SYSTEM INTEGRATION AND ARCHITECTURE

What is system integration and when do you need it?

System integration is the process of joining software and hardware modules into one cohesive infrastructure, enabling all pieces to work as a whole. Often referred to as IT integration or software integration, it results in the following advantages:

Increased productivity. Integrated systems allow for centralized control over the daily processes which adds to the efficiency of the entire workflow. A company gets more work done in less time due to the fact that employees can use all apps and data they need from one entry point.

More accurate and trustworthy data. Data is updated across all components of the system simultaneously, keeping all departments on the same page.

Faster decision-making. Data is no longer scattered across siloed storages. So, to perform analytics, you don't need to manually download and export it to the centralized repository. Instead, with a holistic view of all information, you can extract useful business insights to make good decisions more rapidly.

Cost-effectiveness. More often than not, system integration comes at a lower cost than replacing all disjointed parts with a new single system. Not to mention the tricky process of implementing new computer infrastructures.

Common types of system integration

1. **Legacy System Integration**

   Goal: integration of modern applications into existing outdated systems. Many organizations use outdated software to perform their core business functions. It cannot be removed and replaced with more modern technology as it is critical to a company's day-to-day workflow. Instead, legacy systems can be modernized by establishing a communication channel with newer information systems and technology solutions.

   Example: connecting a legacy CRM system to a data warehouse or a transportation management system (TMS).

2. **Enterprise application integration (EAI)**

   Goal: unification of different subsystems inside one business environment While growing, companies incorporate more and more enterprise applications to streamline their front- and back-office processes. These applications often share no points of convergence and accumulate huge volumes of data separately. Enterprise application integration (EAI) brings all functions into one business chain and automates real-time data exchange between different applications.

Example: creating one ecosystem for accounting, human resources information, inventory management, enterprise resource planning (ERP), and CRM systems of a company

3. **Third-party system integration**

   Goal: expanding functionality of the existing system Integration of third-party tools is a great option when your business needs new functionality but can't afford custom software development or just has no time to wait for features to be built from scratch.

   Example: integrating an existing application with online payment systems (PayPal, WebMoney), social media (Facebook, LinkedIn), online video streaming services (YouTube), etc.

4. **Business-to-business integration**

   Goal: connecting systems of two or more organizations Business-to-business or B2B integration automates transactions and document exchange across companies. It leads to more efficient cooperation and trade with suppliers, customers, and partners.

   Example: connecting a retailer's purchasing system to a supplier's ERP system. Whatever the situation, the main objective of system integration is always the same — to put the fragmented and divided pieces together by means of building a coherent network.

Ways to connect systems

1. **Application programming interfaces (APIs)** provide the most common and straightforward way to connect two systems. Sitting between applications and web services, they enable the transmission of data and functionality in a standardized format. Most online service providers — from social media to travel platforms — build external APIs so that clients can easily link to their products.
2. **Middleware** is the hidden software layer that glues together distributed systems, applications, services, and devices. It handles different tasks such as data management, messaging, API management, or authentication. Cloud middleware can be accessed via APIs. In turn, an API gateway can be considered a type of middleware between a collection of services and systems using them.
3. **Webhooks**, also known as HTTP callbacks are real-time messages, sent by one system to another when a certain event happens. Say, accounting software may receive webhook notifications about transactions from payment gateways or online banking systems.
4. **EDI (Electronic Data Interchange)** is the exchange of business information in a standard electronic format that replaces paper documents. EDI generally happens in two ways: via a value added network (VAN), in which a third-party network is in charge of

data transmission, or direct connections through the Internet. All these connectors can be mixed and leveraged when building complex system integrations. If companies have unique needs and requirements for system integration, it's better to opt for custom built solutions whether they are APIs, webhooks, or middleware.

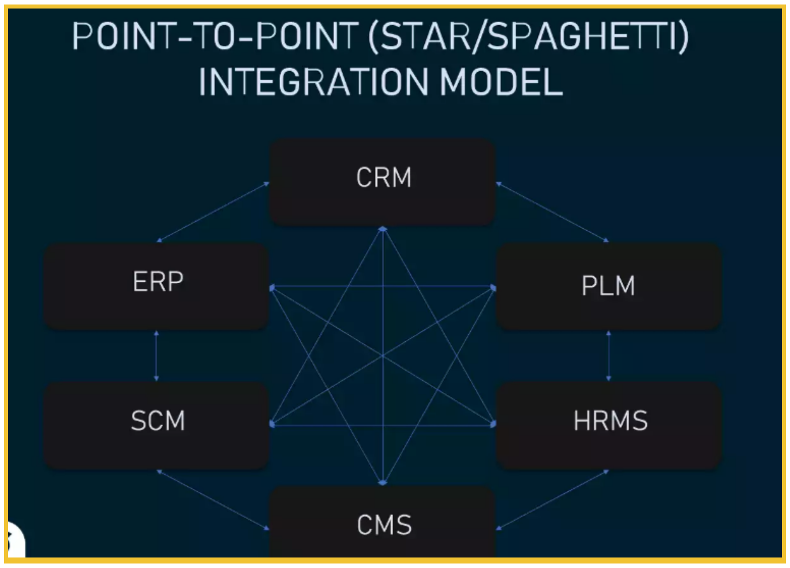
How to approach system integration?

1. Point-to-point model

Point-to-point integration (P2P) is the architectural pattern in which every system is directly connected to all other systems and apps it needs to work in tandem and share information with. This model can be realized via APIs, webhooks, or custom code. With a point-to-point connection, data is extracted from one system, modified or formatted, and then sent to another system. Each application implements all the logic for data translation, transformation, and routing, taking into account the protocols and supported data models of other integrated components.

Pros and cons: Among the main advantages of point-to-point integration is the ability of an IT team to build a small-scale integrated system quite quickly. On the flip side(disadvantage), the model is hard to scale and the management of all the integrations gets very demanding when the number of applications grows. Say, to interconnect six modules you need to perform 15 integrations. This **results in the so-called star/spaghetti integration.**

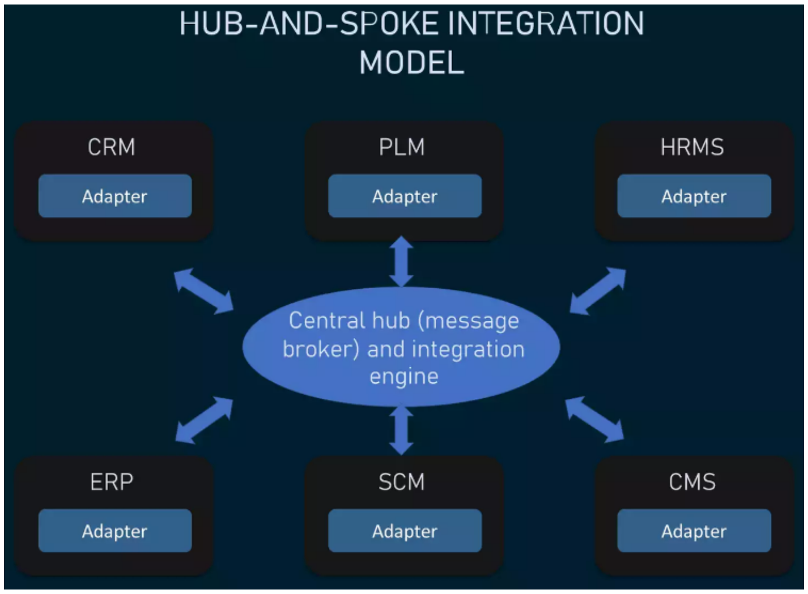
When to use it: This approach suits companies that don't have complex business logic and run their operations on just a few software modules. It is also a perfect option for businesses aiming at connecting to **SaaS applications.**



2. Hub-and-spoke Model

The **hub-and-spoke model** is a more advanced type of integration architecture that addresses the issues of point-to-point and helps to avoid the star/spaghetti mess. The connections between all subsystems are handled by a central hub (message broker), so they don't communicate with each other directly.

The hub serves as a message-oriented middleware with a centralized integration engine to translate operations into a single canonical language and route messages to the right destinations. The spokes (adapters) connecting the hub to the subsystems are managed individually.

