

Week 6: APIs and REST APIs

L6.1: API Design

Distributed Software Architecture

- Distributed software architecture refers to a design approach where the components and functions of a software application are distributed across multiple computers connected over a network.

Web Architecture

- Web architecture is the overall structure of a website or web application, including the way it is designed, implemented and deployed.
- It involves the use of technologies and protocols such as HTML, CSS, JavaScript and HTTP to build and deliver web pages and applications to users.

1. Client-Server Architecture

- Client-server architecture is a software architecture where the client and the server are two separate entities that communicate with each other over a network.
- The client is responsible for requesting data from the server, and the server is responsible for providing that data to the client.
- **Example:**
 - When you browse a website, your web browser (client) sends a request to the web server asking for a specific web page.
 - The web server processes the request and sends back the requested web page, which your browser then displays to you.

a. Clear Separation of Concerns

- The client is responsible for the user interface and user interactions, while the server manages data processing, business logic, and storage.
- This separation allows for easier maintenance and scalability of the system.

b. Distributed Nature

- The client and server can run on different devices connected over a network, allowing users to access resources and services from remote servers, making it suitable for large-scale deployments.

2. Stateless Architecture

- Stateless is a property of a web application that means that the server does not maintain any state about the client.

- **Example:**

- When you search for different items on an e-commerce website, each search request is stateless.
- The server processes each search request independently without remembering your previous searches or interactions.

a. Stateless communication

- The web architecture is stateless, meaning each client request to the server is independent and does not carry information about previous interactions.
- The server treats each request as a new one without relying on any previous context.

b. Simplified Server Management

- Stateless design simplifies server management as the server doesn't need to maintain session data for each client continuously.

c. Scalability

- Stateless systems are easier to scale horizontally, as any server can handle any client request without relying on shared state information.

3. Layered Architecture

- Layered system is a software architecture where the application is divided into a number of layers, each of which is responsible for a specific task.
- This makes the application easier to develop, maintain, and scale.
- **Example:**
 - In a three-tier architecture, the front-end handles user interface interactions, the middle-tier processes business logic, and the back-end stores data.
 - Each layer communicates only with the adjacent layers, promoting modularity.

a. Modular Design

- The web architecture often adopts a layered system, dividing the application into multiple tiers (e.g., presentation, application logic, data storage), with each layer having specific responsibilities.

b. Flexibility and Reusability

- Layers are loosely occupied, allowing changes in one layer without affecting others, promoting flexibility and reusability of components.

c. Easier Maintenance

- Layered systems are easier to maintain and update since modifications can be confined to specific layers, making debugging and troubleshooting more straightforward.

4. Cacheability

- Cacheability is the ability to store data in a cache so that it can be accessed more quickly later.
- This can improve the performance of web applications by reducing the number of requests that need to be

made to the server.

- **Example:**

- After loading a web page, your browser caches the page's static resources (e.g., images, CSS files).
- The next time you visit the same page, these resources can be retrieved from the cache, reducing loading time.

a. Caching for Performance

- Cacheability allows web clients and intermediaries (e.g., proxies) to store responses from servers temporarily.
- Subsequent requests can be satisfied from the cache, reducing server load and improving response times.

b. Reduced Latency

- Cached responses lead to reduced latency, as clients can retrieve resources from the cache without making a new request to the server, especially for frequently accessed content.

c. Cache Control Headers

- Web servers use cache control headers (e.g., Cache-Control, Expires) to instruct clients and intermediaries on caching behavior, allowing developers to control cache duration and content freshness.

5. Uniform Interface

- Uniform interface is a set of constraints that define how resources should be represented and accessed in a RESTful web application.
- These constraints help to ensure that RESTful web applications are consistent and easy to use.
- **Example:**
 - A RESTful API provides a uniform interface for interacting with resources.
 - It uses standard HTTP methods like `GET`, `POST`, `PUT`, `DELETE` to perform actions on resources identified by unique URLs.
 - For instance, `GET /users` retrieves a list of users, while `POST /users` creates a new user.

a. Consistent Interaction

- The uniform interface provides a consistent and standardized way for clients to interact with resources, making the web easier to understand and use.

b. Resource based URLs

- Resources are identified by unique URLs, allowing clients to access and manipulate resources using standard HTTP methods like `GET`, `POST`, `PUT`, `DELETE`.

