HTTP defines 40 status codes to communicate the execution results of a client's request. The status code is divided into the following five categories.

- **1xx Informational**: Communicates transfer protocol level information.
- **2xx Success**: Communicates that the request from the client was successfully received, understood, and accepted.
- **3xx: Redirection**: Communicates that additional action needs to be taken by the user agent like browser in order to fulfil the request.
- **4xx Client Error**: Indicates errors caused by the client.
- **5xx Server Error**: Indicates that server is aware that an error occurred while processing the request and cannot process it further.

Normally, 2xx and 3xx status codes are treated as success codes. Any 4xx or 5xx status code is treated as an error code.

Table 3-3 lists the most commonly used success codes.

Table 3-3. *The Most Commonly Used Success Codes*

Status Code	Reason-Phrase	Meaning
200	OK	Indicates that the request has been processed successfully.
201	Created	Indicates that the request has been processed and a new resource has been created successfully.
202	Accepted	Indicates that the request has been received by the server and is being processed asynchronously.
204	No Content	Indicates that the response body has been purposely left blank.
301	Moved Permanently	Indicates that a new permanent URI has been assigned to the client's requested resource.
303	See Others	Indicates that the response to the request can be found in a different URI.
304	Not Modified	Indicates that the resource has not been modified for the conditional GET request of the client.
307	Use Proxy	Indicates that the request should be accessed through a proxy URI specified in the Location field.

Table 3-4 lists the most commonly used error codes.

Table 3-4. *The Most Commonly Used Error Codes*

Status Code	Reason Phrase	Meaning
400	Bad Request	Indicates that the request had some malformed syntax error due to which it could not be understood by the server. Probable reason is missing mandatory parameters or syntax error.
401	Unauthorized	Indicates that the request could not be authorized, possibly due to missing or incorrect authentication token information.
403	Forbidden	Indicates that the request was understood by the server but it could not be processed due to some policy violation or the client does not have access to the requested resource.
404	Not Found	Indicates that the server did not find anything matching the request URI.
405	Method Not Allowed	Indicates that the method specified in the request line is not allowed for the resource identified by the request URI.
408	Request Timeout	Indicates that the server did not receive a complete request within the time it was prepared to wait.
409	Conflict	Indicates that the request could not be processed due to a conflict with the current state of the resource.
414	Request URI Too Long	Indicates that the request URI length is longer than the allowed limit for the server.
415	Unsupported Media Type	Indicates that the request format is not supported by the server.
429	Too Many Requests	Indicates that the client sent too many requests within the time limit than it is allowed to.
500	Internal Server Error	Indicates that the request could not be processed due to an unexpected error in the server.
501	Not Implemented	Indicates that the server does not support the functionality required to fulfill the request.

(*continued*)

Table 3-4. (*continued*)

Status Code	Reason Phrase	Meaning
502	Bad Gateway	Indicates that the server, while acting as a gateway or proxy, received an invalid response from the back-end server.
503	Service Unavailable	Indicates that the server is currently unable to process the request due to temporary overloading or maintenance of the server. Trying the request at a later time might result in success.
504	Gateway Timeout	Indicates that the server, while active as a gateway or proxy, did not receive a timely response from the back-end server.

Resource Representation Design

A REST API resource entity representation is used to convey the state of the resource. The message body of the request/response is used to convey the state of the resource entity. The client sends the resource entity to the server in the request message payload of a POST, PUT, or PATCH message. The server sends the resource entity state in the response message payload for a GET, POST, PUT, or optionally, DELETE request.

A text-based format is normally used to represent the resource state. JSON and XML are the most commonly used text formats for representing the state of the resource entity. JSON is lightweight and provides a simple way to represent a resource. Due to the seamless integration of JSON with the browser's native runtime environment, JSON is the preferred choice for data representation in the design of a REST API. XML, on the other hand, is verbose, hard to parse, hard to read, and its data model is not compatible with many programing languages. This makes JSON a preferred choice over XML for representing the resource entity for a REST API. Many popular API providers have already moved away from XML to the JSON format. However, if the API consumer base consists of a large number of enterprise customers, you still have to support the XML data format for your APIs.

As a general guideline, it is advisable to support JSON data format by default and provide additional support for the XML format, if required. With support for both JSON and XML formats, how does the client specify the preferred format for the response? There are the following options:

- Use the *'Accept'* header.
- Append `.json` or `.xml` extensions to the endpoint URL.
- Include a query parameter in the URL to specify the response format.

