1 . REST API Design Best Practices

1. Use JSON as the Format for Sending and Receiving Data

In the past, accepting and responding to API requests were done mostly in XML and even HTML. But these days, JSON (JavaScript Object Notation) has largely become the de-facto format for sending and receiving API data.

This is because, with XML for example, it's often a bit of a hassle to decode and encode data – so XML isn't widely supported by frameworks anymore.

JavaScript, for example, has an inbuilt method to parse JSON data through the fetch API because JSON was primarily made for it. But if you are using any other programming language such as Python or PHP, they now all have methods to parse and manipulate JSON data as well.

To ensure the client interprets JSON data correctly, you should set the `Content-Type` type in the response header to `application/json` while making the request.

For server-side frameworks, on the other hand, many of them set the `Content-Type` automatically. Express, for example, now has the `express.json()` middleware for this purpose. The `body-parser` NPM package still works for the same purpose, too.

2. Use Nouns Instead of Verbs in Endpoints

When you're designing a REST API, you should not use verbs in the endpoint paths. The endpoints should use nouns, signifying what each of them does.

This is because HTTP methods such as `GET`, `POST`, `PUT`, `PATCH`, and `DELETE` are already in verb form for performing basic CRUD (Create, Read, Update, Delete) operations.

`GET`, `POST`, `PUT`, `PATCH`, and `DELETE` are the commonest HTTP verbs. There are also others such as `COPY`, `PURGE`, `LINK`, `UNLINK`, and so on.

So, for example, an endpoint should not look like this:

`https://mysite.com/getPosts` or `https://mysite.com/createPost`

Instead, it should be something like this: `https://mysite.com/posts`

In short, you should let the HTTP verbs handle what the endpoints do. So `GET` would retrieve data, `POST` will create data, `PUT` will update data, and `DELETE` will get rid of the data.

4. Use Status Codes in Error Handling

You should always use regular HTTP status codes in responses to requests made to your API. This will help your users to know what is going on – whether the request is successful, or if it fails, or something else.

Below is a table showing different HTTP Status Code ranges and their meanings:

100 – 199	Informational Responses. For example, 102 indicates the resource is being

	processed
300 – 399	Redirects For example, 301 means Moved permanently
400 – 499	Client-side errors 400 means bad request and 404 means resource not found
500 – 599	Server-side errors For example, 500 means an internal server error

5. Use Nesting on Endpoints to Show Relationships

Oftentimes, different endpoints can be interlinked, so you should nest them so it's easier to understand them.

For example, in the case of a multi-user blogging platform, different posts could be written by different authors, so an endpoint such as `https://mysite.com/posts/author` would make a valid nesting in this case.

In the same vein, the posts might have their individual comments, so to retrieve the comments, an endpoint like `https://mysite.com/posts/postId/comments` would make sense.

6. Use Filtering, Sorting, and Pagination to Retrieve the Data Requested

Sometimes, an API's database can get incredibly large. If this happens, retrieving data from such a database could be very slow.

Filtering, sorting, and pagination are all actions that can be performed on the collection of a REST API. This lets it only retrieve, sort, and arrange the necessary data into pages so the server doesn’t get too occupied with requests.

An example of a filtered endpoint is the one below:
`https://mysite.com/posts?tags=javascript`
This endpoint will fetch any post that has a tag of JavaScript.

7. Use SSL for Security

SSL stands for secure socket layer. It is crucial for security in REST API design. This will secure your API and make it less vulnerable to malicious attacks.

Other security measures you should take into consideration include: making the communication between server and client private and ensuring that anyone consuming the API doesn’t get more than what they request.

SSL certificates are not hard to load to a server and are available for free mostly during the first year. They are not expensive to buy in cases where they are not available for free.

The clear difference between the URL of a REST API that runs over SSL and the one which does not is the “s” in HTTP:
`https://mysite.com/posts` runs on SSL.
`http://mysite.com/posts` does not run on SSL.

8. Be Clear with Versioning

REST APIs should have different versions, so you don’t force clients (users) to migrate to new versions. This might even break the application if you're not careful.

One of the commonest versioning systems in web development is semantic versioning.

An example of semantic versioning is 1.0.0, 2.1.2, and 3.3.4. The first number represents the major version, the second number represents the minor version, and the third represents the patch version.

Many RESTful APIs from tech giants and individuals usually comes like this:
`https://mysite.com/v1/` for version 1
`https://mysite.com/v2` for version 2

9. Provide Accurate API Documentation

When you make a REST API, you need to help clients (consumers) learn and figure out how to use it correctly. The best way to do this is by providing good documentation for the API.

The documentation should contain:

- relevant endpoints of the API
- example requests of the endpoints
- implementation in several programming languages
- messages listed for different errors with their status codes

One of the most common tools you can use for API documentation is Swagger. And you can also use Postman, one of the most common API testing tools in software development, to document your APIs.

2 . HTTP Methods best practices

Basically, for CRUD operations (Create, Retrieve, Update and Delete) you can use the HTTP methods as follows:

- POST: create resource or search operation

- GET: read resource operation

- PUT: update resource operation

- DELETE: remove resource operation

- PATCH: partial update resource operation

API end point with POST method:

Used mainly for Create resource operations, or Read operations in some certain cases:

- Read resource if URL / query string exceeds maximum allowed characters. Maximum length of URL and query string is 2,048 characters.
- Read resource if query parameters contain sensitive information

Return status code:

- ***201 Created*** for successful create operation
- ***200 OK***for successful read operation if the response contains data
- ***204 No Content*** for successful read operation if the response contains NO data

API end point with GET method:

Used primary for Read resource operations.

Return status code:

- ***200 OK*** if the response contains data
- ***204 No Content*** if the response contains no data

API end point with PUT method:

Used for Update resource operations.

Return status code:

- ***200 OK*** if the response contains data
- ***204 No Content*** if the response contains no data

API end point with DELETE method:

Used for Delete resource operations.

Return status code: ***204 No Content*** for successful delete operation.

API end point with PATCH method:

Used for Partial update resource operations.

Return status code: ***200 OK***for successful partial update operation.