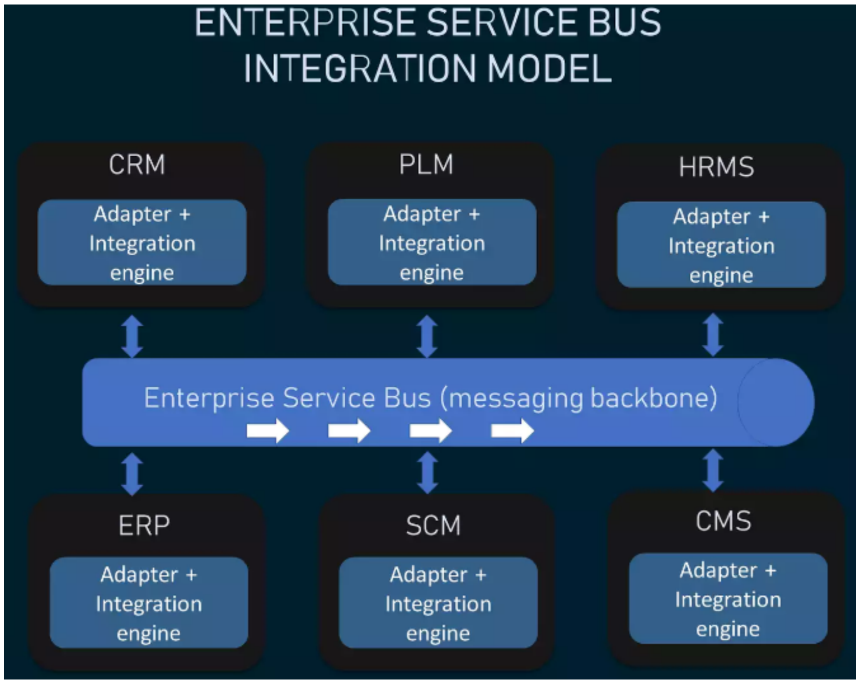

HUB-AND-SPOKE INTEGRATION MODEL
CRM
Adapter
PLM
Adapter
HRMS
Adapter
Central hub (message broker) and integration engine
ERP
Adapter
SCM
Adapter
CMS
Adapter

Pros and cons: As opposed to P2P, the model brings quite a few benefits to the table including higher scalability. Since every system has only one connection to the central hub, things get better in terms of security and architecture simplicity. However, the centralization of the hub can be a weakness in such a model. The whole infrastructure is dependent on the single integration engine which can become the key bottleneck as the workload increases.

When to use it: The hub-and-spoke model is widely-used in e-commerce, financial operations, and payment processing. Besides, it's a preferable architecture for highly regulated industries that face significant security risks.

3. Enterprise Service Bus(ESB) Model

The **ESB architecture** involves the creation of a separate specialized subsystem — an enterprise service bus — that serves as a common user interface layer connecting other subsystems. The ESB can be described as a set of middleware services that glue multiple systems, serving as a messaging backbone. In contrast to hub-and-spoke with a single centralized integration engine, in ESB, each system is supplied with a separate integration engine and an adapter that translates a message into the canonical format and back into the destination supported format. Initially designed to bridge complex internal systems of large enterprises, ESBs can work with cloud services too.


ENTERPRISE SERVICE BUS INTEGRATION MODEL
CRM
Adapter + Integration engine
PLM
Adapter + Integration engine
HRMS
Adapter + Integration engine
Enterprise Service Bus (messaging backbone)
ERP
Adapter + Integration engine
SCM
Adapter + Integration engine
CMS
Adapter + Integration engine

Pros and cons: One of the best things about ESBs is that each subsystem is decoupled by a "messaging bus," so it can be replaced or changed without affecting the functionality of other subsystems. This plays in favor of high scalability. Also, such projects are reliable and quite easy to design. As far as the cons, maintenance and troubleshooting get more complex with the spreading of integration tasks across the systems.
When to use it: An ESB model is an optimal way to implement large projects such as enterprise application integration (EAI), allowing them to scale when needed. It's a good fit if a company needs to bring it together on-premises.

Multitier Architecture

What Is multitier architecture?

In software engineering, multitier architecture (often referred to as n-tier architecture) is a client–server architecture in which presentation, application processing and data management functions are physically separated. The most widespread use of multitier architecture is the three-tier architecture.

N-tier application architecture provides a model by which developers can create flexible and reusable applications. By segregating an application into tiers, developers acquire the option of modifying or adding a specific tier, instead of reworking the entire application. A three-tier architecture is typically composed of a presentation tier, a logic tier, and a data tier.

Three-tier architecture

Three-tier architecture is a client-server software architecture pattern in which the user interface (presentation), functional process logic ("business rules"), computer data storage and data access are developed and maintained as independent modules, most often on separate platforms. It was developed by John J. Donovan in Open Environment Corporation (OEC), a tools company he founded in Cambridge, Massachusetts.

Apart from the usual advantages of modular software with well-defined interfaces, the three-tier architecture is intended to allow any of the three tiers to be upgraded or replaced independently in response to changes in requirements or technology. For example, a change of operating system in the presentation tier would only affect the user interface code.

Typically, the user interface runs on a desktop PC or workstation and uses a standard graphical user interface, functional process logic that may consist of one or more separate modules running on a workstation or application server, and an RDBMS on a database server or mainframe that contains the computer data storage logic. The middle tier may be multitiered itself (in which case the overall architecture is called an "n-tier architecture").

Presentation tier

This is the topmost level of the application. The presentation tier displays information related to such services as browsing merchandise, purchasing and shopping cart contents. It communicates with other tiers by which it puts out the results to the browser/client tier and all other tiers in the network. In simple terms, it is a layer that users can access directly (such as a web page, or an operating system's GUI).

Application tier (business logic, logic tier, or middle tier)

The logical tier is pulled out from the presentation tier and, as its layer, it controls an application’s functionality by performing detailed processing.

Data Tier

The data tier includes the data persistence mechanisms (database servers, file shares, etc.) and the data access layer that encapsulates the persistence mechanisms and exposes the data. The data access layer should provide an API to the application tier that exposes methods of managing the stored data without exposing or creating dependencies on the data storage mechanisms. Avoiding dependencies on the storage mechanisms allows for updates or changes without the application tier clients being affected by or even aware of the change. As with the separation of any tier, there are costs for implementation and often costs to performance in exchange for improved scalability and maintainability.

Web development usage

In the web development field, three-tier is often used to refer to websites, commonly electronic commerce websites, which are built using three tiers:
A **front-end** web server serving static content, and potentially some cached dynamic content. In web based application, front end is the content rendered by the browser. The content may be static or generated dynamically.
A **middle dynamic content processing and generation level application server** (e.g., Symfony, Spring, ASP.NET, Django, Rails, Node.js).
A **back-end database or data store,** comprising both data sets and the database management system software that manages and provides access to the data.

MONOLITHIC ARCHITECTURE

What Is monolithic architecture?

A monolithic architecture is the traditional unified model for the design of a software program. Monolithic, in this context, means "composed all in one piece." According to the Cambridge dictionary,the adjective monolithic also means both "too large" and "unable to be changed."

Monolithic architecture for software

Monolithic software is designed to be self-contained, wherein the program's components or functions are tightly coupled rather than loosely coupled, like in modular software programs. In a monolithic architecture, each component and its associated components must all be present for code to be executed or compiled and for the software to run.

Monolithic applications are single-tiered, which means multiple components are combined into one large application. Consequently, they tend to have large codebases, which can be cumbersome to manage over time.

Furthermore, if one program component must be updated, other elements may also require rewriting, and the whole application has to be recompiled and tested. The process can be time-consuming and may limit the agility and speed of software development teams. Despite these issues, the approach is still in use because it does offer some advantages. Also, many early applications were developed as monolithic software, so the approach cannot be completely disregarded when those applications are still in use and require updates.

Understanding monolithic architecture with an example

If the application uses a monolithic architecture, it is built and deployed as a single application, regardless of how a customer uses it. Thus, whether users access the application from their desktop or from a mobile device, the application remains tightly coupled, and all the various components and modules are directly connected to each other. It may also use a relational database management system as a single data source. Finally, if changes are needed for any one component, code changes are required for all other affected components as well.

