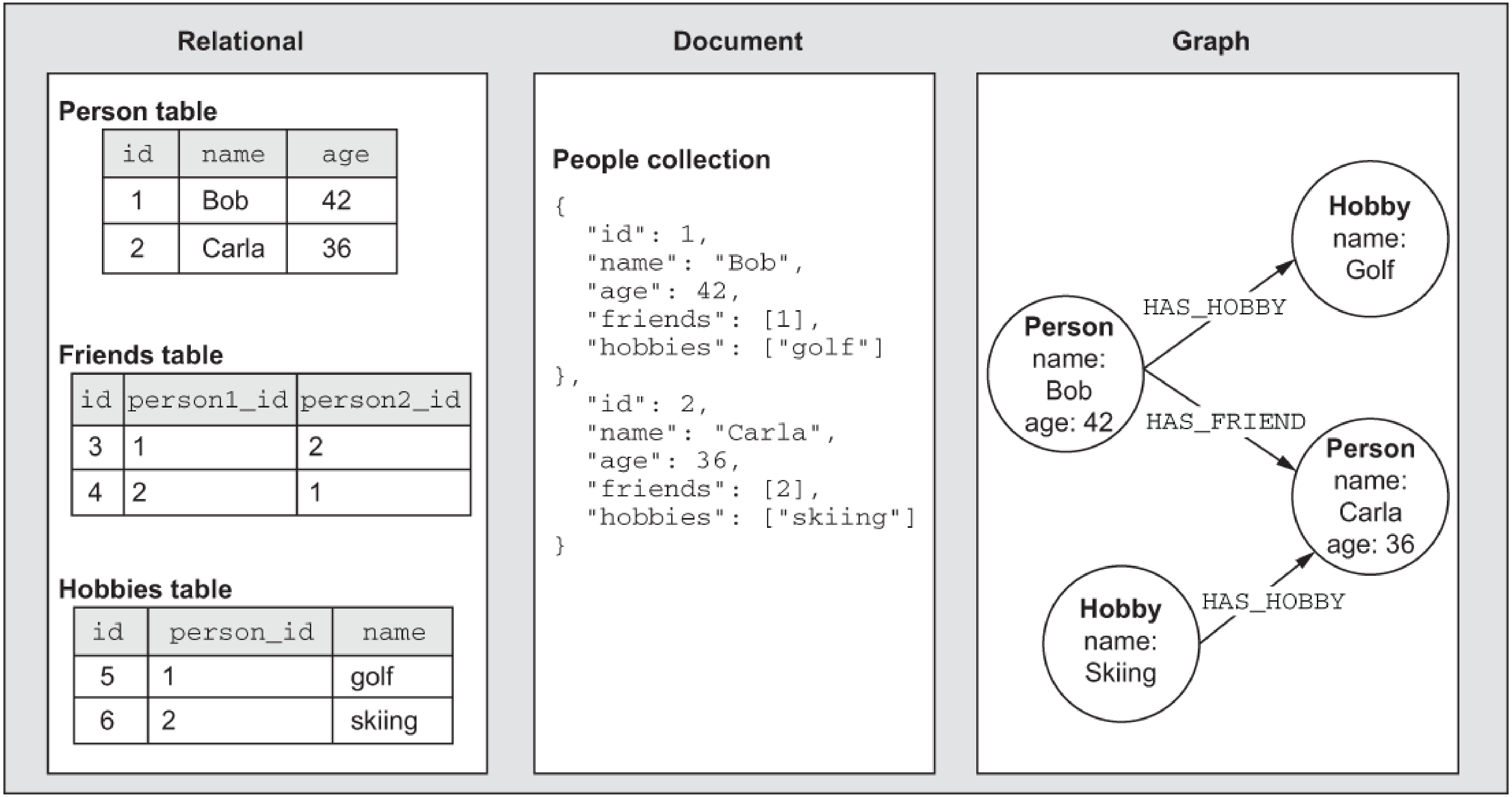
What operations does the client need to complete?

```
type Query {
  allBusinesses: [Business!]!
  businessBySearchTerm(search: String!): [Business!]!
  userById(id: ID!): User
}
```

GraphQL query to search for businesses and reviews

```
{
  businessBySearchTerm(search: "Library") {
    name
    avgStars
    reviews {
      stars
      text
      user {
        name
      }
    }
  }
}
```