Returning the right HTTP status codes in REST APIs is also important, to ensure uniform interface architectural constraint. Besides the status codes above, below is the guideline for common HTTP status codes:

- ***200 OK***: for successful requests
- ***201 Created***: for successful creation requests

- ***202 Accepted***: the server accepts the request, but the response cannot be sent immediately (e.g. in batch processing)
- ***204 No Content***: for successful operations that contain no data
- ***304 Not Modified***: used for caching, indicating the resource is not modified
- ***400 Bad Request***: for failed operation when input parameters are incorrect or missing, or the request itself is incomplete
- ***401 Unauthorized***: for failed operation due to unauthenticated requests
- ***403 Forbidden***: for failed operation when the client is not authorized to perform
- ***404 Not Found***: for failed operation when the resource doesn't exist
- ***405 Method Not Allowed***: for failed operation when the HTTP method is not allowed for the requested resource
- ***406 Not Acceptable***: for failed operation when the Accept header doesn't match. Also can be used to refuse request
- ***409 Conflict***: for failed operation when an attempt is made for a duplicate create operation
- ***429 Too Many Requests***: for failed operation when a user sends too many requests in a given amount of time (rate limiting)
- ***500 Internal Server Error***: for failed operation due to server error (generic)
- ***502 Bad Gateway***: for failed operation when the upstream server calls fail (e.g. call to a third-party service fails)
- ***503 Service Unavailable***: for a failed operation when something unexpected happened at the server (e.g. overload of service fails)

3 . JSON Methods best practices

JSON Best Practices Summary

- : Publish data using developer friendly JSON
- : Use a top-level object
- : Use native values
- : Assume arrays are unordered
- : Use well-known identifiers when describing data
- : Provide one or more types for JSON objects
- : Identify objects with a unique identifier
- : Things not strings
- : Nest referenced inline objects
- : When describing an inverse relationship, use a referenced property
- : External references *SHOULD* use typed term
- : Ordering of array elements
- : Provide a representation of the entity related by URL
- : Cache JSON-LD Contexts

: Publish data using developer friendly JSON

JSON [*json*] is the most popular format for publishing data through APIs; developers like it, it is easy to parse, and it is supported natively in most programming languages.

For example, the following is reasonably idiomatic JSON which can also be interpreted as JSON-LD, given the appropriate context.

Example: Example JSON representing a Person

```
{
  "name": "Barack Obama",
  "givenName": "Barack",
  "familyName": "Obama",
  "jobTitle": "44th President of the United States"
}
```

: Use a top-level object

JSON documents may be in the form of a object, or an array of objects. For most purposes, developers need a single entry point, so the JSON *SHOULD* be in the form of a single top-level object.

: Use native values

When possible, property values *SHOULD* use native JSON datatypes such as numbers (*integer*, *decimal* and *floating point*) and booleans (`true` and `false`).

Note

JSON has a single numeric type, so using native representation of numbers can lose precision.

: Assume arrays are unordered

JSON specifies that the values in an array are ordered, however in many cases arrays are also used for values which are unordered. Unless specified within the JSON-LD Context, multiple array values *SHOULD* be presumed to be unordered. (See *Lists and Sets* in [*JSON-LD*]).

: Use well-known identifiers when describing data

By sticking to basic JSON data expression, and providing a JSON-LD Context, all keys used within a JSON document can have unambiguous meaning, as they bind to URLs which describe their meaning.

By adding an `@context` entry, the previous example can now be interpreted as JSON-LD.

Example: Example JSON-LD identifying a person

```
{
  "@context": "http://schema.org",
  "name": "Barack Obama",
  "givenName": "Barack",
  "familyName": "Obama",
  "jobTitle": "44th President of the United States"
}
```

Note

When expanding such a data representation, a JSON-LD processor replaces these terms with the URIs they expand to (as well as making property values unambiguous):

Example: Example JSON-LD identifying a person (expanded)

```
[
  {
    "http://schema.org/familyName": [{"@value": "Obama"}],
    "http://schema.org/givenName": [{"@value": "Barack"}],
    "http://schema.org/jobTitle": [{"@value": "44th President of the United
States"}],
    "http://schema.org/name": [{"@value": "Barack Obama"}]
  }
]
```

Expanded form is not useful as is, but is necessary for performing further algorithmic transformations of JSON-LD data and is useful when validating that JSON-LD entity descriptions say what the publisher means.

: Provide one or more types for JSON objects

Principles of Linked Data dictate that messages *SHOULD* be self describing, which includes adding a `type` to such messages.

Many APIs use JSON messages where the type of information being conveyed is inferred from the retrieval endpoint. For example, when retrieving information about a Github Commit, you might see the following response:

Example: Github Commit message

```
{
  "sha": "7638417db6d59f3c431d3e1f261cc637155684cd",
  "url":
"https://api.github.com/repos/octocat/Hello-World/git/commits/7638417db6d59f3c43
1d3e1f261cc637155684cd",
  "author": {
    "date": "2014-11-07T22:01:45Z",
    "name": "Scott Chacon",
    "email": "schacon@gmail.com"
  },
  "committer": {
    "date": "2014-11-07T22:01:45Z",
    "name": "Scott Chacon",
    "email": "schacon@gmail.com"
  },
  "message": "added readme, because im a good github citizen\n",
  "tree": {
    "url":
"https://api.github.com/repos/octocat/Hello-World/git/trees/691272480426f78a0138
979dd3ce63b77f706feb",
    "sha": "691272480426f78a0138979dd3ce63b77f706feb"
  },
  "parents": [
    {
      "url":
"https://api.github.com/repos/octocat/Hello-World/git/commits/1acc419d4d6a9ce985
db7be48c6349a0475975b5",
      "sha": "1acc419d4d6a9ce985db7be48c6349a0475975b5"
    }
```

```
  ]
}
```

The only way to know this is a commit s to infer it based on the published API documentation, and the fact that it was returned from an endpoint defined for retrieving information about commits.

Example: Example message of type Person

```
{
  "@context": "http://schema.org",
  "id": "http://www.wikidata.org/entity/Q76",
  "type": "Person",
  "name": "Barack Obama",
  "givenName": "Barack",
  "familyName": "Obama",
  "jobTitle": "44th President of the United States"
}
```

: Identify objects with a unique identifier

Entities described in JSON objects often describe web resources having a URL; entity descriptions *SHOULD* use an identifier uniquely identifying that entity. In this case, using the resource location as the identity of the object is consistent with this practice.

Adding an `id` entry (an alias for `@id`) allows the same person to be referred to from different locations.

Example: Example JSON-LD identifying a person

```
{
  "@context": "http://schema.org",
  "id": "http://www.wikidata.org/entity/Q76",
  "type": "Person",
  "name": "Barack Obama",
  "givenName": "Barack",
  "familyName": "Obama",
  "jobTitle": "44th President of the United States"
}
```

: Things not strings

When describing attributes, entity references *SHOULD* be used instead of string literals.

In some cases, when describing an attribute of an entity, it is tempting to using string values which have no independent meaning. Such values are often used for well known things. A JSON-LD context can define a term for such values, which allow them to appear as strings within the message, but be associated with specific identifiers. In this case, the property must be defined with type `@vocab` so that values will be interpreted relative to a vocabulary rather than the file location.