L6.2: *REST*

- REST stands for **REpresentational State Transfer**.

- It is an architectural style for designing web services.
- RESTful web services use HTTP methods to perform operations on resources.
- The most common HTTP methods are `GET` , `POST` , `PUT` and `DELETE` .
- Resources are identified by **URIs (Uniform Resource Identifiers)**.
- URIs are unique addresses that identify a specific resource.
- Resources are represented in a standard format.
- The most common formats are **JSON (JavaScript Object Notation)** and **XML (Extensible Markup Language)**.
- RESTful web services are stateless. This means that the server does not maintain any state about the client.
- RESTful web services are cacheable. This means that the client can cache the results of a request, so that it does not have to make the same request again.

Idempotent Operations

- An operation is idempotent if it can be applied multiple times without changing the result beyond the initial application.
- For example: adding 0 to a number, or multiplying a number by 1.
- Idempotent operations are often used in RESTful web services. This is because they can be used to ensure that the state of the server is not changed by repeated requests.

Example:

- `GET` : The `GET` method is used to retrieve a resource. It does not change the state of the resource.
- `PUT` : The `PUT` method is used to update a resource. If the resource does not exist, it will be created. If it does exist, it may give error.
- `DELETE` : The `DELETE` method is used to delete a resource. If the resource does not exist, nothing will happen. If it does exist, it will be deleted, multiple deletions lead to an error.

REST vs CRUD

Feature	REST	CRUD
<i>Definition</i>	REST stands for REpresentational State Transfer. It is an architectural style for designing web services.	CRUD stands for Create, Read, Update, and Delete. It is a set of four basic operations that can be performed on data.
<i>Underlying principles</i>	REST is based on the HTTP protocol. It uses HTTP methods to perform operations on resources.	CRUD is not based on any specific protocol. It is a set of abstract operations that can be implemented using any protocol.
<i>Resources</i>	In REST, resources are identified by URIs.	In CRUD, resources are identified by names.

Feature	REST	CRUD
Representations	In REST, resources are represented in a standard format, such as <code>JSON</code> or <code>XML</code> .	In CRUD, resources can be represented in any format.
Operations	REST defines four standard operations for manipulating resources: <code>GET</code> , <code>POST</code> , <code>PUT</code> , and <code>DELETE</code> .	CRUD defines four basic operations for manipulating data: <code>CREATE</code> , <code>READ</code> , <code>UPDATE</code> , and <code>DELETE</code> .

API data transfer format

- **INPUT:** text - `HTTP`
- **OUTPUT:** text - `HTML`, `JSON`, `XML`, `YAML` etc...

HTML

- HTML stands for **HyperText Markup Language**
- HTML is the most common format for delivering web pages. It is easy to read and write, and it is supported by all web browsers.
- However, HTML is not very efficient for transmitting data, and it can be difficult to parse.
- **Example:**

```
<html>
  <head>
    <title>API Response</title>
  </head>
  <body>
    <h1>List of Users</h1>
    <ul>
      <li>
        <strong>User 1</strong>
        <ul>
          <li>Name: John Doe</li>
          <li>Email: johndoe@example.com</li>
          <li>Phone: 123-456-7890</li>
        </ul>
      </li>
      <li>
        <strong>User 2</strong>
        <ul>
          <li>Name: Jane Doe</li>
          <li>Email: janedoe@example.com</li>
          <li>Phone: 987-654-3210</li>
        </ul>
      </li>
    </ul>
  </body>
</html>
```