Comparison of DB Models



Relational

Person table

id	name	age
1	Bob	42
2	Carla	36

Friends table

id	person1_id	person2_id
3	1	2
4	2	1

Hobbies table

id	person_id	name
5	1	golf
6	2	skiing

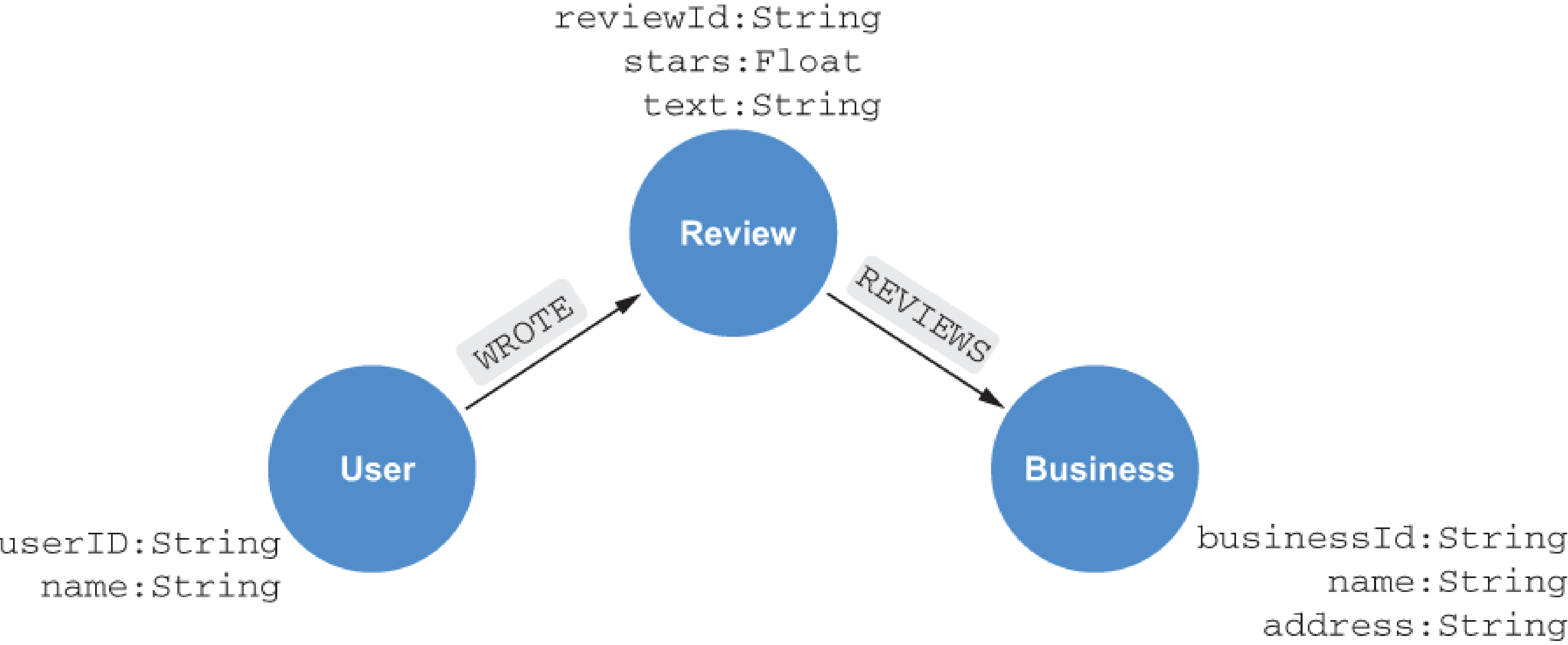
Document

People collection

```
{
  "id": 1,
  "name": "Bob",
  "age": 42,
  "friends": [1],
  "hobbies": ["golf"]
},
  "id": 2,
  "name": "Carla",
  "age": 36,
  "friends": [2],
  "hobbies": ["skiing"]
}
```

Graph

Case Study : Models



reviewId:String
stars:Float
text:String

userID:String
name:String

businessId:String
name:String
address:String

Question

Design a simple GraphQL schema for a recipe sharing platform with advanced features like ingredient search and recipe ratings.

Answer

```
type Recipe {
  id: ID!
  title: String!
  description: String!
  author: User!
  ingredients: [Ingredient!]!
  instructions: [String!]!
  ratings: [Rating!]
  averageRating: Float
}

type Ingredient {
  id: ID!
  name: String!
  recipes: [Recipe!]!
}
```

```
type User {
  id: ID!
  username: String!
  email: String!
  recipes: [Recipe!]!
}

type Rating {
  id: ID!
  value: Int!
  recipe: Recipe!
  user: User!
}
```

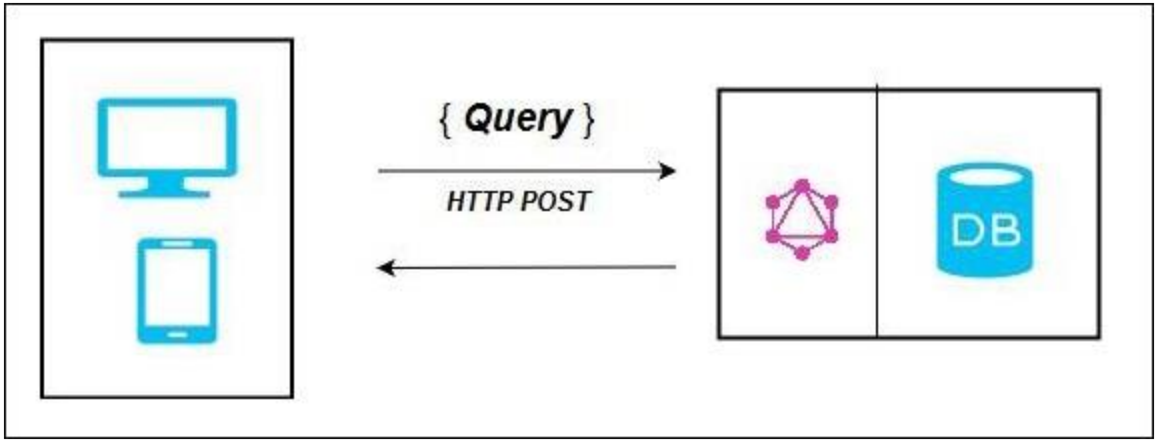
Answer (cont)

```
type Query {
  recipe(id: ID!): Recipe
  ingredient(id: ID!): Ingredient
  recipesByIngredient(ingredientId: ID!): [Recipe!]!
}

type Mutation {
  createRecipe(authorId: ID!, title: String!, description: String!, ingredients: [ID!]!, instructions: [String!]!): Recipe
  rateRecipe(recipeId: ID!, userId: ID!, value: Int!): Rating
}

schema {
  query: Query
  mutation: Mutation
}
```

Tips

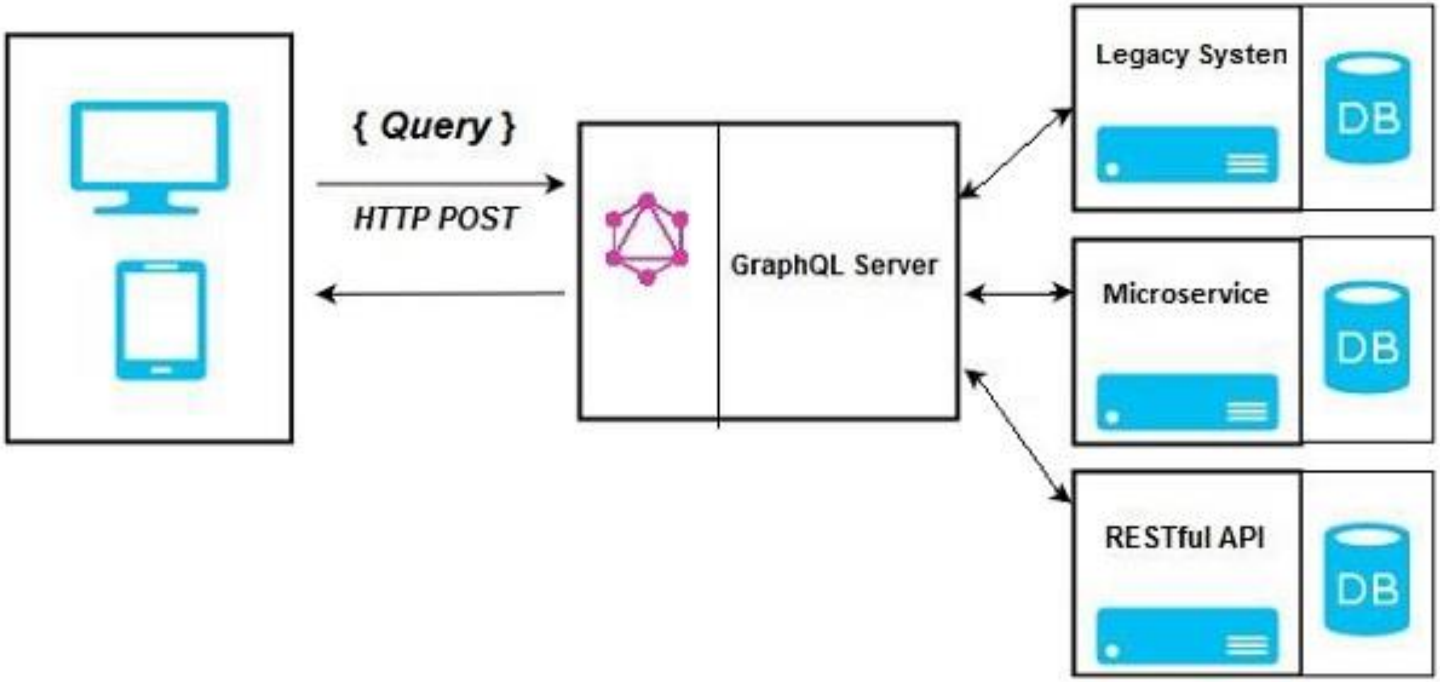
1. Start with a clear understanding of your data: Before designing your schema, thoroughly understand your data model and the relationships between different entities. This understanding will inform your schema structure.
2. Use GraphQL type system effectively: GraphQL's type system allows you to define the shape of your API. Use scalar types (int, float, string, boolean, ID) for simple values and object types for more complex structures.
3. Normalize your data: Aim to keep your GraphQL schema normalized to avoid redundancy and inconsistency. Use object relationships to represent connections between different types.
4. Define clear relationships: Use GraphQL's built-in types like GraphQLObjectType, GraphQLList, and GraphQLNonNull to define relationships between types. Ensure that relationships accurately represent the underlying data model.
5. Avoid overly nested structures: While GraphQL allows nesting of queries, be cautious of creating overly nested structures that can lead to inefficient queries. Strike a balance between depth and simplicity.
6. Test your schema: Test your GraphQL schema thoroughly to ensure that it behaves as expected, handles edge cases gracefully, and performs well under load. Use tools like GraphQL Playground, GraphiQL, or Jest for schema testing.