Example: Example enumerated value

```
{
  "@context": ["http://schema.org", {
    "gender": {"@id": "schema:gender", "@type": "@vocab"}
  }],
```

```
  "id": "http://www.wikidata.org/entity/Q76",
  "type": "Person",
  "name": "Barack Obama",
  "givenName": "Barack",
  "familyName": "Obama",
  "jobTitle": "44th President of the United States",
  "gender": "Male"
}
```

:Nest referenced inline objects

When multiple related entity descriptions are provided inline, related entities *SHOULD* be nested.

For example, when relating one entity to another, where the related entity is described in the same message:

Example: Nested relationships

```
{
  "@context": "http://schema.org",
  "id": "http://www.wikidata.org/entity/Q76",
  "type": "Person",
  "name": "Barack Obama",
  "givenName": "Barack",
  "familyName": "Obama",
  "jobTitle": "44th President of the United States",
  "spouse": {
    "id": "http://www.wikidata.org/entity/Q13133",
    "type": "Person",
    "name": "Michelle Obama",
    "spouse": "http://www.wikidata.org/entity/Q76"
  }
}
```

Note

In this example, the `spouse` relationship is bi-directional, we have arbitrarily rooted the message with Barack Obama, and created a symmetric relationship from Michelle back to Barack by reference, rather than by nesting.

: When describing an inverse relationship, use a referenced property

FIXME

: External references *SHOULD* use typed term

When using a property intended to reference another entity, properties *SHOULD* be defined to type string values as being references.

For example, the `schema:image` property a `Thing` to an `Image`:

Example: Example typed relationship

```
{
  "@context": "http://schema.org",
  "id": "http://www.wikidata.org/entity/Q76",
```

```
  "type": "Person",
  "name": "Barack Obama",
  "givenName": "Barack",
  "familyName": "Obama",
  "jobTitle": "44th President of the United States",
  "image": "https://commons.wikimedia.org/wiki/File:President_Barack_Obama.jpg"
}
```

This will be interpreted as a reference, rather than a string literal, because (at the time of publication), the schema.org JSON-LD Context defines `image` to be of type `@id`:

Example: schema.org context snippet for image

```
{
  "@context": {
    ...
    "image": { "@id": "schema:image", "@type": "@id"},
    ...
  }
}
```

If not defined as such in a remote context, terms may be (re-) defined in a local context:

Example: Example typed relationship

```
{
  "@context": ["http://schema.org", {
    "image": { "@id": "schema:image", "@type": "@id"}
  }],
  "id": "http://www.wikidata.org/entity/Q76",
  "type": "Person",
  "name": "Barack Obama",
  "givenName": "Barack",
  "familyName": "Obama",
  "jobTitle": "44th President of the United States",
  "image": "https://commons.wikimedia.org/wiki/File:President_Barack_Obama.jpg"
}
```

: Ordering of array elements

Unless specifically described ordered as an `@list`, do not depend on the order of elements in an array.

By default, arrays in JSON-LD do not convey any ordering of contained elements . However, for the processing of contexts, the ordering of elements in arrays *does* matter. When writing array-based contexts, this fact should be kept in mind.

Ordered contexts in arrays allow inheritance and overriding of context entries. When processing the following example, the first `name` entry will be overridden by the second `name` entry.

Example: Overriding name term

```
{
  "@context": [
    {
      "id": "@id",
      "name": "http://schema.org/name"
    },
    {
      "name": "http://xmlns.com/foaf/0.1/name"
    }
```

```
  ],
  "@id": "http://www.wikidata.org/entity/Q76",
  "name": "Barack Obama"
}
```

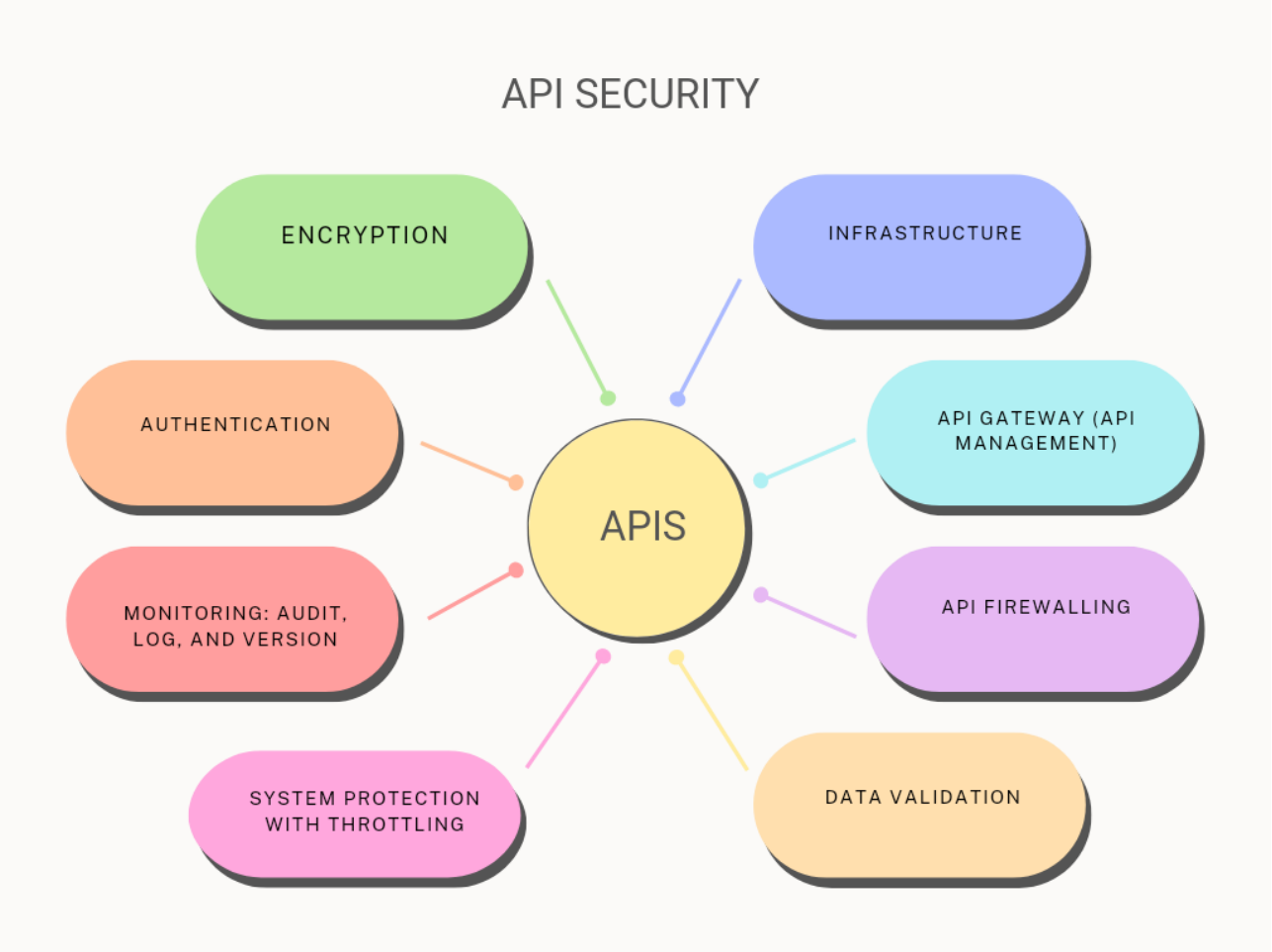
Order is important when processing protected terms. While the first example will cause a term redefinition error, the second example will not throw this error.