JSON

- JSON stands for **JavaScript Object Notation**
- JSON is a lightweight format that is well-suited for transmitting data between applications.
- It is easy to read and write, and it is supported by most programming languages.
- JSON is also a very efficient format, which makes it ideal for transferring large amounts of data.
- **Example:**

```
[
  {
    "name": "John Doe",
    "email": "johndoe@example.com",
    "phone": "123-456-7890"
  },
  {
    "name": "Jane Doe",
    "email": "janedoe@example.com",
    "phone": "987-654-3210"
  }
]
```

XML

- XML stands for **Extensible Markup Language**
- XML is a more versatile format than JSON.
- It can be used to represent a wider variety of data, and it is supported by a wider range of applications.
- However, XML can be more difficult to read and write than JSON, and it is not as efficient.
- **Example:**

```
<users>
  <user>
    <name>John Doe</name>
    <email>johndoe@example.com</email>
    <phone>123-456-7890</phone>
  </user>
  <user>
    <name>Jane Doe</name>
    <email>janedoe@example.com</email>
    <phone>987-654-3210</phone>
  </user>
</users>
```

YAML

- YAML stands for **Yet Another Markup Language**
- YAML is a human-readable format that is similar to JSON.
- It is easy to read and write, and it is supported by most programming languages.
- YAML is also a very efficient format, which makes it ideal for transferring large amounts of data.
- However, YAML is not as widely supported as JSON or XML.
- **Example:**

```
users:
```

- name: John Doe
email: johndoe@example.com
phone: 123-456-7890
- name: Jane Doe
email: janed@example.com
phone: 987-654-3210

L6.3: *REST APIs - Example*

This example is based on the [MediaWiki REST API](#).

Request

curl

- Curl is a command-line tool that can be used to transfer data from or to a server.
- It is a powerful tool that can be used to perform a variety of tasks, such as downloading files, uploading files, and making HTTP requests.

```
# Search English Wikipedia for up to 3 pages containing information about earth
curl "https://en.wikipedia.org/w/rest.php/v1/search/page?q=earth&limit=3"
```

Python

```
# Search English Wikipedia for up to 3 pages containing information about earth
import requests

search_query = 'earth'
number_of_results = 3
url = 'https://en.wikipedia.org/w/rest.php/v1/search/page'
headers = {'User-Agent': 'MediaWiki REST API docs examples/0.1 (https://www.mediawiki.org/wiki/API_talk:)'

response = requests.get(url, headers=headers, params={'q': search_query, 'limit': number_of_results})
data = response.json()

print(data)
```

parameters:

- `q`: The search query.
- `limit`: The maximum number of results to return.

responses

- **200** : Success results found. Returns a **pages** object containing an array of search results.
- **200** : No results found. Returns an empty **pages** object.
- **400** : Query parameter not set. Add a **q** parameter to the request.
- **400** : Invalid query parameter. The **q** parameter must be a string.
- **500** : Search error / Internal error. Try again later.

Response


```

{
  "pages": [
    {
      "id": 9228,
      "key": "Earth",
      "title": "Earth",
      "excerpt": "<span class=\"searchmatch\">Earth</span> is the third planet from the Sun and the",
      "matched_title": null,
      "description": "Third planet from the Sun",
      "thumbnail": {
        "mimetype": "image/jpeg",
        "width": 60,
        "height": 60,
        "duration": null,
        "url": "//upload.wikimedia.org/wikipedia/commons/thumb/c/cb/The_Blue_Marble_%28remastered%29.jpg/60px-The_Blue_Marble_%28remastered%29.jpg"
      }
    },
    {
      "id": 2126501,
      "key": "Google_Earth",
      "title": "Google Earth",
      "excerpt": "Google <span class=\"searchmatch\">Earth</span> is a computer program that renders",
      "matched_title": null,
      "description": "3D globe-based map program owned by Google",
      "thumbnail": {
        "mimetype": "image/svg+xml",
        "width": 60,
        "height": 60,
        "duration": null,
        "url": "//upload.wikimedia.org/wikipedia/commons/thumb/e/e4/Google_Earth_icon.svg/60px-Google_Earth_icon.svg"
      }
    },
    {
      "id": 19331,
      "key": "Moon",
      "title": "Moon",
      "excerpt": "The Moon is <span class=\"searchmatch\">Earth</span>'s only natural satellite. It",
      "matched_title": null,
      "description": "Natural satellite orbiting the Earth",
      "thumbnail": {
        "mimetype": "image/jpeg",
        "width": 60,
        "height": 57,
        "duration": null,
        "url": "//upload.wikimedia.org/wikipedia/commons/thumb/e/e1/FullMoon2010.jpg/60px-FullMoon2010.jpg"
      }
    }
  ]
}