GraphQL Server with Connected Database



This architecture has a GraphQL Server with an integrated database and can often be used with new projects. On the receipt of a Query, the server reads the request payload and fetches data from the database. This is called resolving the query. The response returned to the client adheres to the format specified in the official GraphQL specification.

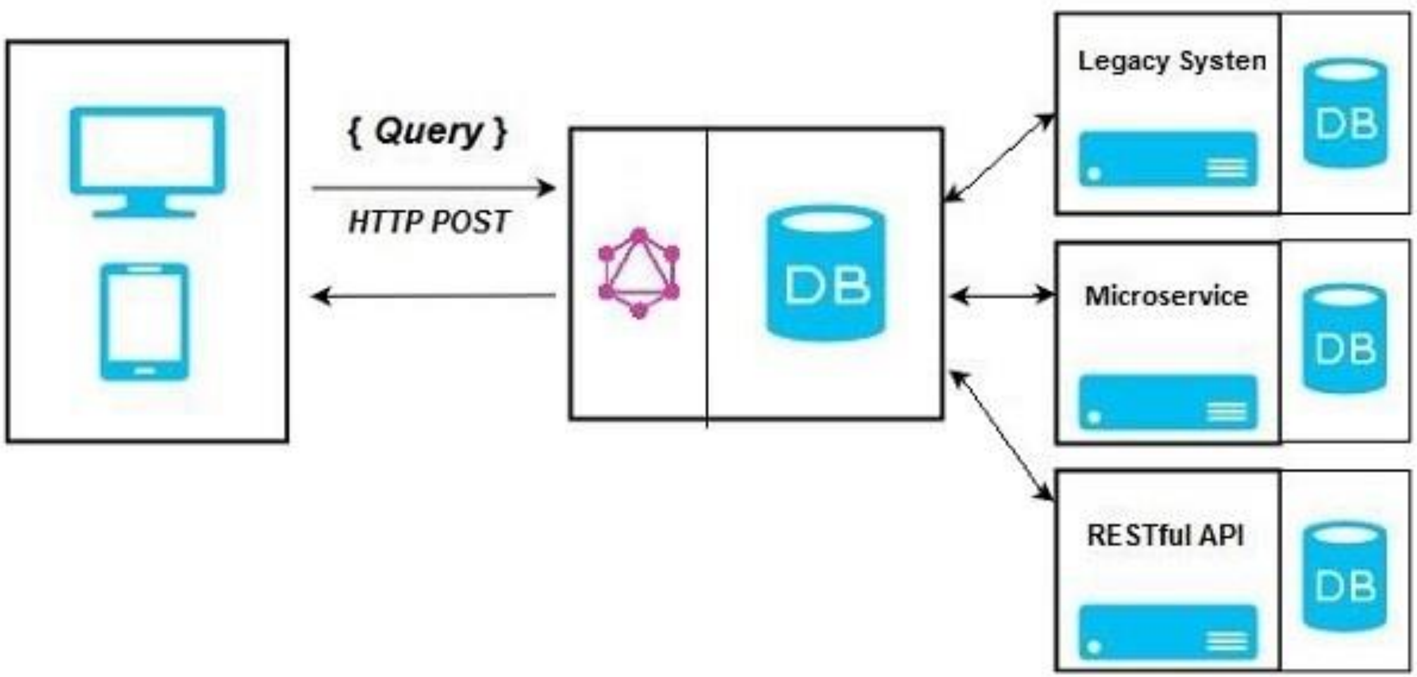
GraphQL Server Integrating Existing Systems



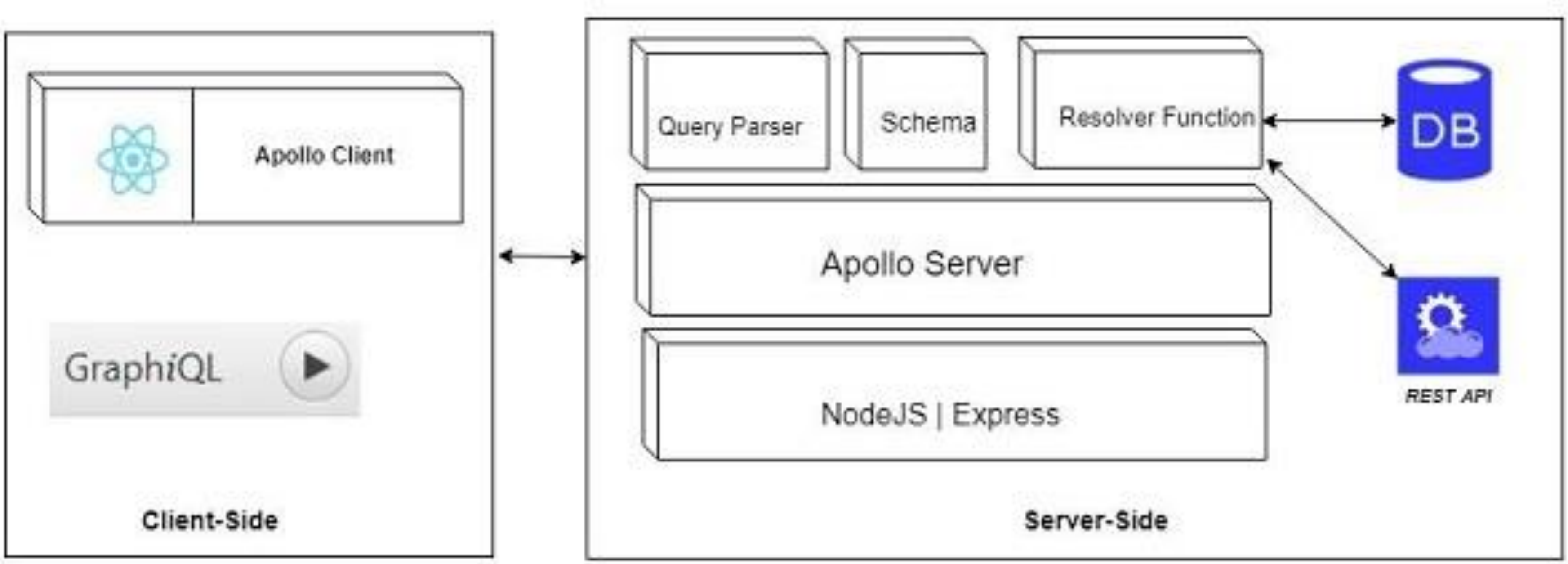
This approach is helpful for companies which have legacy infrastructure and different APIs. GraphQL can be used to unify microservices, legacy infrastructure and third-party APIs in the existing system. In the above diagram, a GraphQL API acts as an interface between the client and the existing systems. Client applications communicate with the GraphQL server which in turn resolves the query.