Example: Failing term redefinition

```
{
  "@context": [
    {
      "@version": 1.1,
      "name": {
        "@id": "http://schema.org/name",
        "@protected": true
      }
    },
    {
      "name": "http://xmlns.com/foaf/0.1/name"
    }
  ],
  "@id": "http://www.wikidata.org/entity/Q76",
  "name": "Barack Obama"
}
```

Example: Non-failing term redefinition

```
{
  "@context": [
    {
      "name": "http://xmlns.com/foaf/0.1/name"
    },
    {
      "@version": 1.1,
      "Person": "http://schema.org/Person",
      "knows": "http://schema.org/knows",
      "name": {
        "@id": "http://schema.org/name",
        "@protected": true
      }
    }
  ],
  "@id": "http://www.wikidata.org/entity/Q76",
  "name": "Barack Obama"
}
```



4 . Rest Api Security best practices

1. Encryption

Be cryptic. Nothing should be in the clear for internal or external communications. Encryption will convert your information into code. This will make it much more difficult for sensitive data to end up in the wrong hands.

You and your partners should cipher all exchanges with TLS (the successor to SSL), whether it is one-way encryption (standard one-way TLS) or, even better, mutual encryption (two-way TLS).

Use the latest TLS versions to block the usage of the weakest cipher suites.

2. Authentication

Don’t talk to strangers. In simple terms, authenticity is real. It means something (or someone) is who they say they are. In the digital world, authentication is the process of verifying a user’s identity. It essentially pulls off the mask of anyone who wants to see your information.

So, you should always know who is calling your APIs. There are several methods to authenticate:

- HTTP Basic authentication where a user needs to provide user ID and password
- API key where a user needs to a unique identifier configured for each API and known to API Gateway
- A token that is generated by an Identity Provider (IdP) server. OAuth 2 is the most popular protocol that supports this method.

At the very least you should use an API key (asymmetric key) or basic access authentication (user/password) to increase the difficulty of hacking your system. But you should consider using OAuth 2 as your protocol of choice for a robust security of your APIs.

3. OAuth & OpenID Connect

Delegate all responsibilities. A good manager delegates responsibility, and so does a great API. You should be delegating authorization and/or authentication of your APIs to third party Identity Providers (IdP).

What is OAuth 2? It is a magical mechanism preventing you from having to remember ten thousand passwords. Instead of creating an account on every website, you can connect through another provider’s credentials, for example, Facebook or Google.

It works the same way for APIs: the API provider relies on a third-party server to manage authorizations. The consumer doesn’t input their credentials but instead gives a token provided by the third-party server. It protects the consumer as they don’t disclose their credentials, and the API provider doesn’t need to care about protecting authorization data, as it only receives tokens.

OAuth is a commonly used delegation protocol to convey authorizations. To secure your APIs even further and add authentication, you can add an identity layer on top of it: this is the Open Id Connect standard, extending OAuth 2.0 with ID tokens.

4. Call security experts

Don’t be afraid to ask for (or use) some help. Call in some security experts. Use experienced Antivirus systems or ICAP (Internet Content Adaptation Protocol) servers to help you with scanning payload of your APIs. It will help you to prevent any malicious code or data affecting your systems.

There are several security APIs you can use to protect your data. They can do things like:

- Integrate two-factor authentication
- Create passwordless login, or time-based one-time passwords
- Send out push alerts if there’s a breach
- Protect against viruses and malware
- Prevent fraud
- Let you know if a password is a known password used by hackers

- Add threat intelligence
- Provide security monitoring

The best part is that some of these antivirus systems are free to use. Others offer monthly plans. Premium plans will provide more protection, but you can decide for yourself the type of security you need.

5. Monitoring: audit, log, and version

Be a stalker. Continually monitoring your API and what it's up to can pay off. Be vigilant like that overprotective parent who wants to know everything about the people around their son or daughter.

How do you do this? You need to be ready to troubleshoot in case of error. You'll want to audit and log relevant information on the server — and keep that history as long as it is reasonable in terms of capacity for your production servers.

Turn your logs into resources for debugging in case of any incidents. Keeping a thorough record will help you keep track and make anything that's suspicious more noticeable.

Also, monitoring dashboards are highly recommended tools to track your API consumption.

Do not forget to add the version on all APIs, preferably in the path of the API, to offer several APIs of different versions working and to retire and depreciate one version over the other.

6. Share as little as possible

Be paranoid. It's OK to be overly cautious. Remember, it's vital to protect your data.

Display as little information as possible in your answers, especially in error messages. Lock down email subjects and content to predefined messages that can't be customized. Because IP addresses can give locations, keep them for yourself.

Use IP Whitelist and IP Blacklist, if possible, to restrict access to your resources. Limit the number of administrators, separate access into different roles, and hide sensitive information in all your interfaces.

7. System protection with throttling and quotas

Throttle yourself. You should restrict access to your system to a limited number of messages per second to protect your backend system bandwidth according to your servers' capacity. Less is more.

You should also restrict access by API and by the user (or application) to ensure that no one will abuse the system or anyone API in particular.

Throttling limits and quotas – when well set – are crucial to prevent attacks coming from different sources flooding your system with multiple requests (DDOS – Distributed Denial of Service Attack). A DDOS can lock legitimate users out of their own network resources.

8. Data validation

Be picky and refuse surprise gifts, especially if they are significantly large. You should check everything your server accepts. Be careful to reject any added content or data that is too big, and

always check the content that consumers are sending you. Use JSON or XML schema validation and check that your parameters are what they should be (string, integer…) to prevent any SQL injection or XML bomb.