Key components of monolithic applications

Monolithic applications typically consist of multiple components that are interconnected to form one large application. These components may include these features:

Authorization. To authorize a user and allow them to use the application.
Presentation. To handle Hypertext Transfer Protocol requests and respond with Hypertext Markup Language, Extensible Markup Language or JavaScript Object Notation.
Business logic. The underlying business logic that drives the application's functionality and features.
Database layer. Includes the data access objects that access the application's database.
Application integration. Controls and manages the application's integration with other services or data sources. Some applications may also include a notification module to control and send automated email communications to users.

Benefits of monolithic architecture

There are benefits to monolithic architectures, which is why many applications are still created using this development paradigm. For one, monolithic programs may have better throughput than modular applications. They may also be easier to test and debug because, with fewer elements, there are fewer testing variables and scenarios that come into play.

At the beginning of the software development lifecycle, it is usually easier to go with the monolithic architecture since development can be simpler during the early stages. A single codebase also simplifies logging, configuration management, application performance monitoring and other development concerns. Deployment can also be easier by copying the packaged application to a server. Finally, multiple copies of the application can be placed behind a load balancer to scale it horizontally.

That said, the monolithic approach is usually better for simple, lightweight applications. For more complex applications with frequent expected code changes or evolving scalability requirements, this approach is not suitable.

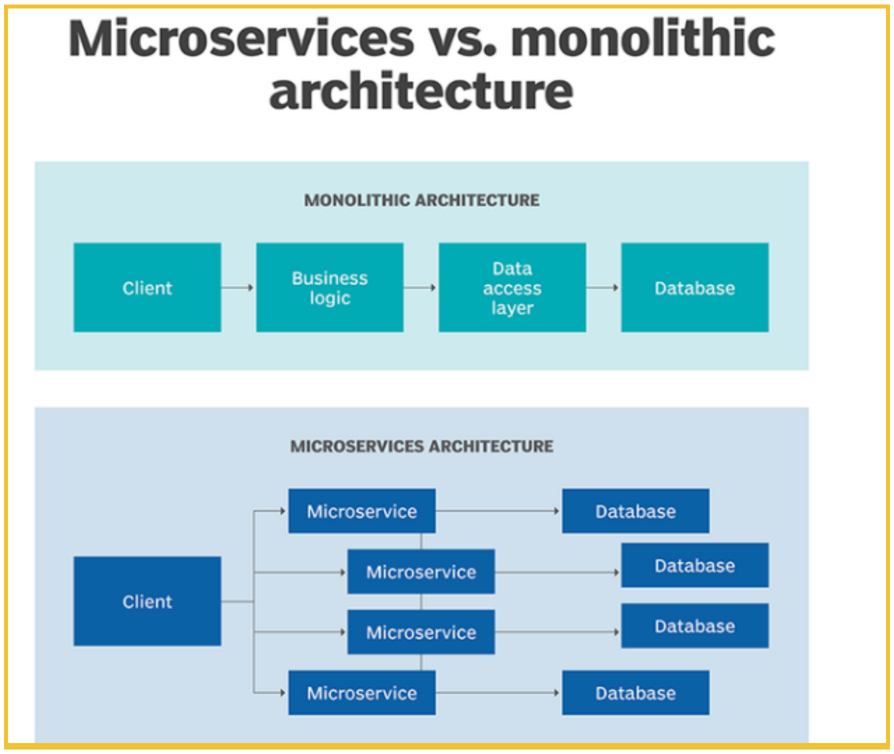
Drawbacks of monolithic architecture

Generally, monolithic architectures suffer from drawbacks that can delay application development and deployment. These drawbacks become especially significant when the product's complexity increases or when the development team grows in size. The codebase of monolithic applications can be difficult to understand because they may be extensive, which can make it difficult for new developers to modify the code to meet changing business or technical requirements. As requirements evolve or become more complex, it becomes difficult to correctly implement changes without hampering the quality of the code and affecting the overall operation of the application.

The application's size can also increase startup time and add to delays. In some cases, different parts of the application may have conflicting resource requirements. This makes it harder to find the resources required to scale the application.

In addition to limited scalability, reliability is another concern with monolithic software. A bug in any one component can potentially bring down the entire application. Considering the banking application example, suppose there's a memory leak in the user authorization module. This bug can bring the entire application down and make it unavailable to all users.

Finally, by virtue of their size and complexity, monolithic applications are not particularly adaptable to new technologies. A new development framework or language can affect the application as a whole, so adopting it can be both time-consuming and costly. Small organizations or companies on tight budgets may not have the funds or staff available to update the application, so they may end up maintaining the status quo, potentially leaving them unable to take advantage of a new language or framework.



REST APIs

What IsARESTAPI?

An API is **an application programming interface**. It is a set of rules that allow programs to talk to each other. The developer creates the API on the server and allows the client to talk to it.

REST determines how the API looks like. It stands for "Representational State Transfer". It is a set of rules that developers follow when they create their API. One of these rules states that you should be able to get a piece of data (called a resource) when you link to a specific URL.

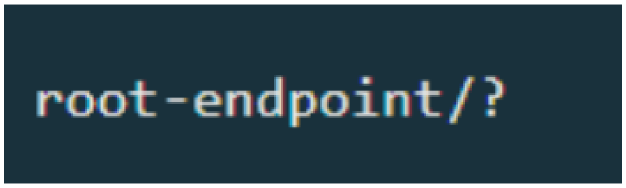
Each **URL is called a request** while the data sent back to you is called a response.

The Anatomy Of A Request

It's important to know that a request is made up of four things:

1. The endpoint
2. The method
3. The headers
4. The data (or body)

The endpoint (or route) is the url you requested for. Itfollows this structure:



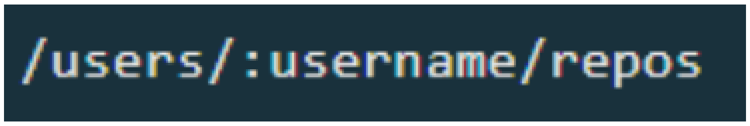
```
root-endpoint/?
```

The **root-endpoint** is the starting point of the API you're requesting from. The root-endpoint of Github's API is https://api.github.com while the root-endpoint Twitter's API is https://api.twitter.com.

The path determines the resource you're requesting for. Think of it like an automatic answering machine that asks you to press 1 for a service, press 2 for another service, 3 for yet another service and so on.

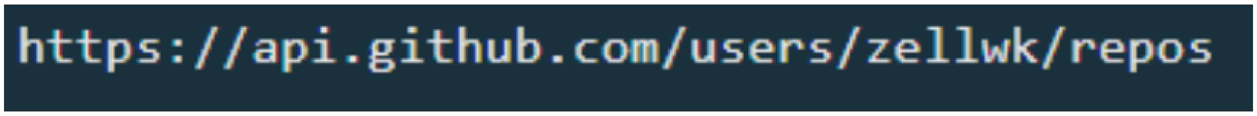
You can access paths just like you can link to parts of a website. For example, to get a list of all posts tagged under "JavaScript" on Smashing Magazine, you navigate to https://www.smashingmagazine.com/tag/javascript/. https://www.smashingmagazine.com/ is the root endpoint and /tag/javascriptis the path.

To understand what paths are available to you, you need to look through the API documentation. For example, let's say you want to get a list of repositories by a certain user through Github's API. The docs tells you to use the the following path to do so:



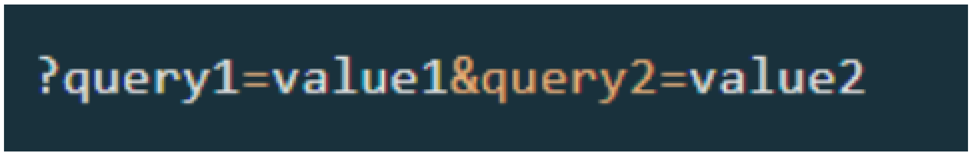
```
/users/:username/repos
```

Any colons (:) on a path denotes a variable. You should replace these values with actual values of when you send your request. In this case, you should replace :username with the actual username of the user you're searching for. If I'm searching for my Github account, I'll replace :username with zellwk.