Hybrid Approach

Finally, we can combine the above two approaches and build a GraphQL server. In this architecture, the GraphQL server will resolve any request that is received. It will either retrieve data from connected database or from the integrated API's.



Application Components



Structure of GraphQL

Request Structure:

A GraphQL request over HTTP typically includes the following information:

- **query:** This is the mandatory GraphQL query string, specifying the desired data and operations.
- **operationName (optional):** If the query contains multiple named operations, this field identifies which one to execute.
- **variables (optional):** This object holds values for any variables defined within the query.
- **extensions (optional):** This section allows for custom extensions, specific to the implementation.

Response Structure:

The GraphQL server responds with a JSON object containing:

- **data:** This field holds the requested data, structured according to the query.
- **errors (optional):** If errors occurred during execution, an array of error objects is included here.

Validation and Execution

GraphQL employs a two-step process to handle incoming queries: validation and execution.

Validation:

- **Purpose:** This stage ensures the received query is syntactically correct, adheres to the defined schema, and follows any custom validation rules.
- **Process:**
  - **Parsing:** The incoming query string is transformed into an Abstract Syntax Tree (AST), a hierarchical representation of the query structure.
  - **Validation Rules:** The AST is traversed against a set of validation rules defined by the GraphQL specification and potentially extended by your server implementation. These rules check for things like:
    - Valid field names and types based on the schema.
    - Correct usage of directives (if applicable).
    - Required arguments are provided.
    - Custom rules you might implement for specific validation needs (e.g., checking email format).

Outcome of validation

- **Outcome:**
  - **Valid Query:** If the query passes all validation rules, processing continues to the execution phase.
  - **Invalid Query:** If any validation rule fails, the server responds with an error message detailing the specific issue(s) within the query.

Execution

- **Purpose:** This stage interprets the validated query and retrieves the requested data from your data sources.
- **Process:**
  - **Resolvers:** Each field in the query is mapped to a corresponding resolver function. This function is responsible for fetching the data for that specific field. Resolvers can interact with your database, application logic, or external APIs to retrieve the data.
  - **Data Resolution:** Resolvers can leverage arguments provided in the query to filter or manipulate the retrieved data.
  - **Recursive Execution:** If a field has nested sub-selections (fields within fields), the process is recursively repeated for each sub-selection.

Example : Bookstore API with GraphQL Validation and Execution

```
type Book {
  id: ID!
  title: String!
  author: Author!
}

type Author {
  id: ID!
  name: String!
}

type Query {
  books(genre: String): [Book!]!
  book(id: ID!): Book
}
```

A client wants to retrieve a book with a specific ID (id: 123).

Validation:

1. **Parsing**: The query string "query { book(id: 123) { title } }" is parsed into an AST.
2. **Validation Rules**: The AST is validated against the schema:
   - **Valid field names and types**: book and title fields are present in the schema with correct types.
   - **Required arguments**: The id argument for the book field is provided.
3. **Outcome**: The query is valid and proceeds to execution.

Execution

```
{
  "data": {
    "book": {
      "id": "123",
      "title": "The Lord of the Rings"
    }
  }
}
```

Resolvers:
The book query field maps to a resolver function. This resolver function retrieves book data from the database based on the provided id (123).
Data Resolution: The resolver might filter specific fields or format the data before returning it.
Response: The server responds with a JSON object containing:

Question

Is REST faster than GraphQL?

Is REST faster than GraphQL?

The speed of GraphQL vs Rest api depends on various factors, such as the specific use case, network conditions, and implementation details. While REST’s simplicity and caching mechanisms can offer excellent performance in certain scenarios, GraphQL’s optimized data fetching and fine-grained control can provide superior performance in others. It’s important to evaluate the specific requirements of your project before choosing between REST and GraphQL.

Does GraphQL always outperform REST?

No, GraphQL doesn’t always outperform REST. While GraphQL’s efficiency in data retrieval is well-suited for certain use cases, REST’s simplicity and caching mechanisms can offer superior performance in other scenarios. The choice between REST and GraphQL should be based on careful consideration of your project’s specific requirements and performance objectives.

Does GraphQL introduce more complexity than REST?

GraphQL’s power and flexibility come at the cost of increased complexity compared to REST. The nature of GraphQL schemas, resolvers, and type systems requires a learning curve for developers. However, this additional complexity can be justified by the improved performance and flexibility that GraphQL brings, especially for complex and evolving data requirements.

Can I use both REST and GraphQL together?

Yes, it is possible to use REST and GraphQL together within a single project. This approach, often referred to as “GraphQL over REST,” allows you to leverage the strengths of both paradigms. By wrapping existing RESTful APIs with a GraphQL layer, you can benefit from GraphQL’s flexibility and data-fetching optimizations while still utilizing the existing REST infrastructure.

Case Study

Case

Read the case and answer the questions

XYZ Corporation is a large e-commerce company that has been using a REST API to manage its product catalog. The company has been facing several issues with its REST API, including slow performance, complex data queries, and frequent changes in the data schema. The company has decided to migrate its REST API to GraphQL to address these issues.

As a senior developer in the company, your task is to design and implement the migration strategy. Your solution should address the following challenges

- Data Modeling: The existing data schema needs to be mapped to a GraphQL schema.
- Query Complexity: The GraphQL API needs to handle complex data queries efficiently.
- Caching: The GraphQL API needs to cache frequently accessed data to improve performance.
- Authentication and Authorization: The GraphQL API needs to implement a secure authentication and authorization mechanism.
- Testing: The GraphQL API needs to be thoroughly tested to ensure its functionality and performance.