9. Infrastructure

Network and be up to date. A good API should lean on a good security network, infrastructure, and up-to-date software (for servers, load balancers) to be solid and always benefit from the latest security fixes.

10. OWASP Top 10

Avoid wasps. The OWASP (Open Web Application Security Project) Top 10 is a list of the ten worst vulnerabilities, ranked according to their exploitability and impact. In addition to the above points, to review your system, ensure you have secured all OWASP vulnerabilities.

11. API firewalling

Build a wall. For some people, building a wall can solve all the immigration problems. This is the case, for APIs at least! Your API security should be organized into two layers:

- The first layer is in DMZ, with an API firewall to execute basic security mechanisms like checking the message size, SQL injections, and any security based on the HTTP layer, blocking intruders early. Then forward the message to the second layer.
- The second layer is in LAN with advanced security mechanisms on data content.

The more challenging you make it for cyber attackers to get at your information, the better.

12. API Gateway (API Management)

Gateway to heaven. All the above mechanisms are long to implement and maintain. Instead of reinventing the wheel, you should opt for a mature and high-performing API Management solution with all these options to save your money, time, and resources and increase your time to market. An API Gateway will help you secure, control, and monitor your traffic.

In addition to helping you secure your APIs easily, an API Management solution will help you make sense of your API data to make technical and business decisions: the key to success!

Now you know more about the basic mechanisms to protect your APIs! Have fun securing your APIs, hopefully with a great API Management solution.

5 . Back-End Developer Interview Questions

1 . What REST stands for?

Answer

REST stands for *REpresentational State Transfer*. REST is web standards based architecture and uses HTTP Protocol for data communication. It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods. REST was first introduced by Roy Fielding in 2000.

In REST architecture, a REST Server simply provides access to resources and REST client accesses and presents the resources. Here each resource is identified by URIs/ global IDs. REST uses various representations to represent a resource like text, JSON and XML. Now a days JSON is the most popular format being used in web services.

2 . What are NoSQL databases? What are the different types of NoSQL databases?

Answer

A NoSQL database provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases (like SQL, Oracle, etc.).

Types of NoSQL databases:

- Document Oriented
- Key Value
- Graph
- Column Oriented

3 . What do you understand by NoSQL databases? Explain.

Answer

At the present time, the internet is loaded with big data, big users, big complexity etc. and also becoming more complex day by day. NoSQL is answer of all these problems; It is not a traditional database management system, not even a relational database management system (RDBMS). NoSQL stands for “Not Only SQL”. NoSQL is a type of database that can handle and sort all type of unstructured, messy and complicated data. It is just a new way to think about the database.

4 . What is SQL injection?

Answer

Injection attacks stem from a lack of strict separation between program instructions (i.e., code) and user-provided (or external) input. This allows an attacker to inject malicious code into a data snippet.

SQL injection is one of the most common types of injection attack. To carry it out, an attacker provides malicious SQL statements through the application.

How to prevent:

- **Prepared statements with parameterized queries**
- **Stored procedures**
- **Input validation** - blacklist validation and whitelist validation
- **Principle of least privilege** - Application accounts shouldn't assign DBA or admin type access onto the database server. This ensures that if an application is compromised, an attacker won't have the rights to the database through the compromised application.

5 . What is meant by *Continuous Integration*?

Answer

Continuous Integration (CI) is a development practice that requires developers to integrate code into a shared repository several times a day. Each check-in is then verified by an automated build, allowing teams to detect problems early.

6 . Name some performance testing steps

Answer

Some of the performance testing steps are:

- Identify the testing environment
- Identify performance metrics
- Plan and design performance tests
- Configure the test environment
- Implement your test design
- Execute tests
- Analyze, report, retest

7 . Name the difference between *Acceptance Test* and *Functional Test*

Answer

- **Functional testing**: This is a *verification* activity; *did we build a correctly working product?* Does the software meet the business requirements? A functional test verifies that the product actually works as you (the developer) think it does.
- **Acceptance testing**: This is a *validation* activity; *did we build the right thing?* Is this what the customer really needs? Acceptance tests verify the product actually solves the problem it was made to solve. This can best be done by the user (customer), for instance performing his/her tasks that the software assists with.

8 . What are the advantages of Web Services?

Answer

Some of the advantages of web services are:

- **Interoperability**: Web services are accessible over network and runs on HTTP/SOAP protocol and uses XML/JSON to transport data, hence it can be developed in any programming language. Web service can be written in java programming and client can be PHP and vice versa.
- **Reusability**: One web service can be used by many client applications at the same time.
- **Loose Coupling**: Web services client code is totally independent with server code, so we have achieved loose coupling in our application.
- **Easy to deploy and integrate**, just like web applications.
- **Multiple service versions** can be running at same time.

9 . What are the benefits of using Go programming?

Answer

Following are the benefits of using Go programming:

- Support for environment adopting patterns similar to dynamic languages. For example type inference (`x := 0` is valid declaration of a variable `x` of type `int`).
- Compilation time is fast.
- In built concurrency support: light-weight processes (via goroutines), channels, select statement.
- Conciseness, Simplicity, and Safety.
- Support for Interfaces and Type embedding.
- The go compiler supports static linking. All the go code can be statically linked into one big fat binary and it can be deployed in cloud servers easily without worrying about dependencies.

10 . What does *Containerization* mean?

Answer

Containerisation is a type of *virtualization* strategy that emerged as an alternative to traditional hypervisor-based virtualization.

In containerization, the operating system is shared by the different containers rather than cloned for each virtual machine. For example Docker provides a container virtualization platform that serves as a good alternative to hypervisor-based arrangements.

11 . Why Would You Opt For Microservices Architecture?

Answer

There are plenty of pros that are offered by Microservices architecture. Here are a few of them:

- Microservices can adapt easily to other frameworks or technologies.
- Failure of a single process does not affect the entire system.
- Provides support to big enterprises as well as small teams.
- Can be deployed independently and in relatively less time.

12 . Name some Performance Testing best practices

Answer

- Test as early as possible in development.
- Conduct multiple performance tests to ensure consistent findings and determine metrics averages.
- Test the individual software units separately as well as together
- Baseline measurements provide a starting point for determining success or failure
- Performance tests are best conducted in test environments that are as close to the production systems as possible
- Isolate the performance test environment from the environment used for quality assurance testing
- Keep the test environment as consistent as possible
- Calculating averages will deliver actionable metrics. There is value in tracking outliers also. Those extreme measurements could reveal possible failures.

13 . What Is ACID Property Of A System?