```

Schema

- `id` : Page identifier.

- `key` : Page title in URL-friendly format.
- `title` : Page title in reading-friendly format.
- `excerpt` : A few lines giving a sample of page content with search terms highlighted with `` tags
- `matched_title` : The title of the page redirected from, if the search term originally matched a redirect page or null if search term did not match a redirect page.
- `description` : Short summary of the page topic based on the corresponding entry on Wikidata or null if no entry exists.
- `thumbnail` : Information about the thumbnail image for the page or null if no thumbnail exists.
 - `mimetype` : MIME type of the thumbnail image.
 - `size` : Size of the thumbnail image in bytes.
 - `width` : Width of the thumbnail image in pixels.
 - `height` : Height of the thumbnail image in pixels.
 - `duration` : Duration of the thumbnail image in seconds.
 - `url` : URL of the thumbnail image.

L6.4: *REST APIs - Example 2*

L6.5: *OpenAPI*

- OpenAPI (or Swagger) is a specification for describing RESTful APIs.
- It is used to define the endpoints, parameters, and data schemas of an API.
- OpenAPI is a popular specification, and there are many tools that can be used to generate code, documentation, and test cases from OpenAPI definitions.

1. API Description language

- OpenAPI uses a standardized API description language, typically written in YAML or JSON format, to define the structure and details of an API.
- This description acts as a contract between the API provider and consumers, providing a clear understanding of how to interact with the API.

2. Endpoint Documentation

- OpenAPI allows developers to document each API endpoint with details such as the HTTP method (`GET` , `POST` , `PUT` , `DELETE`), request parameters, request body, response schema, and more.
- This documentation makes it easier for developers to consume the API by understanding the available resources and how to use them.

3. Tooling and Interoperability

- OpenAPI has extensive tooling support, including code generators, API explorers, and automatic documentation generators.
- These tools enable developers to easily create client SDKs, validate requests and responses, and generate interactive API documentation.
- The standardized format also promotes interoperability, allowing various programming languages and frameworks to interact seamlessly with APIs.

4. OpenAPI Specification

- The OpenAPI Specification is a community-driven open specification for describing RESTful APIs.

4.1 API version and Paths

- The specification defines the version of the API and the available paths (endpoints) that clients can access.

4.2 HTTP Methods and Parameters

- It outlines the HTTP methods supported by each endpoint (`GET` , `POST` , `PUT` , `DELETE` , etc.) and the parameters required for each request.

4.3 Request and Response Schemas

- The specification specifies the expected format for request payloads and the structure of the response data.

4.4 Authentication and Security

- It can include details about authentication mechanisms and security requirements for accessing the API.

4.5 Error Handling

- The specification may define standard error codes and messages that the API uses to communicate errors to clients.