Case (contd)

Read the question and answer accordingly

Your solution should include the following:

- A **detailed plan** for mapping the existing REST API data schema to a GraphQL schema, including the tools and technologies that will be used.
- A **strategy** for handling complex data queries in the GraphQL API, including the use of query optimization techniques.
- A **caching strategy** for frequently accessed data, including the use of a caching layer such as Redis or Memcached.
- An **authentication** and **authorization** mechanism for the GraphQL API, including the use of OAuth 2.0 or JWT tokens.
- A **comprehensive testing plan** for the GraphQL API, including unit testing, integration testing, and performance testing.

Your solution should also include a cost-benefit analysis, outlining the costs associated with implementing the strategy and the potential benefits to XYZ Corporation.

Solution

Mapping the existing REST API data schema to a GraphQL schema

- The first step in the migration process is to map the existing REST API data schema to a GraphQL schema. This can be done using a tool like Apollo GraphQL or a custom schema mapping solution. The GraphQL schema should be designed in a way that allows for efficient and flexible querying of the data. The schema should also be designed to handle future changes in the data schema.

Solution

Handling complex data queries in the GraphQL API

GraphQL provides several techniques for handling complex data queries. These include query batching, pagination, and data caching. Query batching can be used to combine multiple queries into a single request, reducing the number of round trips to the server. Pagination can be used to limit the amount of data returned by a query, making it easier to manage large data sets. Data caching can be used to store frequently accessed data in memory or in a caching layer like Redis or Memcached.

Solution

Caching frequently accessed data

Caching frequently accessed data is critical for improving the performance of the GraphQL API. A caching layer like Redis or Memcached can be used to store frequently accessed data in memory, reducing the number of database queries required to serve a request. The caching layer should be configured to expire data after a certain amount of time to ensure that stale data is not served to clients.

Solution

Authentication and authorization mechanism

The GraphQL API should implement a secure authentication and authorization mechanism to ensure that only authorized users can access sensitive data. OAuth 2.0 or JWT tokens can be used to implement the authentication and authorization mechanism. The authentication and authorization mechanism should be designed to be flexible and extensible, allowing for future changes in the authentication and authorization requirements.

Solution

Testing plan for the GraphQL API

The GraphQL API should be thoroughly tested to ensure its functionality and performance. The testing plan should include unit testing, integration testing, and performance testing. Unit tests should be used to test individual components of the GraphQL API, while integration tests should be used to test the interaction between different components of the API. Performance testing should be used to test the performance of the API under various load conditions.

Solution

Cost-benefit analysis:

The migration from REST API to GraphQL API involves significant costs, including the cost of developing and testing the new API and the cost of training developers on the new API. However, the potential benefits of the migration are also significant. These benefits include improved performance, increased flexibility, and easier management of the data schema. The improved performance of the GraphQL API can lead to better user experiences and increased revenue. Additionally, the flexibility of the GraphQL API can lead to faster development cycles and reduced maintenance costs. Overall, the benefits of the migration outweigh the costs.

Case – 2 (Migration from REST to GraphQL)

University Library of Collection of Books

- Book Title: "The Alchemist"
- Author: Paulo Coelho
- ISBN: 978-0061122415
- Publisher: HarperOne
- Publication Date: April 25, 2006
- Image: URL to book cover image
- Book Ratings: Average rating of 4.5 out of 5 stars, based on 500 reviews.

Case -2

REST API End Points

- **GET /books**: Retrieves a list of all the books in the library.
- **GET /books/:id**: Retrieves information about a specific book based on its ID.
- **GET /books/:id/image**: Retrieves the cover image of a specific book based on its ID.
- **GET /books/:id/ratings**: Retrieves the average rating and number of reviews for a specific book based on its ID.

Case-2 (GraphQL code)

Transform REST to GraphQL

```
type Book {
  id: ID!
  title: String!
  author: String!
  isbn: String!
  publisher: String!
  publicationDate: String!
  image: String!
  ratings: Rating!
}

type Rating {
  average: Float!
  totalReviews: Int!
}

type Query {
  books: [Book!]!
  book(id: ID!): Book
}
```

```
type Mutation {
  addBook(title: String!, author: String!, isbn: String!, publisher: String!, publicationDate: String!, image: String!): Book!
  updateBook(id: ID!, title: String, author: String, isbn: String, publisher: String, publicationDate: String, image: String): Book!
  deleteBook(id: ID!): ID!
}

query {
  books {
    id
    title
    author
    ratings {
      average
      totalReviews
    }
  }
}
```

Case-2 (GraphQL)

Other Query and Mutations

```
query {
  book(id: "123") {
    id
    title
    author
    isbn
    publisher
    publicationDate
    image
    ratings {
      average
      totalReviews
    }
  }
}
```

```
mutation {
  updateBook(id: "123", title: "The Alchemist: 25th
Anniversary Edition", author: "Paulo Coelho", isbn:
"978-0062315007", publisher: "HarperOne",
publicationDate: "April 15, 2014", image:
"https://example.com/alchemist-25th-
anniversary.jpg") {
    id
    title
    author
    isbn
    publisher
    publicationDate
    image
    ratings {
      average
      totalReviews
    }
  }
}
```

Case-2: GraphQL code

```
mutation {
  addBook(title: "The Alchemist", author: "Paulo Coelho",
isbn: "978-0061122415", publisher: "HarperOne",
publicationDate: "April 25, 2006", image:
"https://example.com/alchemist.jpg") {
    id
    title
    author
    isbn
    publisher
    publicationDate
    image
    ratings {
      average
      totalReviews
    }
  }
}
```

```
mutation {
  deleteBook(id: "123")
}
```

Paginating APIs

Quiz

What is pagination in API design?

- a) A method for encrypting API data
- b) A method for compressing API data
- c) A technique for breaking up large API responses into smaller, more manageable pieces
- d) A method for caching API data

Quiz- Answer

What is pagination in API design?

- a) A method for encrypting API data
- b) A method for compressing API data
- c) A technique for breaking up large API responses into smaller, more manageable pieces
- d) A method for caching API data

- Answer: c) A technique for breaking up large API responses into smaller, more manageable pieces

Paginating APIs

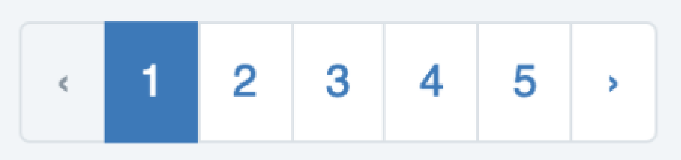
- Paginating APIs can help with scaling
  - ✓ Often, APIs need to handle large datasets
  - ✓ An API call might end up returning thousands of items
  - ✓ Returning too many items can overload the backend and even slow down clients that can't handle large datasets

- Important to paginate large result sets
  - ✓ Splits long lists of data into smaller chunks
  - ✓ Minimizes response times for requests
  - ✓ Makes responses easier to handle

nordicapi

Offset-Based Pagination

- The most widely used pagination technique



- Uses limits and offsets to implement pagination
- Clients provide
  - ✓ a page size that defines the maximum number of items to return
  - ✓ a page number that indicates the starting position in the list of items
- Servers can easily construct a query to fetch results
- APIs, such as GitHub’s, support this kind of pagination
  - ✓ Clients can simply make a request with page and per_page parameters specified in the URL
  - ✓ https://api.github.com/user/repos?page=5&per_page=10