Answer

ACID is a acronym which is commonly used to define the properties of a relational database system, it stand for following terms

- **Atomicity** - This property guarantees that if one part of the transaction fails, the entire transaction will fail, and the database state will be left unchanged.
- **Consistency** - This property ensures that any transaction will bring the database from one valid state to another.
- **Isolation** - This property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially.
- **Durable** - means that once a transaction has been committed, it will remain so, even in the event of power loss.

14 . What are disadvantages of REST web services?

Answer

Some of the disadvantages of REST are:

- Since there is no contract defined between service and client, it has to be communicated through other means such as documentation or emails.

- Since it works on HTTP, there can’t be asynchronous calls.
- Sessions can’t be maintained.

15 . What are the DRY and DIE principles?

Answer

In software engineering, **Don't Repeat Yourself (DRY)** or **Duplication is Evil (DIE)** is a principle of software development.

16 . What are the differences between *Continuous Integration*, *Continuous Delivery*, and *Continuous Deployment*?

Answer

- Developers practicing **continuous integration** merge their changes back to the main branch as often as possible. By doing so, you avoid the integration hell that usually happens when people wait for release day to merge their changes into the release branch.
- **Continuous delivery** is an extension of continuous integration to make sure that you can release new changes to your customers quickly in a sustainable way. This means that on top of having automated your testing, you also have automated your release process and you can deploy your application at any point of time by clicking on a button.
- **Continuous deployment** goes one step further than continuous delivery. With this practice, every change that passes all stages of your production pipeline is released to your customers. There's no human intervention, and only a failed test will prevent a new change to be deployed to production.

18 .What is the difference between `JOIN` and `UNION`?

Answer

`UNION` puts lines from queries after each other, while `JOIN` makes a *cartesian* product and subsets it -- completely different operations. Trivial example of `UNION`:

```
mysql> SELECT 23 AS bah
    -> UNION
    -> SELECT 45 AS bah;
+-----+
| bah |
+-----+
|  23 |
|  45 |
+-----+
2 rows in set (0.00 sec)
```

similary trivial example of `JOIN`:

```
mysql> SELECT * FROM
```

```
    -> (SELECT 23 AS bah) AS foo
    -> JOIN
    -> (SELECT 45 AS bah) AS bar
    -> ON (33=33);
+-----+-----+
| foo | bar |
+-----+-----+
|  23 |  45 |
+-----+-----+
1 row in set (0.01 sec)
```

19.What do you understand by Distributed Transaction?

Answer

Distributed Transaction is any situation where a single event results in the mutation of two or more separate sources of data which cannot be committed atomically. In the world of microservices, it becomes even more complex as each service is a unit of work and most of the time multiple services have to work together to make a business successful.

20.When to use Redis or MongoDB?

Answer

- **Use MongoDB if you don't know yet how you're going to query your data or what schema to stick with.** MongoDB is suited for Hackathons, startups or every time you don't know how you'll query the data you inserted. MongoDB does not make any assumptions on your underlying schema. While MongoDB is schemaless and non-relational, this does not mean that there is no schema at all. It simply means that your schema needs to be defined in your app (e.g. using Mongoose). Besides that, MongoDB is great for prototyping or trying things out. Its performance is not that great and can't be compared to Redis.
- **Use Redis in order to speed up your existing application.** It is very uncommon to use Redis as a standalone database system (some people prefer referring to it as a "key-value"-store).

21 . Explain the purpose of the backend?

Answer

The backend, also known as the server-side, is the software that powers a website or app. It's responsible for storing and organizing data, handling user requests, and delivering content to the front end.

22. What is a typical workflow for implementing a new feature on the backend?

Workflows for implementing features on the backend can vary depending on the company and the technology stack. However, a typical workflow would involve discussing the feature with the stakeholders, designing and prototyping the feature, writing the code, and doing Quality Assurance (QA) testing. In most cases, the backend developer will work with the front-end developer to ensure that data is correctly transmitted between the client and server. It's also essential to ensure that any new features are backward compatible with previous versions of the application.

23. Explain the essence of DRY and DIE principles?

The DRY (Don't Repeat Yourself) principle is a software development principle that states that developers should not duplicate code. Duplicated code can lead to maintenance issues because changes need to be made in multiple places. The DIE (Duplication Is Evil) principle is similar to the DRY principle. Still, it takes it one step further by stating that even slight amounts of duplication are inadequate and should be avoided.

24. What is a web server?

A web server is a computer that stores and delivers web pages. When you type a URL into your browser, the browser contacts the web server and requests the page. The web server then sends the page back to the browser, which displays it on your screen. Apache and NGINX are some of the most popular web servers used by backend applications.

Web servers can also host other resources, like images or videos.

25. What is the difference between a GET and a POST request?

A GET request retrieves data from a server, whereas a POST sends data to a server. With a GET request, parameters get passed in the URL. With a POST request, parameters get passed in the request's body.

26. What is an example of when you would use caching?

Caching is often used to improve the performance of an application. For example, frequently accessed information from a database may be cached to avoid unnecessary queries.

27. How would you select a cache strategy (e.g., LRU, FIFO)?

Selecting a cache strategy depends on the application's specific needs. For example, LRU (Least Recently Used) is a good choice for applications where frequently accessed data is also likely to be reaccessed. FIFO (First In, First Out) is a good choice for an application where data expires after a particular time.

28. What are some common issues with ORMs?

Some common issues with ORMs include performance degradation, incorrect data mapping, and difficulty handling complex queries.

29. When should you use asynchronous programming?

Asynchronous programming is often used when there is a need to improve the performance of an application. For example, if an application needs to make many database queries, it may be beneficial to use asynchronous programming to avoid blocking the main thread.

30. What is the difference between promises and callbacks?

A promise is an object that represents the result of an asynchronous operation. A callback is a function that is invoked when an asynchronous operation completes.

31. What is a closure?

A closure is a function that accesses variables “outside” itself, in another function’s scope, even after the parent function has closed.

32. What is the difference between a Class and an Interface in Java?

A class is a blueprint for an object, whereas an interface is a contract that defines the methods that a class must implement.

33. What is continuous integration?

Continuous Integration (CI) is a software development practice in which developers regularly merge their code changes into a shared repository. CI helps to ensure the codebase is always stable and there are no conflicts between different branches of code.

34. What is a software development kit (SDK)?

SDK is a set of tools that helps developers build software applications. It typically includes a compiler, debugger, and other utilities. Some common SDKs used for backend development include the Java Development Kit (JDK) and the Python Software Development Kit (SDK).

35. What are the tradeoffs of client-side rendering vs. server-side rendering?

There are a few tradeoffs when deciding between client-side rendering and server-side rendering.