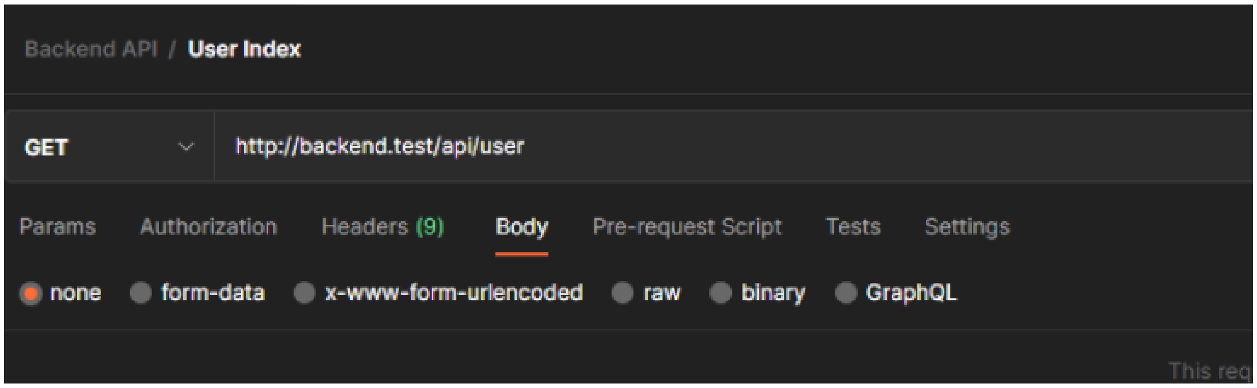
```
https://api.github.com/users/zellwk/repos
```

The final part of an endpoint is query parameters. Technically, query parameters are not part of the REST architecture, but you'll see lots of APIs use them. So,to help you completely understand how to read and use API's we're also going to talk about them. Query parameters give you the option to modify your request with key-value pairs. They always begin with a question mark (?). Each parameter pair is then separated with an ampersand (&), like this:



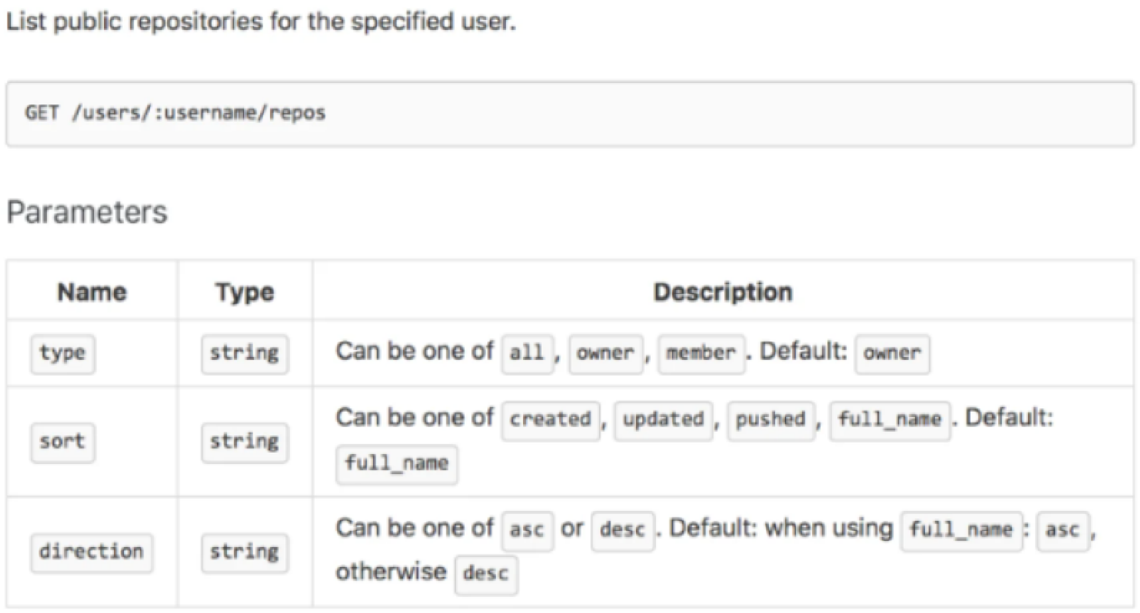
```
?query1=value1&query2=value2
```

Endpoint in using postman



```
http://backend.test/api/user
```

When you try to get a list of a user's repositories on Github, you add three possible parameters to your request to modify the results given to you:



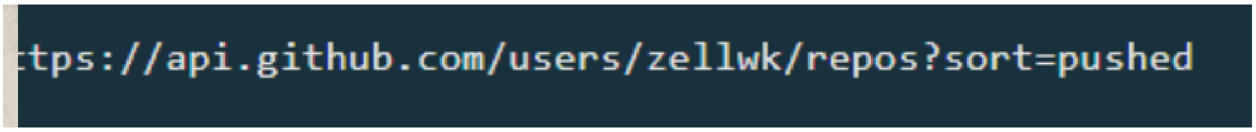
List public repositories for the specified user.

```
GET /users/:username/repos
```

Parameters

Name	Type	Description
`type`	`string`	Can be one of `all`, `owner`, `member`. Default: `owner`
`sort`	`string`	Can be one of `created`, `updated`, `pushed`, `full_name`. Default: `full_name`
`direction`	`string`	Can be one of `asc` or `desc`. Default: when using `full_name`: `asc`, otherwise `desc`

If you'd like to get a list of the repositories that I pushed to recently, you can set sort to push.

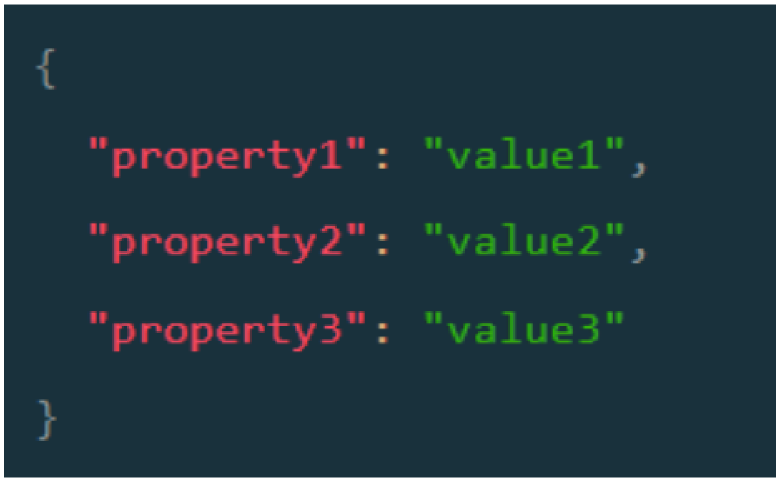


```
tps://api.github.com/users/zellwk/repos?sort=pushed
```

JSON (API response)

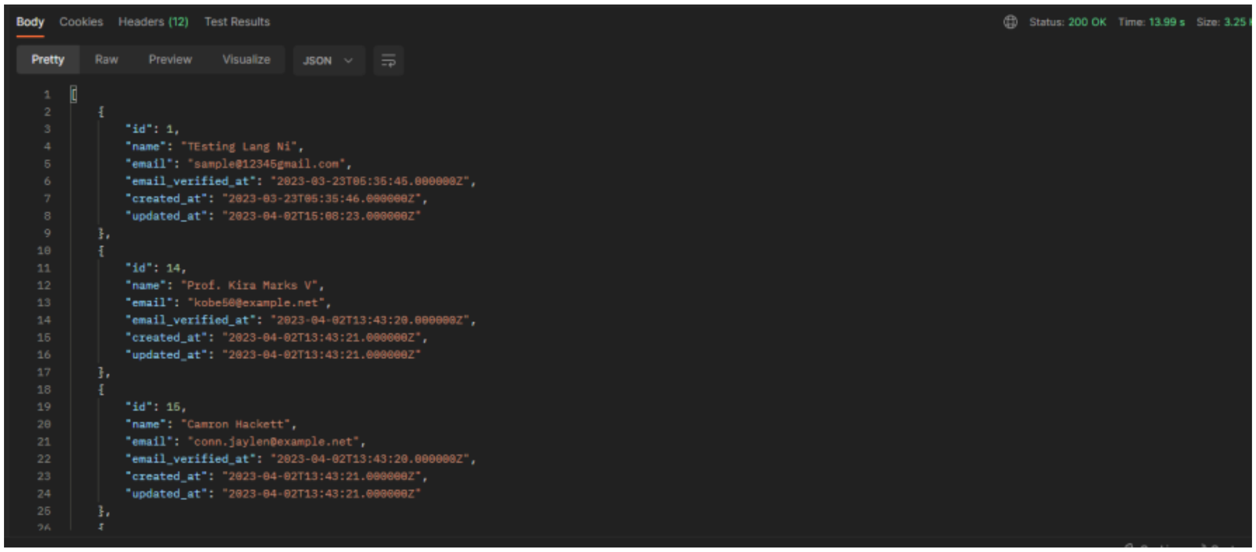
JSON (JavaScript Object Notation) a common format for sending and requesting data through a REST API. The response that Github sends back to you is also formatted as JSON.