Offset-Based Pagination (2)

Advantages and disadvantages

- Advantages
  - ✓ extremely simple to implement, both for clients and the server
  - ✓ has user experience advantages
  - ✓ allows users to jump into any arbitrary page instead of forcing them to scroll through the entire content

- Disadvantages
  - ✓ Inefficient for large datasets - SQL queries with large offsets are pretty expensive
  - ✓ Can be unreliable when the list of items changes frequently
  - ✓ Can be tricky in a distributed system
  - ✓ For large offsets, might need to scan a number of shards before get to the desired set of items

- Offset paginations can be great when pagination depth is limited and clients can tolerate duplicates or missed items

Cursor-Based Pagination

- To address the problems of offset-based pagination, cursor-based pagination can be used
- Clients first send a request while passing only the desired number of items
  - ✓ Server then responds with the requested number of items, in addition to a next cursor
  - ✓ In the subsequent request, along with the number of items, clients pass this cursor indicating the starting position for the next set of items

- Systems that store data in a SQL database can create queries based on the cursor values and retrieve results
- Several modern APIs, including those of Slack, Stripe, Twitter, and Facebook, offer cursor-based pagination

- Twitter API
  - – Consider this scenario: a developer wants to obtain the list of a user's followers' IDs
  - – To fetch the first page of results, the developer makes an API request, as shown here:
    - ✓ GET https://api.twitter.com/1.1/followers/ids.json?screen_name=<user>&count=50

    The Twitter API returns the following response:

    ```
    {
     "ids": [  385752029,   ...  333165023 ],
     "next_cursor": 1374004777531007833,
    }
    ```
  - – Using the value from next_cursor, the developer can then request the next page of results with the following request:
    - ✓ GET https://api.twitter.com/1.1/followers/ids.json?user_id=12345&count=50&cursor=1374004777531007833
  - – As the developer makes subsequent requests to advance through the next pages, they will eventually receive a response with "next_cursor" as 0
    - ✓ will indicate the end of the entire paginated result set

Cursor-Based Pagination(2)

Advantages and disadvantages

- Advantages
  - Performance
    - ✓ With an index on the column used in the cursor for pagination, even queries requiring scanning large tables are fast
  - Consistency
    - ✓ While paginating across results, the server returns every item exactly once.
  - Cursor-based pagination is great for large and dynamic datasets
- Disadvantages
  - Clients cannot jump to a given page
    - ✓ need to traverse through the entire result set page by page
  - The results must be sorted on a unique and sequential database column, used for the cursor value
  - Implementation is a bit more complicated than offset-based pagination, particularly for clients
  - Clients often need to store the cursor value to use it in subsequent requests

Pagination Best Practices

Best practices to paginate

BEST PRACTICE

- When implementing pagination
  - ✓ do not forget to set reasonable default and maximum values for the page size
- Avoid using offset-based pagination if clients will run queries with large offsets
- With pagination, sorting data such that newer items are returned first and older items later is sometimes better
  - ✓ clients don’t need to paginate through to the end if they are interested only in newer items
- If API does not support pagination today, introduce it later in a way that maintains backward compatibility
- Do not encode any sensitive information inside cursors
  - ✓ Clients can generally decode them

Quiz

What is the difference between limit/offset pagination and cursor pagination?

- a) Limit/offset pagination uses page numbers and page sizes, while cursor pagination uses offsets and limits
- b) Limit/offset pagination is less efficient than cursor pagination
- c) Cursor pagination requires server-side state, while limit/offset pagination does not
- d) Cursor pagination is only used for real-time data, while limit/offset pagination is used for batch data

Quiz - Answer

What is the difference between limit/offset pagination and cursor pagination?

- a) Limit/offset pagination uses page numbers and page sizes, while cursor pagination uses offsets and limits
- b) Limit/offset pagination is less efficient than cursor pagination
- c) Cursor pagination requires server-side state, while limit/offset pagination does not
- d) Cursor pagination is only used for real-time data, while limit/offset pagination is used for batch data

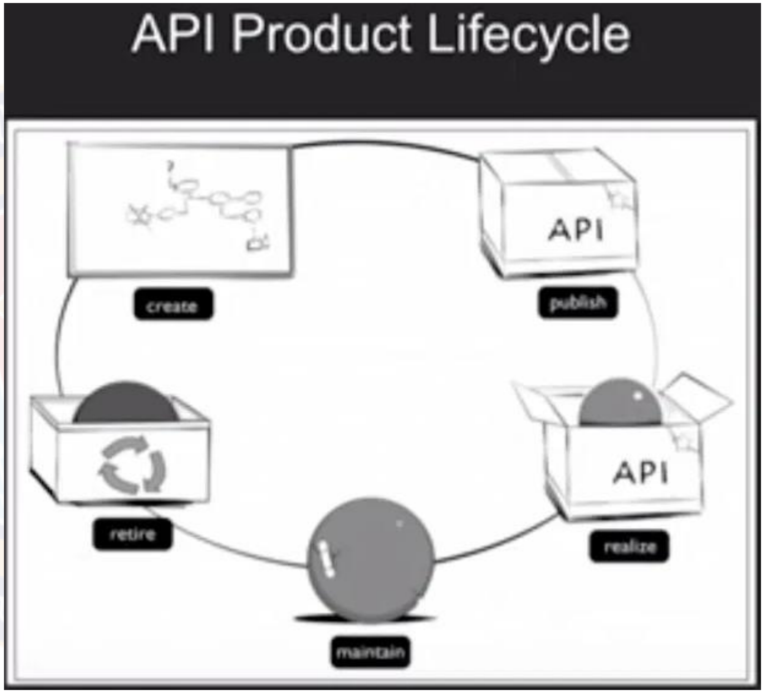
- Answer: c) Cursor pagination requires server-side state, while limit/offset pagination does not. Limit/offset pagination is a simpler pagination method that only requires clients to provide a page number and page size. Cursor pagination, on the other hand, requires the server to maintain state and provide clients with a cursor to the next or previous set of data. While cursor pagination can be more efficient for large datasets, it is more complex and requires more server resources than limit/offset pagination.