Client-side rendering can be more complex to set up because the application needs to be able to run in a browser. Meaning the code must be transpiled from a higher-level language (such as JavaScript) to a lower-level language (such as HTML). In addition, client-side rendering can be slower because the application needs to download all of the necessary resources before it can start rendering.

On the other hand, server-side rendering is typically easier to set up because developers can use any language capable of generating HTML. In addition, server-side rendering can be faster because the HTML can be generated on the server and then sent to the client.

Please note that these tradeoffs often depend on the site's complexity and function.

36. What are high-order functions? Why are they useful?

High-order functions are functions that take other functions as arguments. These functions help abstract common patterns of code. For example, a high-order function could create a function that logs the arguments it is called with. This would be useful for debugging purposes.

37. What is a microservice?

A microservice is a small, independent component of a more extensive application. Each microservice has its own functionality and can be deployed independently.

Microservices often build into large, complex applications that are easy to maintain and scale. One of the benefits of using microservices is that they can be written in different programming languages and deployed on different servers.

Common examples of microservices include user authentication, payment processing, and image manipulation.

38. How would you design an API?

When designing an API, it's helpful to consider the needs of the developers who use the API. The API should be easy to use and well-documented. It's also critical to consider the API's security and ensure that only authorized users can access the data. Additionally, the API should be able to handle a large number of requests without overloading the server.

39. What is the difference between a RESTful and a SOAP API?

RESTful APIs are designed to be easy to use and well-documented. They use a standard set of rules, which makes them easy to learn and use. On the other hand, SOAP APIs are designed to be more secure and can handle a larger number of requests. However, they are more complex to learn and use.

40. How do you handle errors when making API calls?

When making API calls, it's industry standard to handle errors in a way consistent with the rest of the application. For example, if the API returns a 404 error, you might want to display a message to the user saying the data could not be found.

41. What is a database?

A database is a place where information is stored. Backend applications use databases to store and retrieve data, like user information or app data. There are many different databases, but the most common ones backend applications use are relational databases like MySQL, PostgreSQL, and Oracle. Databases are usually managed by a database management system (DBMS), which provides an interface for administrators to manage the data. Some backend applications also use NoSQL databases, which are non-relational and often more scalable than relational databases.

42. How would you handle optimizing an existing database?

Once you've selected a database, it's vital to keep it optimized. This process can be done by periodically running maintenance tasks, such as indexing the data or purging old data that is no longer needed. Additionally, monitoring the database's performance and ensuring it can handle the application's load is crucial.

43. What is the difference between a relational and a non-relational database?

Relational databases store data in tables and use primary keys to identify each row. Non-relational databases, on the other hand, store data in documents and use object IDs to identify each record.

44. How would you query data from a MongoDB database?

MongoDB uses a query language called MongoDB Query Language (MQL). You would use the find() method to query data from a MongoDB database. This method takes a set of parameters that specify the criteria for the query. For example, to find all documents in the "users" collection that have a "firstName" of "John," you would use the following query:

```
db.users.find({"firstName": "John"})
```

45. What are some benefits of using a NoSQL database?

NoSQL databases have a few advantages over relational databases. They are generally more scalable and easier to manage. Additionally, they can be more flexible because they do not require a schema. However, NoSQL databases can be more difficult to query, and they often do not provide the same level of data consistency as relational databases.

46. How would you normalize data in a relational database?

To normalize data in a relational database, you would create separate tables for each entity type and use foreign keys to relate the data. For example, if you had a "users" table and an "orders" table, you would use a foreign key to link the data in the two tables.

47. How would you design a software system for scalability?

When designing a software system for scalability, it's essential to consider the application's needs. For example, if the application needs to handle many concurrent users, you might want to design the system using a microservices architecture. This architecture allows each component of the system to be scaled independently. Additionally, you might want to use a message queue to decouple the components of the system. This will allow each component to scale independently without affecting the performance of the other components.

48. What are some common scalability issues? How can they be addressed?

Some common scalability issues include performance degradation, data loss, and downtime. You can address these issues using various techniques, such as caching, sharding, and replication. Additionally, it is crucial to have a well-designed architecture that can handle increased loads.

Scalability is an essential consideration for any application. While a few common scalability issues can occur, you can solve them with proper infrastructure. By understanding these issues and how to solve them, you can ensure that your application will be able to scale as needed.

49. Scale-out vs. scale-up: how are they different? When to apply one, when the other?

Scale out and scale up are two different approaches to increasing the capacity or performance of a system. Scale-out involves adding additional components to the system, while scale-up involves making existing features more powerful.

Scale-out is usually the preferred approach when dealing with web applications or other systems that are highly parallelizable since it allows for near-linear increases in capacity. Scale-up may be more appropriate when working with legacy systems or those that are not efficiently parallelizable.

50. What are some common security risks when building a web application?

Some common security risks when building a web application include SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF).

SQL injection is a type of attack where malicious code is injected into an SQL statement, resulting in the execution of unintended actions. XSS attacks occur when malicious code is injected into a web page, resulting in the execution of unintentional actions. CSRF attacks happen when a malicious user tricks a victim into submitting a request that performs an unwanted activity, such as changing their password or transferring funds.

51. How would you implement authentication and authorization on a new project?

There are many ways to implement authentication and authorization on a new project. One way would be to use an existing third-party service, such as Auth0 or Okta. Another way would be to roll out your solution using JSON Web Tokens (JWTs) or similar technology.

Regardless, you need to create a login page where users can enter their credentials. Once the user's credentials were verified, you would generate a JWT and send it back to the user. The user would then need to send the JWT with each authentication request.

You would also need to implement an authorization system to check if the user has the correct permissions to access a particular resource. One way to do this would be to create roles and assign users to those roles. Then, you would use that to check the user's permissions when handling each request.

52. What is the difference between a cookie and a session?

A cookie is a small piece of data stored on the user's browser. A session is a server-side data structure that holds information about the user's current session.

Cookies store information such as the user's ID, language preference, or any other preferences. Sessions store information from a series of requests, such as the user's shopping cart or other information that needs to persist across multiple requests.

53. How would you unit test a new feature?

When testing a new feature, you would first need to write a test covering the new feature's functionality. You should isolate these tests from any other code in the system. Once you write the test, you need to run it and verify that it passes.

If the test fails, you need to debug the code and find the cause of the failure. Once you find the cause, you need to fix the code and re-run the test. If the test passes, you can then commit the code and move on to testing other features.

54. How would you integrate tests into your workflow?

When integrating tests into your workflow, you need to create a testing environment that mirrors the production environment as closely as possible. You need to set up this environment with all the necessary dependencies and data.