A JSON object looks like a JavaScript Object. In JSON, each property and value must be wrapped with double quotation marks, like this:



```
{
   "property1": "value1",
   "property2": "value2",
   "property3": "value3"
}
```

Json response in postman



```
[
    {
        "id": 1,
        "name": "TEsting Lang Ni",
        "email": "sample@12345gmail.com",
        "email_verified_at": "2023-03-23T05:35:45.000000Z",
        "created_at": "2023-03-23T05:35:46.000000Z",
        "updated_at": "2023-04-02T15:08:23.000000Z"
    },
    {
        "id": 14,
        "name": "Prof. Kira Marks V",
        "email": "kobe50@example.net",
        "email_verified_at": "2023-04-02T13:43:20.000000Z",
        "created_at": "2023-04-02T13:43:21.000000Z",
        "updated_at": "2023-04-02T13:43:21.000000Z"
    },
    {
        "id": 15,
        "name": "Camron Hackett",
        "email": "conn.jaylen@example.net",
        "email_verified_at": "2023-04-02T13:43:20.000000Z",
        "created_at": "2023-04-02T13:43:21.000000Z",
        "updated_at": "2023-04-02T13:43:21.000000Z"
    },
    {
```

Back To TheAnatomy Of A Request

You’ve learned that a request consists of four parts.

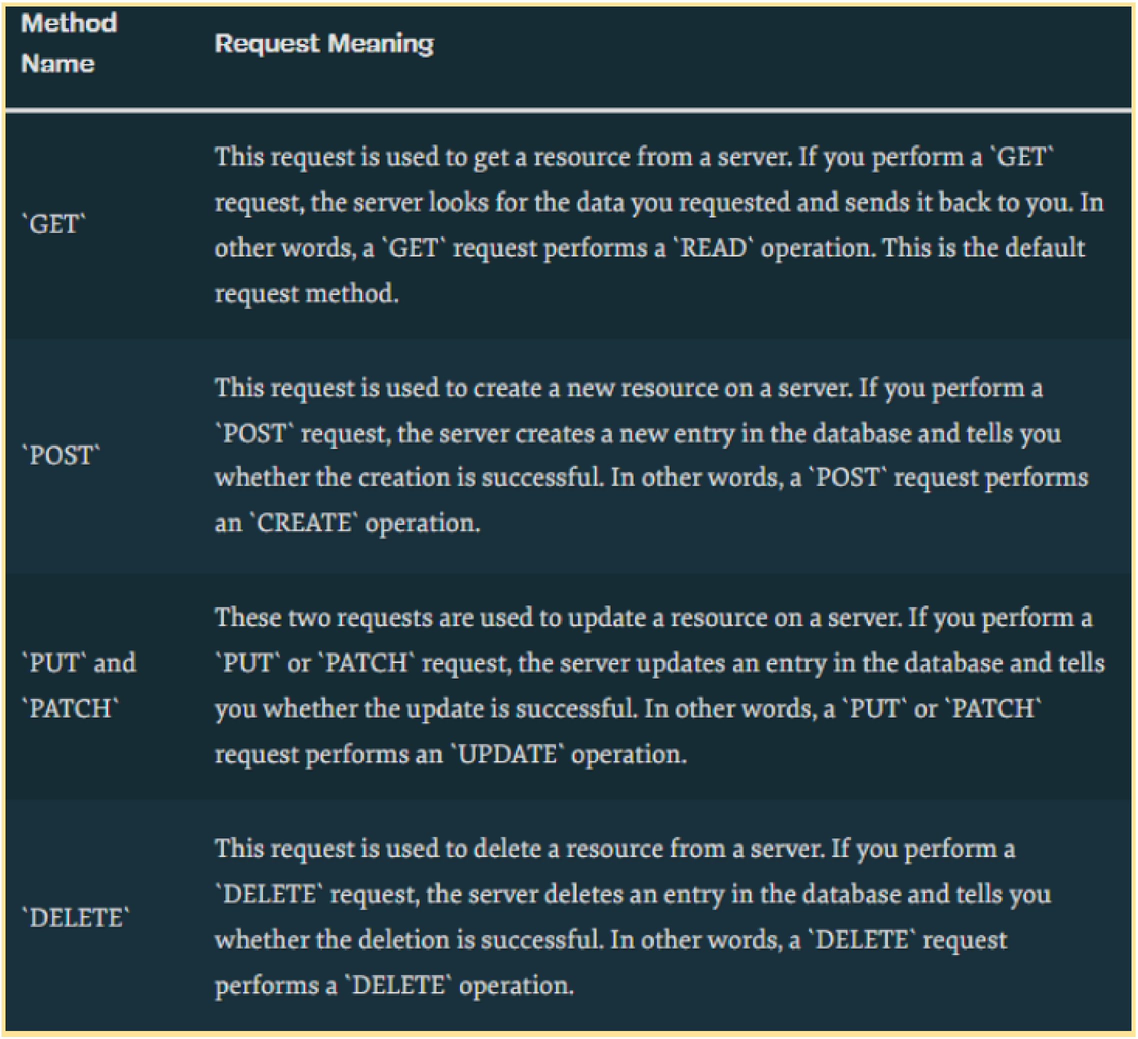
1. The endpoint
2. The method
3. The headers
4. The data (or body)

The Method

The method is the type of request you send to the server. You can choose from these five types below:

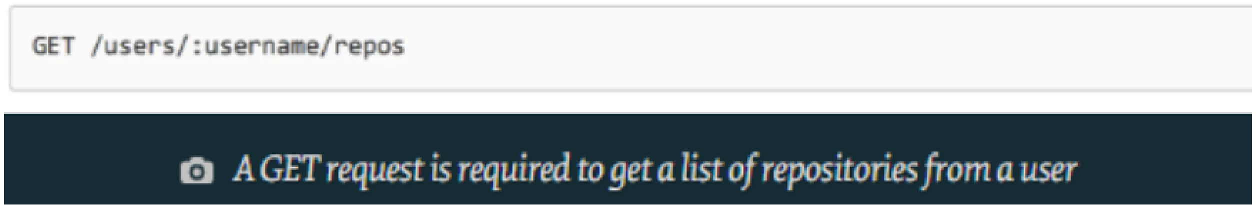
1. GET
2. POST
3. PUT
4. PATCH
5. DELETE

These methods provide meaning for the request you're making. They are used to perform four possible actions: Create, Read, Update and Delete (CRUD).



Method Name	Request Meaning
\`GET\`	This request is used to get a resource from a server. If you perform a \`GET\` request, the server looks for the data you requested and sends it back to you. In other words, a \`GET\` request performs a \`READ\` operation. This is the default request method.
\`POST\`	This request is used to create a new resource on a server. If you perform a \`POST\` request, the server creates a new entry in the database and tells you whether the creation is successful. In other words, a \`POST\` request performs an \`CREATE\` operation.
\`PUT\` and \`PATCH\`	These two requests are used to update a resource on a server. If you perform a \`PUT\` or \`PATCH\` request, the server updates an entry in the database and tells you whether the update is successful. In other words, a \`PUT\` or \`PATCH\` request performs an \`UPDATE\` operation.
\`DELETE\`	This request is used to delete a resource from a server. If you perform a \`DELETE\` request, the server deletes an entry in the database and tells you whether the deletion is successful. In other words, a \`DELETE\` request performs a \`DELETE\` operation.