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

API Product lifecycle

Five stages of the API product lifecycle

- Similar to any product lifecycle
  - ✓ API products also have lifecycle stages
- Five stages
  - ✓ Create
  - ✓ Publish
  - ✓ Realize
  - ✓ Maintain
  - ✓ Retire
- Product lifecycle is superset of release cycle
  - ✓ Stage many undergo many releases
  - ✓ Release does not result into stage change always



Continuous API Management

Create

- It is important to have an active design process for API products
  - ✓ one that includes a business rationale for the API itself
  - ✓ Designing, implementing, and maintaining APIs in production is a costly endeavor
  - ✓ Doing all that work without a clear use case (and way to measure success) can lead to needless expense

- At the start, APIs go through a design, prototyping, and test phase
  - ✓ APIs might even be “complete,” but they haven’t been released into production
  - ✓ are not considered available for wide use

- While in the create phase,
  - ✓ the interface is likely to change
  - ✓ exhibit errors
  - ✓ and experience periods of non-responsiveness while work is being carried out to handle the edge cases

- Characteristics
  - ✓ A new API or a replacement of API that no longer exists
  - ✓ Has not been deployed in a production environment
  - ✓ Has not been made available for reliable use

Publish

- The API is placed into production and made available to one or more developer communities for use
  - ✓ considered reliable, the interface is stable
  - ✓ the proper security and scalability elements are in place

- Publishing an API is more than just pushing it onto production servers
- Making an API public likely means investing in
  - ✓ documentation
  - ✓ training courses
  - ✓ sales materials
  - ✓ and even staffing up support forums and online communities
- Wide adoption of the API can hinge on whether the publishing effort was adequate for the target community

- Characteristics
  - ✓ API instance has been deployed to a production environment
  - ✓ Made available to one or more developer communities
  - ✓ Strategic value of API is not yet being realized

Realize

- Goes into the phase focused on realizing the initial reason that product was created
  - ✓ might change the interface to better meet goals,
  - ✓ tweak performance, availability, scalability, etc.
  - ✓ All these changes are driven by the desire to realize the initial purpose of the product

- Good APIs solve a problem
- A good API program keeps track of just how well those APIs are doing at solving their stated problems

- The work of realizing an API is
  - ✓ actually the work of “proving” that the API product is doing a good job
  - ✓ solving the problem it was designed to address

- That means need a clear way to track the APIs behavior
  - ✓ compare that to the planned results documented during the create phase

- Characteristics
  - ✓ Published API instance exists and is available
  - ✓ Being used in a way that realizes its objective
  - ✓ Realized value is generally trending upward

Maintain

- Assuming API meets our intended goal
  - ✓ it is realizing it’s planned value
  - ✓ can eventually place it into “maintenance mode”
- Should focus on getting the most value from the product without investing a great deal in changes to it
  - ✓ may do some minor optimizations, look for ways to reduce operational costs, etc.
  - ✓ but it is important to resist making any major changes or costly investments to the product
- Essentially, just counting the repeated incoming revenue (or cost savings) the API was designed to produce
- It is important to identify when the API reaches the maintain phase
  - ✓ since that will affect how you judge whether it is worth it to invest more time and effort into changing it
  - ✓ Placing APIs into maintenance mode means you can dial back on adding new features and focus on improving performance, reducing operational costs, and maximizing return
- Characteristics
  - ✓ Being actively used by one or more consuming applications
  - ✓ Realized value is stagnant or downward trending
  - ✓ No longer actively being improved

Retire

- Finally, as the realized value starts to wane, as the operational costs outweigh the revenue/saving benefits
  - ✓ time to place the product into retirement
- May begin a program to migrate remaining users of this product to a new, more valuable, alternative
  - ✓ or if the service is no longer needed, simply work out an end-of-life plan for the API
  - ✓ begin the “wind-down” process
- Retiring an API frees up resources (infrastructure, staff, support, etc.)
  - ✓ and helps to clear out little-used parts of the ecosystem
- Too easy to forget about actively managing the retirement phase of APIs
  - ✓ But ignoring unused APIs can lead to needless costs in infrastructure and support
  - ✓ means have less money for new, important API products in the future
- Characteristics
  - ✓ Published API instance exists and is available
  - ✓ Realization value is no longer enough to justify continued maintenance
  - ✓ End of life decision has been made

Question

What are some key considerations for managing APIs in a multi-cloud environment, and how can these challenges be addressed?

- a) Ensuring consistent security and access control across multiple cloud providers, and using API gateway solutions to manage and orchestrate API traffic
- b) Maintaining consistent API design and documentation across multiple cloud providers, and using API management platforms to centralize API governance and lifecycle management
- c) Ensuring high availability and performance of APIs across multiple cloud providers, and using load balancing and caching solutions to optimize API traffic
- d) All of the above

Question - Answer

What are some key considerations for managing APIs in a multi-cloud environment, and how can these challenges be addressed?

- a) Ensuring consistent security and access control across multiple cloud providers, and using API gateway solutions to manage and orchestrate API traffic
- b) Maintaining consistent API design and documentation across multiple cloud providers, and using API management platforms to centralize API governance and lifecycle management
- c) Ensuring high availability and performance of APIs across multiple cloud providers, and using load balancing and caching solutions to optimize API traffic
- d) All of the above

- Answer: d) All of the above. Managing APIs in a multi-cloud environment can be challenging, as it requires ensuring consistency and reliability across multiple cloud providers, each with their own unique characteristics and requirements. To address these challenges, API management solutions should focus on maintaining consistent security and access control, API design and documentation, and high availability and performance of APIs across all cloud providers. This can be achieved through the use of API gateway solutions, API management platforms, and load balancing and caching solutions, among other tools and strategies. By addressing these key considerations, organizations can effectively manage APIs in a multi-cloud environment and ensure that they are delivering the best possible user experience to their customers.

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Product Pillars to Lifecycle Mapping

API Pillars

- Each of the API pillar forms a boundary for a work domain
  - ✓ Defines works needs to be done to build and maintain API product
  - ✓ Need to put time and effort in understanding and managing these foundation blocks
  - ✓ Some pillars can be stronger than others in certain phases of product life cycle

- Ten pillars of API
  - ✓ Strategy
  - ✓ Design
  - ✓ Documentation
  - ✓ Development
  - ✓ Testing
  - ✓ Deployment
  - ✓ Security
  - ✓ Monitoring
  - ✓ Discovery
  - ✓ Change Management

Applying product pillars to Lifecycle

- API product lifecycle is useful way of understanding maturity of API product
  - ✓ Helpful in estimating changeability cost of API in each stage
  - ✓ Helps to manage the work that needs to be carried out for APIs

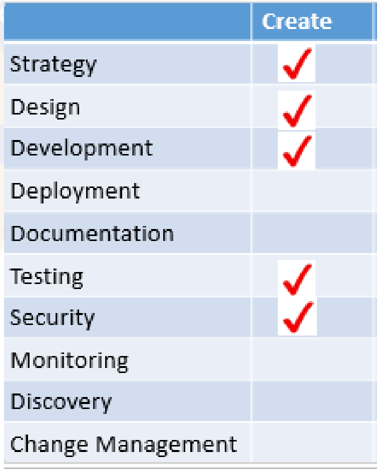
- Ten pillars have significance in every lifecycle stage
  - ✓ Will not be able to ignore them
  - ✓ Not every pillar needs to be considered in every stage of API
  - ✓ Certainly some pillar will dominate a phase as compared to the others
  - ✓ Provides pointers where focus should be kept during that stage of API