Once you have set up the testing environment, you need to write tests covering the system's functionality. These tests should be run automatically whenever new code gets pushed to the repository. If any of the tests fail, the build should be marked as failed, and the developers notified.

You should also manually run tests before each release. This will ensure that all of the functionality is working as expected and that there are no regressions.

55. What are some performance testing steps?

1. Identify the critical areas of the application that need testing.
2. Create test cases that exercise those areas of the application.
3. Run the test cases and collect data on the application's performance.
4. Analyze the data and identify any areas of improvement.

56. Why are TDD tests written before code?

TDD (Test-Driven Development) is a development methodology in which you write before code. The idea behind TDD is that by writing tests first, developers can ensure that their code meets the requirements. In addition, TDD can help to find bugs early and prevent them from being introduced into the code. However, TDD can be time-consuming and requires a good understanding of testing principles.

57. How would you deploy a new version of an application?

There are many ways to deploy a new version of an application. One way would be to use a tool like Ansible or Chef to automate the process. Another way would be to run the necessary scripts on each server manually.

If you use a tool like Ansible or Chef, you need to update the configuration files and run the tool. This process deploys the new code to all of the servers in the environment.

If you deploy the code manually, you must log in to each server and run the necessary scripts. Doing this updates the code on each server individually.

Once the new code gets deployed, you need to run your tests to ensure everything is working as expected. If there are any issues, you must roll back the changes and fix the problem, then redeploy the code.

58. How would you roll back a failed deployment?

If a deployment fails, you must roll back the changes and fix the problem. Once the problem is fixed, you can then redeploy the code.

To roll back a failed deployment, you need to undo any changes made during the deployment. This process could include reverting code changes, restarting services, or rolling back database changes. Once the changes are back, you can redeploy the code.

6 . Request – Response BEST Practices

- Request object
- Request object properties
- Request object methods
- Response object
- Response object properties
- Response object methods

Request object

Express.js is a request & response objects parameters of the callback function and are used for the Express applications. The request object represents the HTTP request and contains properties for the request query string, parameters, body, HTTP headers, etc.

Syntax : app.get('/', function (req, res) { })

Request Object Properties

These properties are represented below:

S.No	Properties	Description
1	req.app	Used to hold a reference to the instance of the express application.

2	req.body	Contains key-value pairs of data submitted in the request body. By default, it is undefined and middleware such as body-parser.
3	req.cookies	This property contains cookies sent by the request, used for the cookie-parser middleware.
4	req.ip	req.ip is remote IP address of the request.
5	req.path	req.path contains the path part of the request URL.
6	req.route	req.route is currently-matched route.

Request Object Methods

There are various types of request object method, these methods are represented below:

req.accepts (types)

It is used to the check content types are acceptable, based on the request accept HTTP header field.

Example :

```
req.accepts('html');
//=>?html?
req.accepts('text/html');
// => ?text/html?
Copy
```

req.get(field)

req.get is used to returns the specified HTTP request header field.

Example :

```
req.get('Content-Type');
// => "text/plain"
req.get('content-type');
// => "text/plain"
req.get('Something');
// => undefined
Copy
```

req.is(type)

If the incoming request is “CONTENT-TYPE”, this method returns true. HTTP header field matches the MIME type by the type parameter.

Example :

```
// With Content-Type: text/html; charset=utf-8
req.is('html');
req.is('text/html');
req.is('text/*');
// => true
Copy
```

req.param(name [, defaultValue])

req.param method is used to fetch the value of param name when present.

Example :

```
// ?name=sonia
          req.param('name')
```

```
          // => "sonia"
          // POST name=sonia
           req.param('name')
          // => "sonia"
          // /user/soniafor /user/:name
          req.param('name')
         // => "sonia"
Copy
```

Response Object

The response object specifies the HTTP response when an Express app gets an HTTP request. The response is sent back to the client browser and allows you to set new cookies value that will write to the client browser.

Response Object Properties

S.No	properties	Description
1	res.app	res.app is hold a reference to the instance of the express application that is using the middleware.
2	res.locals	Specify an object that contains response local variables scoped to the request.

Response Object Method

There are various types of response object method, these methods are represented below :

Response Append Method

Syntax : res.append(field, [value])

Response append method appends the specified value to the HTTP response header field. That means if the specified value is not appropriate so this method redress that.

Example :

```
res.append('Link', ['<http://localhost/>', '<http://localhost:3000/>']);
res.append('Warning', '299 Miscellaneous warning');
Copy
```

Response Attachment Method

Syntax : res.attachment(‘path/to/js_pic.png’);

Response attachment method allows you to send a file as an attachment in the HTTP response.

Example :

```
res.attachment('path/to/js_pic.png');    .
Copy
```

Response Cookie Method

Syntax: res.cookie(name, value [, options])

It is used to set a cookie name to value. The value can be a string or object converted to JSON.

Example :

```
res.cookie('name', 'alish', { domain: '.google.com', path: '/admin', secure:
true });
res.cookie('Section', { Names: [sonica,riya,ronak] });
res.cookie('Cart', { items: [1,2,3] }, { maxAge: 900000 });
Copy
```

Response Download Method

Syntax: res.download(path [, filename] [, fn])

Example:

```
res.download('/report-12345.pdf');
Copy
```

res.download method is transfer file at path as an “attachment” and the browser to prompt user for download.

Response End Method

Syntax: res.end([data] [, encoding])

Response end method is used to end the response process.

Example :

```
res.end();
res.status(404).end();
Copy
```

Response Get Method

Syntax : res.get(field)

res.get method provides HTTP response header specified by field.

Example :

```
res.get('Content-Type');
Copy
```

Response JSON Method

Syntax : res.json([body])

Response JSON method returns the response in JSON format.

Example :

```
res.json(null)
res.json({ name: 'alish' })
Copy
```

Response Render Method

Syntax : res.render(view [, locals] [, callback])

Response render method renders a view and sends the rendered HTML string to the client.

Example :

```
// send the rendered view to the client

res.render('index');

// pass a local variable to the view

res.render('user', { name: 'monika' }, function(err, html) {

 // ...

});
Copy
```

Response Status Method

Syntax : res.status(code)

res.status method sets an HTTP status for the response.

Example :

```
res.status(403).end();
res.status(400).send('Bad Request');
Copy
```

Response Type Method

Syntax : res.type(type)

res.type method sets the content-type HTTP header to the MIME type.

Example :

```
res.type('.html');              // => 'text/html'
res.type('html');               // => 'text/html'
res.type('json');               // => 'application/json'
res.type('application/json');   // => 'application/json'
res.type('png');                // => image/png:
```