4.6 API Documentation

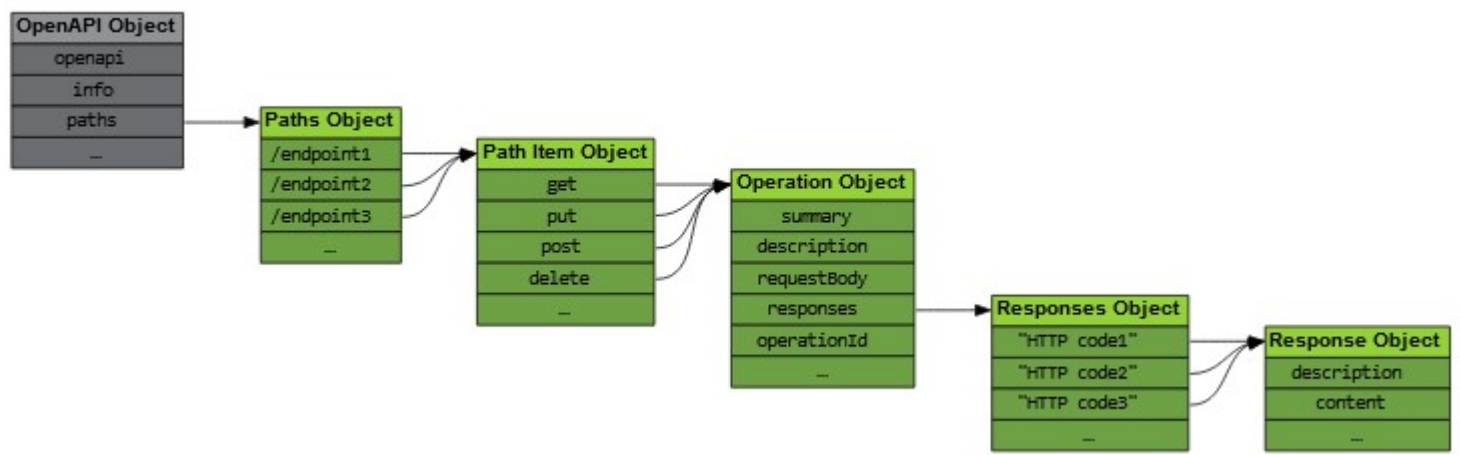
- The OpenAPI Specification acts as the primary source of API documentation, making it easier for developers to understand and consume the API.

4.7 Data Models

- The specification can describe data models or schemas used by the API, enabling clients to understand the structure of data being exchanged.

L6.6: Important Concepts of an API

Endpoints list



Best Practices

- Design first vs Code first
 - Always prefer design-first!
- Single source of truth
 - The structure of the code should be derived from the OAS
 - Spec should be derived from code
 - Minimize chances of code and documentation diverging
- Source code version control
- OPENAPI is Open public documentation better to identify problems
- Use a linter to validate the OAS, code.
- Check this website [OpenAPI.tools](https://openapi.tools)

Watch full video lecture [here](#)

Screencast

6.1 Documenting Open API

6.2 Introduction to Flask-RESTful - I

6.3 Introduction to Flask-RESTful - II

What is an API?

• by Rapid API

What is an API ?

@Rapid_API

APIs (application programming interface) are the intermediary between two programs, and allow data to be transferred.

APIs are very versatile and can be used on the web, databases, and operating systems.



Request

@Rapid_API


An API Call is initiated. This is the process of the client app submitting a request to a server.