Stage 1 : Create

Focus is on developing the best API model before releasing to users

- Special focus on strategy, design, development, testing and security work

- Strategy
  - ✓ Design initial strategy
  - ✓ Test strategy for design and implementation practicality
  - ✓ Update goals and tactics based on feasibility
- Design
  - ✓ Design initial interface model
  - ✓ Test the design from user perspective
  - ✓ Validate implement ability of interface model
- Development
  - ✓ Develop prototypes
  - ✓ Test interface design from implementation perspective
  - ✓ Develop initial implementation of API
- Testing
  - ✓ Define and execute testing strategy for interface model
  - ✓ Define strategy for implementation testing
- Security
  - ✓ Define security requirements
  - ✓ Validate interface design from security perspective

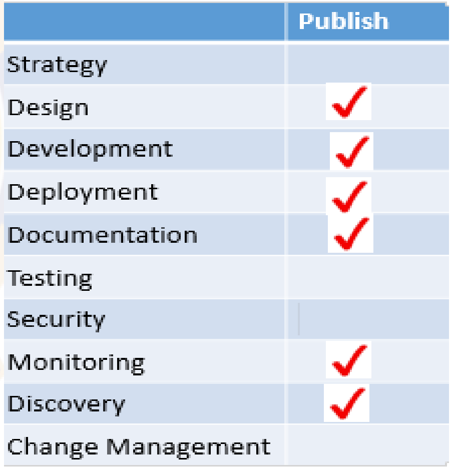


	Create
Strategy	✓
Design	✓
Development	✓
Deployment	
Documentation	
Testing	✓
Security	✓
Monitoring	
Discovery	
Change Management	

Stage 2 : Publish

Door opening for API product – makes API available for consumers

- Focus is on design, development, deployment, documentation, monitoring and discovery
- Design
  - ✓ Analyse the usability of interface
  - ✓ Test design assumptions made in create stage
  - ✓ Improve interface model based on findings
- Development
  - ✓ Optimize implementation for scalability and performance
  - ✓ Optimize implementation for changeability
- Deployment
  - ✓ Deploy API instance
  - ✓ Focus on making API available
  - ✓ Plan and design deployment for future demand
- Documentation
  - ✓ Publish documentation
  - ✓ Improved docs based on actual usage
- Monitoring
  - ✓ Design and implement strategic measures for API
  - ✓ Build monitoring system that can be used during realization
- Discovery
  - ✓ Invest in marketing, engagement and findability of API

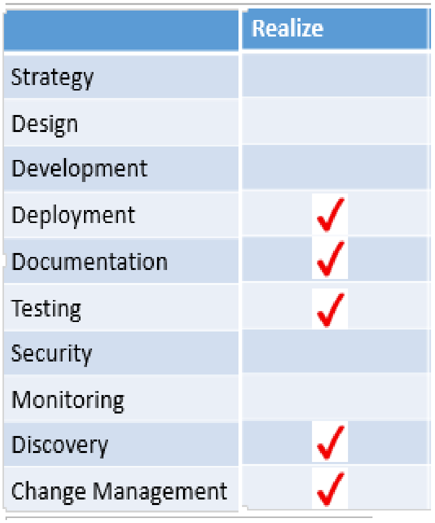


	Publish
Strategy	
Design	✓
Development	✓
Deployment	✓
Documentation	✓
Testing	
Security	
Monitoring	✓
Discovery	✓
Change Management	

Stage 3 : Realize

Goal of API product to reach to this stage – increase value from API

- Focus is on deployment, documentation, testing, discovery and change management
- Deployment
  - ✓ Making sure instances are available
  - ✓ Continually improve and optimize deployment architecture
  - ✓ Improve implementation as necessary
- Documentation
  - ✓ Continue to improve docs
  - ✓ Experiment with additional supporting assets – guides, client libraries, blogs, demos etc
  - ✓ Drive new usage by reducing learning gap
- Testing
  - ✓ Implement testing strategy for interface, implementation and instance changes
  - ✓ Continually improve testing framework
- Discovery
  - ✓ Continue to invest in API marketability, engagement and findability
  - ✓ Invest more in high value user communities
- Change management
  - ✓ Design and implement change management system
  - ✓ Carefully communicate changes to users, maintainers and stakeholders



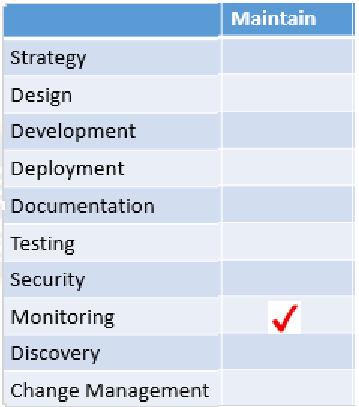
	Realize
Strategy	
Design	
Development	
Deployment	✓
Documentation	✓
Testing	✓
Security	
Monitoring	
Discovery	✓
Change Management	✓

Stage 4 : Maintain

Not getting any new value from API, but don’t want to harm existing users

- Keep the engine running and maintain it
- Most important work is involved in monitoring

- Monitoring
  - ✓ Ensure that monitoring system is operational
  - ✓ Identify patterns that will require special care
  - ✓ Observe metrics that could trigger a retirement decision

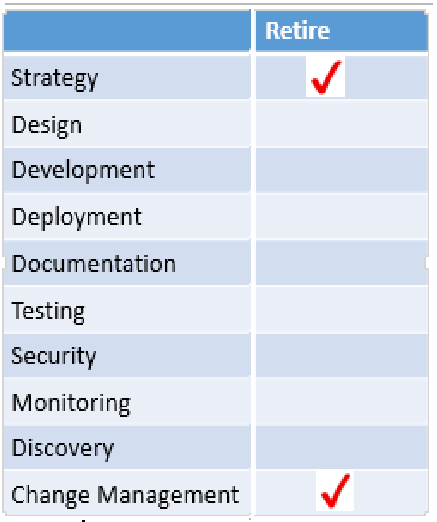


	Maintain
Strategy	
Design	
Development	
Deployment	
Documentation	
Testing	
Security	
Monitoring	✓
Discovery	
Change Management	

Stage 5 : Retire

API not yet gone – needs to be deprecated in planned manner

- Most important pillars are strategy and change management
- Strategy
  - ✓ Define a retirement (or transition) strategy
  - ✓ Identify a new goal, tactical plan and set of actions
  - ✓ Measure progress toward this retirement goal
- Change management
  - ✓ Assess the impact of retiring of API
  - ✓ Design and implement a plan of communication and deprecation
  - ✓ Manage implementation and instance changes to support that deprecation



	Retire
Strategy	✓
Design	
Development	
Deployment	
Documentation	
Testing	
Security	
Monitoring	
Discovery	
Change Management	✓

Summarized

	Create	Publish	Realize	Maintain	Retire
Strategy	✓				✓
Design	✓	✓			
Development	✓	✓			
Deployment		✓	✓		
Documentation		✓	✓		
Testing	✓		✓		
Security	✓				
Monitoring		✓		✓	
Discovery		✓	✓		
Change Management			✓		✓

From Continuous API Management