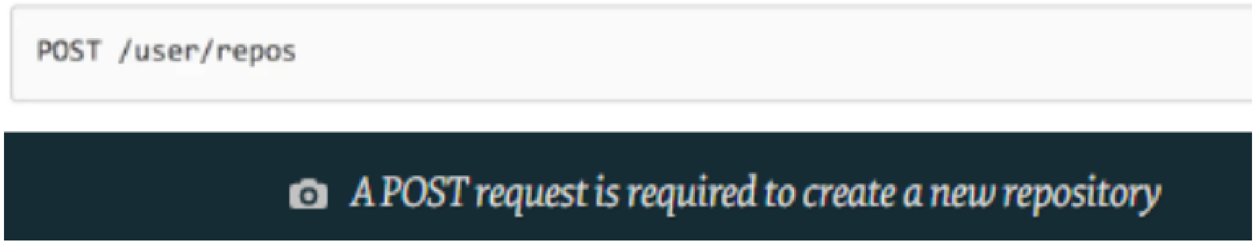
The API lets you know what request method to use each request. For example, to get a list of a user’s repositories, you need a GET request:



```
GET /users/:username/repos
```

A GET request is required to get a list of repositories from a user

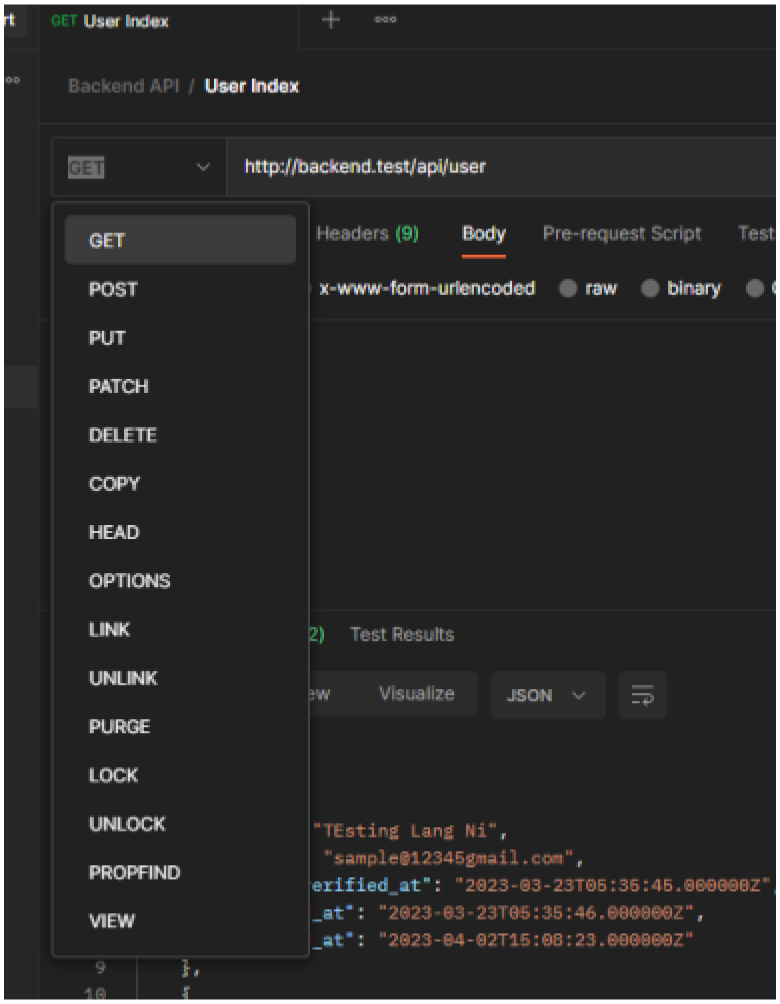
A GET request is required to get a list of repositories from a user. To create a new Github repository, you need a POST request



```
POST /user/repos
```

A POST request is required to create a new repository

List of methods in postman



```
"TEsting Lang Ni",
 "sample@12345gmail.com",
erified_at": "2023-03-23T05:35:45.000000Z"
_at": "2023-03-23T05:35:46.000000Z",
_at": "2023-04-02T15:08:23.000000Z"
},
{
```

The Headers

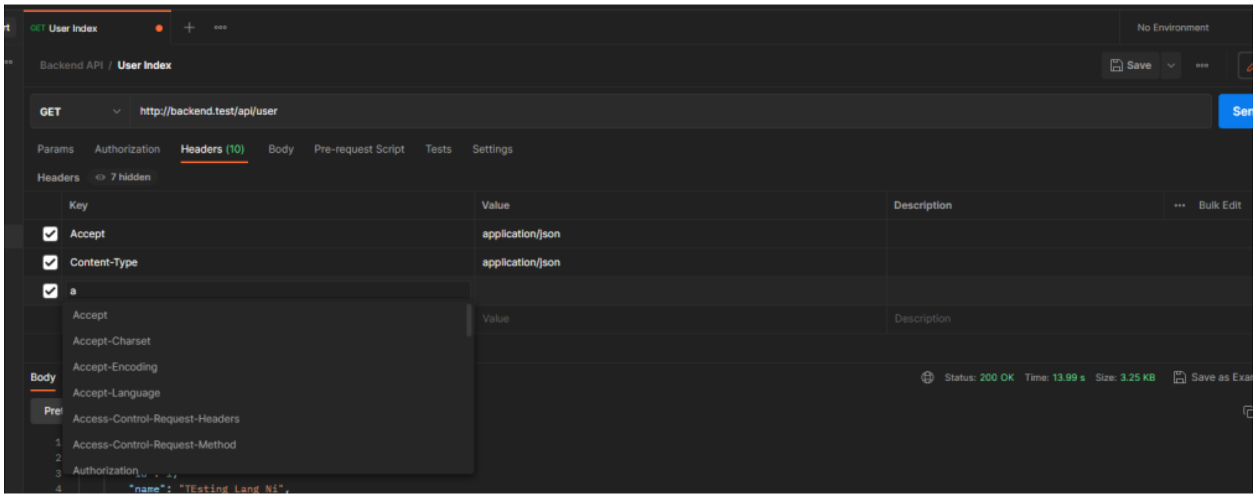
Headers are used to provide information to both the client and server. It can be used for many purposes, such as authentication and providing information about the body content. You can find a list of valid headers on MDN's HTTP Headers Reference.

```
"Content-Type: application/json". Missing the opening ".
```

You can send HTTP headers with curl through the -H or --header option. To send the above header to Github's API, you use this command:

```
-H "Content-Type: application/json" https://api.github.com
```

Setting Headers in postman



```
GET http://backend.test/api/user
```

Key	Value	Description
✓ Accept	application/json	
✓ Content-Type	application/json	
✓ a		
	Value	Description

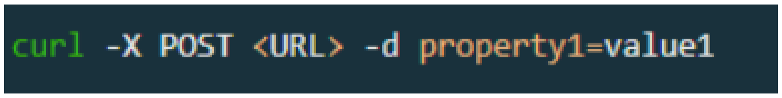
Status: 200 OK Time: 13.99 s Size: 3.25 KB

```
"name": "TEsting Lang Ni",
```

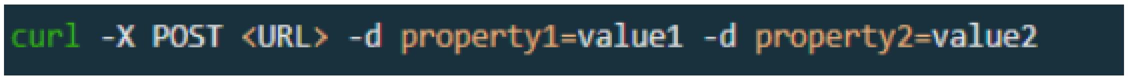
The Data (Or "Body")

The data (sometimes called "body" or "message") contains information you want to be sent to the server. This option is only used with POST, PUT, PATCH or DELETE requests.