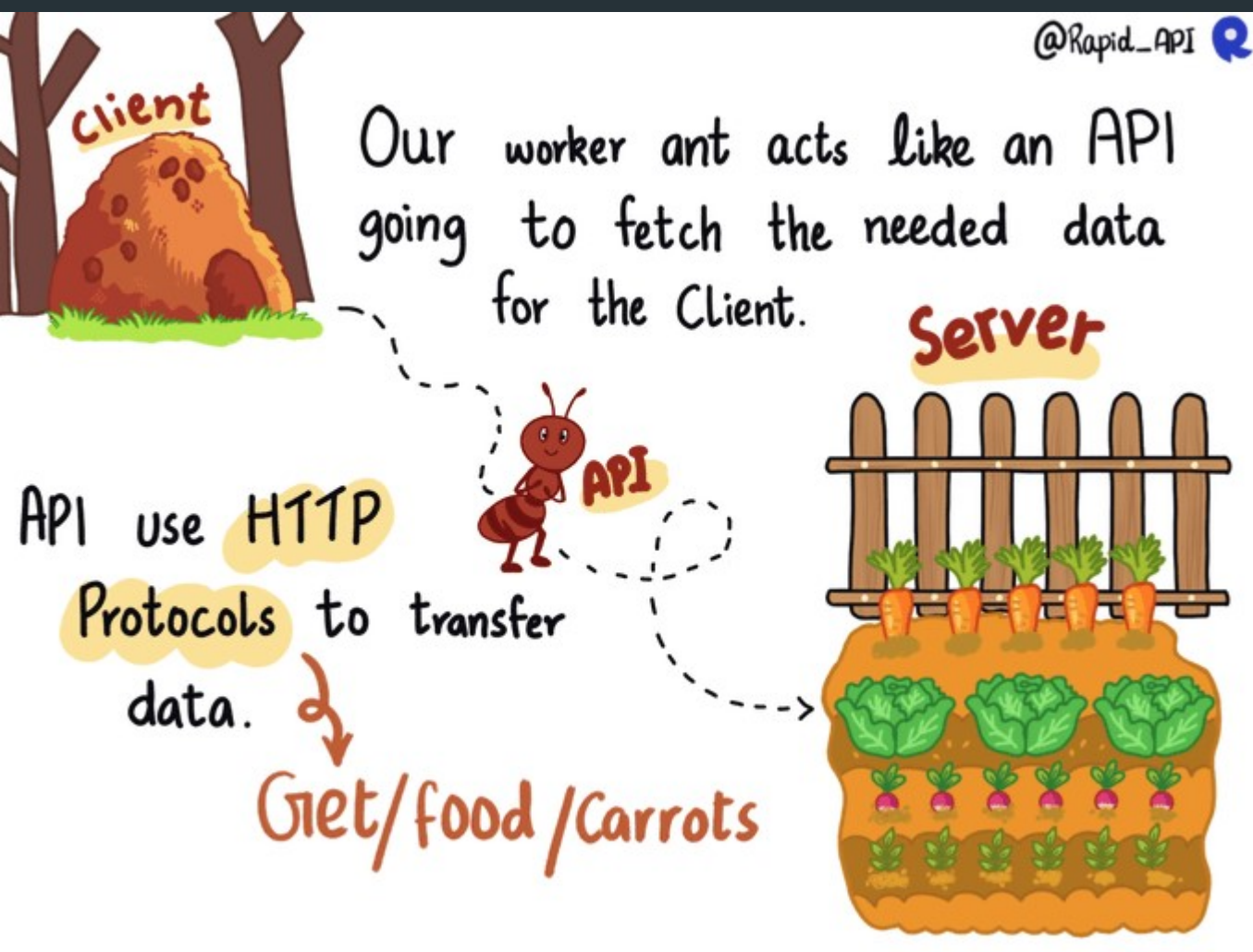
Hey!
we need some
more food (data)
for the
Colony (app).

APIs can be used to share data, embed content.



and more.

@Rapid_API 



Our worker ant acts like an API going to fetch the needed data for the Client.

Server


API

API use **HTTP**

Protocols to transfer data.

Get/food/Carrots

Our ant (API) collecting food (data)

@Rapid_API 

To find the right data, APIs have endpoints.

Endpoints are essentially the URLs that navigate to the correct resource.


Our endpoint in

FACT

Each time you open up Twitter, google maps, a weather app, and so many more, you're using APIs. APIs are everywhere!

this case is a **Carrot**.

Response

@Rapid_API 

As long as the server (field) can return the requested data (food) to the client Successfully, then **mission accomplished!**



If the server can not return what the client asked for, the API will return the appropriate **error message.**

Cheatsheets

[XML Cheatsheet](#) 

[JSON Cheatsheet](#) 

[YAML Cheatsheet](#) 