To send data through cURL, you can use the -d or --data option:

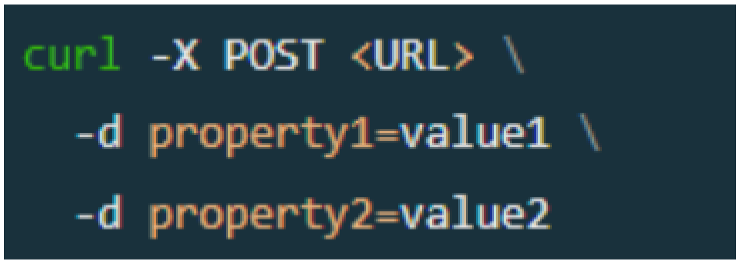


```
curl -X POST <URL> -d property1=value1
```



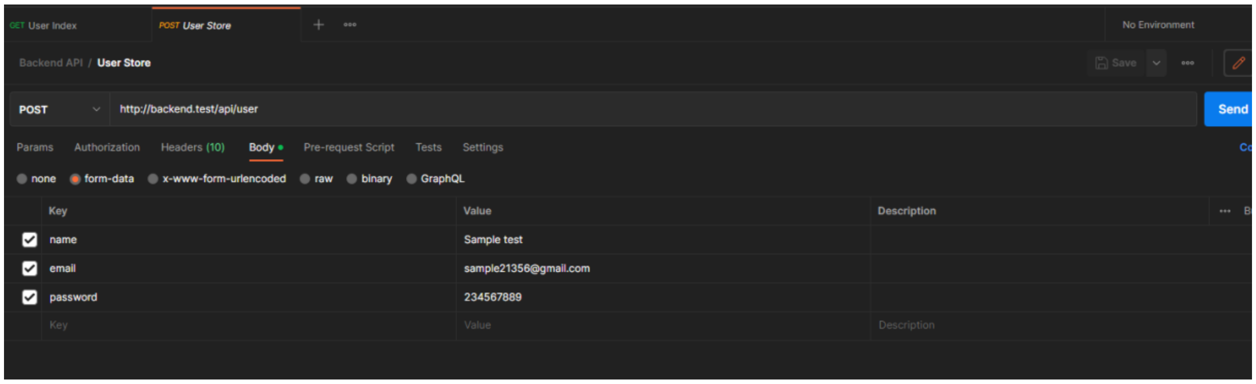
```
curl -X POST <URL> -d property1=value1 -d property2=value2
```

If it makes sense, you can break your request into multiple lines \to make it easier to read.



```
curl -X POST <URL> \
  -d property1=value1 \
  -d property2=value2
```

The Data (Or “Body”) Using postman



POST http://backend.test/api/user

Key	Value	Description
✓ name	Sample test	
✓ email	sample21356@gmail.com	
✓ password	234567889	
Key	Value	Description

Authentication

You wouldn’t allow anyone to access your bank account without your permission, would you? On the same line of thought, developers put measures in place to ensure you perform actions only when you’re authorized to do so. This prevents others from impersonating you.

Since POST, PUT, PATCH and DELETE requests alter the database, developers almost always put them behind an authentication wall. In some cases, a GET request also requires authentication (like when you access your bank account to check your current balance, for example).

On the web,there are two main ways to authenticate yourself:

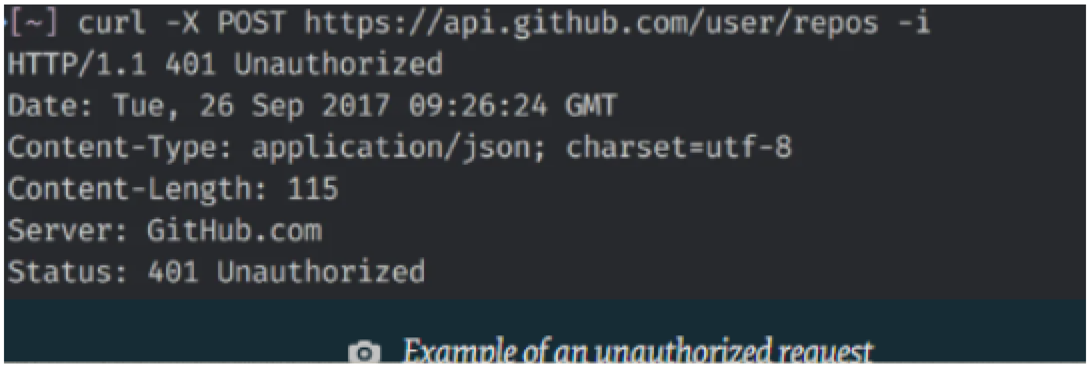
1. With a username and password (also called basic authentication)
2. With a secret token

HTTP Status Codes And Error Messages

Some of the messages you’ve received earlier, like “Requires authentication” and “Problems parsing JSON” are error messages. They only appear when something is wrong with your request. HTTP status codes let you tell the status of the response quickly. The range from 100+ to 500+. In general, the numbers follow the following rules:

1. 200+ means the request has succeeded.
2. 300+ means the request is redirected to another URL
3. 400+ means an error that originates from the client has occurred
4. 500+ means an error that originates from the server has occurred

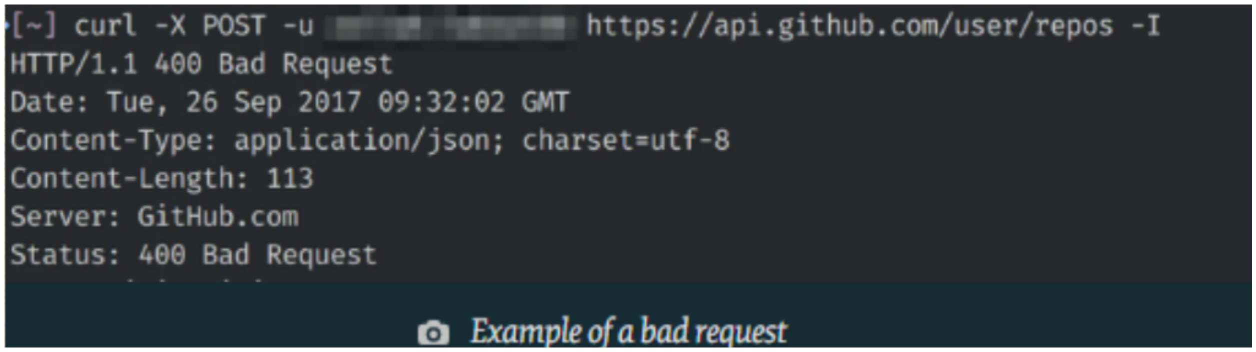
For example, if you tried adding -Ito a POST request without providing your username and password, you'll get a 401 status code (Unauthorized):



```
[~] curl -X POST https://api.github.com/user/repos -i
HTTP/1.1 401 Unauthorized
Date: Tue, 26 Sep 2017 09:26:24 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 115
Server: GitHub.com
Status: 401 Unauthorized
```

Example of an unauthorized request

If your request is invalid because your data is wrong or missing, you usually get a 400 status code (Bad Request).



```
[~] curl -X POST -u [redacted] https://api.github.com/user/repos -I
HTTP/1.1 400 Bad Request
Date: Tue, 26 Sep 2017 09:32:02 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 113
Server: GitHub.com
Status: 400 Bad Request
```

Example of a bad request

API Versions

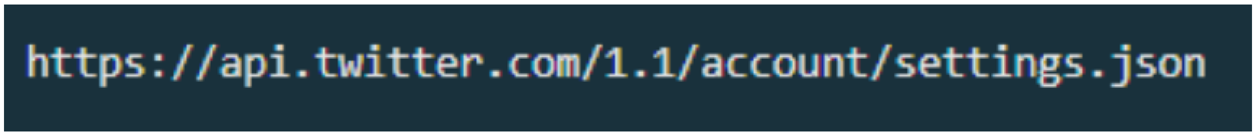
Developers update their APIs from time to time. Sometimes, the API can change so much that the developer decides to upgrade their API to another version. If this happens, and your application breaks, it's usually because you've written code for an older API, but your request points to the newer API.

You can request for a specific API version in two ways. Which way you choose depends on how the API is written.

These two ways are:

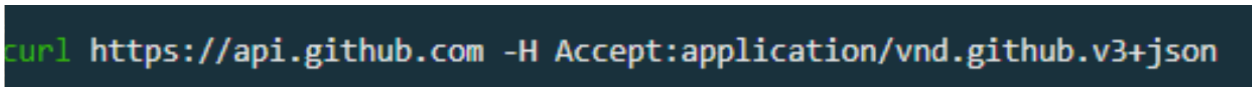
1. Directly in the endpoint
2. In a request header

Twitter, for example, uses the first method. At The time of writing, Twitter's API is at version 1.1, which is evident through its endpoint:



```
https://api.twitter.com/1.1/account/settings.json
```

Github, on the other hand, uses the second method. At The time of writing, Github’s API is at version 3, and you can specify the version with an Accept header:



```
curl https://api.github.com -H Accept:application/vnd.github.v3+json
```

XHR vs fetchAPI vsAjax vsAxios forAPI Requests

HOW DO WE USE API?

API’s work on the CRUD principle. CRUD stands for Create, Read, Update, and Delete. But put more simply, in regards to its use in RESTful APIs, CRUD is the standardized use of HTTP Action Verbs. This means that if you want to create a new record you should be using “POST. ” If you are trying to read a record, you should be using “GET. ” To update a record utilizing "PUT " or "PATCH". And to delete a record, using “DELETE. ”

The most commonly used API technologies are XMLHttpRequest, Ajax, FetchAPI, and Axios. Let us see how each of these works.

XMLHttpRequest

XMLHttpRequest method is the Godfather of all API technologies. It has been in existence ever since the firstInternet Explorer was launched in the year 2000.

XMLHttpRequest (XHR) objects are used to interact with servers. You can retrieve data from a URL without having to do a full page refresh. This enables a Web page to update just part of a page without disrupting what the user is doing.

Even though the name suggests otherwise it can handle all types of data.

To send an HTTP request, create an XMLHttpRequest object, open a URL, and send the request. After the transaction completes, the object will contain useful information such as the response body and the HTTP status of the result.

```
function reqListener () {
console.log(this.responseText);
}
var oReq = new XMLHtttpRequest();
oReq.addEventListener(“load”, reqListener);
oReq.open(“GET”, “http://www.example.org/example.txt");
oReq.send();
```

A request made via XMLHttpRequest can fetch the data in one of two ways, asynchronously or synchronously. The type of request is dictated by the optional async argument (the third argument) that is set on the XMLHttpRequest.open() method. If this argument is true or not specified, the XMLHttpRequest is processed asynchronously, otherwise, the process is handled synchronously.

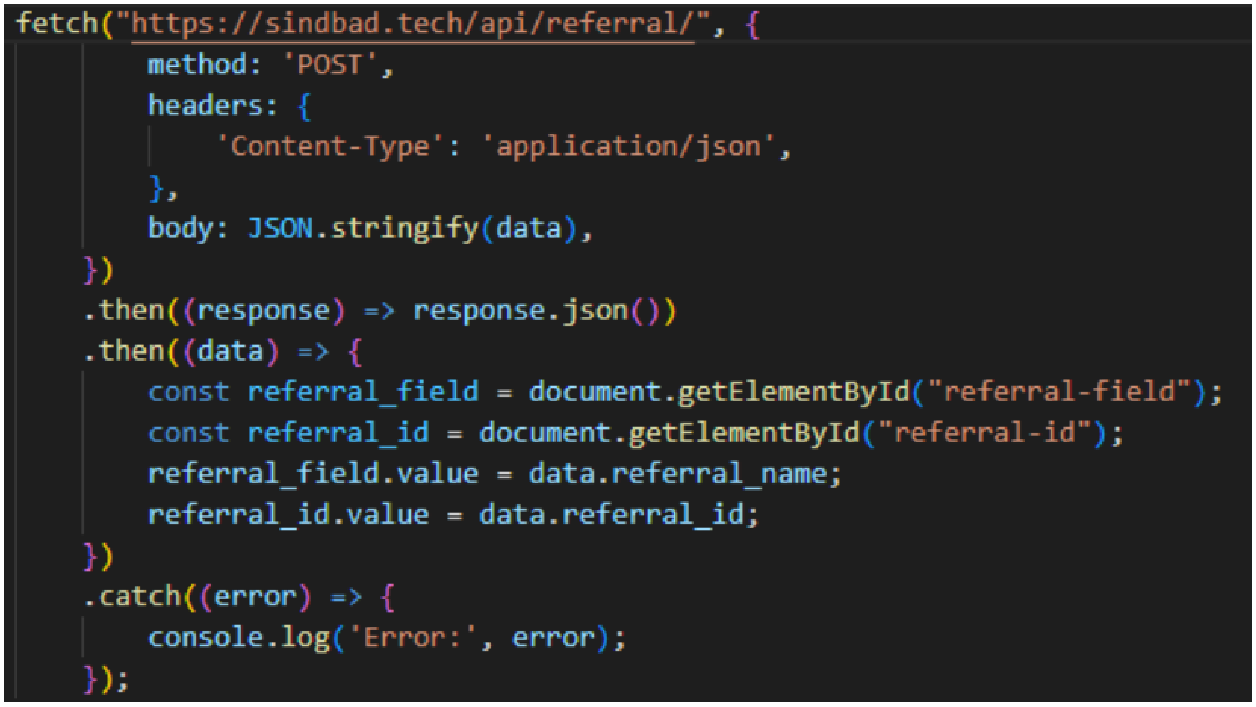
FetchAPI

fetch() allows you to make network requests similar to XMLHttpRequest (XHR). The main difference is that the Fetch API uses Promises, which enables a simpler and cleaner API, avoiding callback hell and having to remember the complex API of XMLHttpRequest.

1. Created on xhr with built-in promises.
2. Compact syntax than xhr.

The below-given code snippet showcases the syntax of a simple fetch() request.

```
parameter
.then(function() {
// Your code for handling the data you get from the API
})
.catch(function() {
// This is where you run code if the server returns any errors
```



```
fetch("https://sindbad.tech/api/referral/", {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
        },
        body: JSON.stringify(data),
    })
    .then((response) => response.json())
    .then((data) => {
        const referral_field = document.getElementById("referral-field");
        const referral_id = document.getElementById("referral-id");
        referral_field.value = data.referral_name;
        referral_id.value = data.referral_id;
    })
    .catch((error) => {
        console.log('Error:', error);
    });
```

Working with fetch():

The Fetch API allows you to asynchronously request a resource.

Use the fetch() method to return a promise that resolves into a Response object. To get the actual data, you call one of the methods of the Response object e.g.,text() or json(). These methods also resolve into the actual data.

Use the status and statusText properties of the Response object to get the status and status text of the response.

Use the catch() method or try…catch statement to handle a failure request.

jQueryAJAX

"Ajax" stands for **Asynchronous Javascript And XML**, jQuery provides a condensed format to make XMLHTTPRequest. AJAX is the art of exchanging data with a server and updating parts of a web page — without reloading the whole page.

1. Cross-browser support
2. Simple methods to use
3. Define the type of request

jQuery ajax base syntax:

```
$.ajax({name:value, name:value, … })
```

$.ajax makes the call to ajax,then the methods are called in place of a name and the callbacks as value, as the example given below.

jQuery ajax Example:

```
$.ajax({
url: “demo_test.txt”,
type: “get”,
success: function(result){
result = JSON.parse(result)
console.log(result);
}});
```

$(“button”).click(function()

The onclick listener is added to the button tag.

$.ajax({

url: “demo_url”,

type: “get”,

success: function(result){

The ajax function uses xmlHTTPRequest to make a get request to the specified URL:“demo_url” , and the response is stored in the result.

```
success: function(result){

result = JSON.parse(result)

console.log(result);

}});
```

The response is received in a JSON format here so we first need to convert it into a javascript object using JSON.parse() function, now we can use the response obtained from the request as a regular javascript object, it can be logged into the console to test.

Axios

Axios is a popular promise-based 3rd party library client that has an easy-to-use API and can be used in both browser and node.js, it supports the promise API, also supports the feature to cancel requests and automatically transform JSON data.

1. has a way to set a response timeout
2. has a way to cancel a request
3. performs automatic JSON data transformation
4. supports upload progress

```
=>axios.get('http://www.example.org/example.txt').then(
function(response) {
//your code for handling API data
console.log(response)
}).catch(
function(err) {
//your code for handling API error
console.log(err);
});
document.getElementById('get-button').addEventListener('click',
getData());
```

The Axios request is being stored in a const getData function which can be activated on an eventlistener.

Conclusion

When it comes to ease of use and syntax XHR may not be the favored choice due to explicitly adding promises which makes it unnecessarily complex, this is easily overcome by using the fetch API which comes with promises built-in and hence an easier syntax. Both the jQuery ajax and Axios are similar to the fetch API, the difference is the added features that they bring.

- Kheart nyo pagod na