Of the three options, use of *'Accept'* header to specify the response message format is most preferred. The following are some of the basic best practices for the JSON format representation of the resource entity.

- JSON should be in a well-formed format, with the variable names and their values enclosed in double quotes.
- JSON names should use mixed lowercase and uppercase letters. Special characters should be avoided whenever possible. JSON names like `fooName` is preferred over `foo-Name` because it allows the use of the cleaner dot notation for property access in JavaScript.
- The *`'Content-Type'`* header in the message should be set to `application/json` when a JSON format payload is included in the message.

Hypermedia Controls and Metadata

HTTP headers in the request/response convey metadata about the messages and about the resource entity contained in the message. HTTP specification defines a set of standard headers that can be used for various purposes. The specification also allows extension mechanisms to include custom HTTP headers. HTTP headers are classified under four types.

- **Entity headers**: This type of header provides metainformation about the entity body or resource in the message. Information such as the allowed HTTP methods, the media type, size, and location of the resource entity or cache expiration date-time, and so forth, are some of the examples of `Entity Header` types.
- **General headers**: This type of header provides information that can be applicable for both request and response messages. Caching directive, connection information, message origination date-time, and any message transformation applied on the whole message, are some examples of `General Header` types.
- **Client request headers**: This type of header is included *only* in the request message sent by the client or browser to the server. Authorization information, user agent information, information about the character set, encoding, or language that the client can accept, are some examples of information provided by `Client Request` headers.
- **Server response headers:** This type of header is included *only* by the server in the response sent to the client. Information about the age of the response generated by origin server, ETag information for caching purposes, the duration for which the server is unavailable for the requesting client, are some examples of `Server Response` headers.

This section looks at the most commonly used HTTP headers and how they can be used to design a better RESTful interface.

Accept (Client Request Header)

The *Accept* header is used in the request message to specify the media types that are acceptable by the client for the response. It is a mechanism for the client application or browser to indicate to the server which MIME types it is expecting.

The client can specify a range of media types using an asterisk (*) or multiple media types using comma-separated values. Media ranges can be overridden by specific media ranges or specific media types. If more than one media range applies to a given type, the most specific reference has precedence.

For example,

```
Accept: text/*, application/xhtml+xml, application/xml;q=0.9, */*
```

has the following precedence:

1. `application/xml;q=0.9`
2. `application/xhtml+xml`
3. `text/*`
4. `*/*`

The client can specify its relative preference for a media type using an optional *q* parameter. The following is an example:

```
Accept: audio/*; q=0.3, audio/basic
```

These examples indicate that `audio/basic` is preferred, but any audio type is also acceptable if it is the best available after a 70% markdown in quality.

If no `Accept` header field is specified, then it is assumed that the client accepts all media types. If an `Accept` header field is present but the server cannot send a response that is acceptable according to the Accept field value, then the server should respond with a HTTP status code of *406 Not Acceptable.*

Accept-Charset (Client Request Header)

The *Accept-Charset* request header is used by the client to specify the character sets that it understands and therefore can be included by the server in the response. As with the `Accept` header, the client can specify multiple charsets in a comma-separated list.
A *q* value on a scale of 0 to 1 can also be included to specify the acceptable quality level for non-preferred character sets.

If the client does not include an `Accept-Charset` header in the request, it is assumed that any character set is acceptable. If a `Accept-Charset` header is present but the server cannot send a response that is acceptable according to the `Accept-Charset` header, then the server should send an error response with the *406 Not Acceptable* HTTP status code, though the sending of an unacceptable response is also allowed as per the HTTP specs.

The following is an example of Accept-Charset header:

```
Accept-Charset: iso-8859-5, unicode-1-1;q=0.8
```

Authorization (Client Request Header)

The `Authorization` header is used by the client to include authentication information needed to access a server resource. If the server needs authentication and the `Authorization` header is not present in the request or is having an incorrect value, the server should send an error response with a 401 Unauthorized HTTP status code. The server should also include the `WWW-Authenticate` header in the response, which indicates the authentication scheme(s) required. The authentication schemes can be basic or digest access.

The following is an example of *Authorization* header:

```
Authorization: BASIC Z3Vlc3Q6Z3Vlc3QxMjM=
```

Host (Client Request Header)

The *`Host`* request header specifies the server address and the port of the resource requested. A `Host` without any port information implies the default port. The default port is 80 for HTTP and 443 for HTTPS.

The following is an example of *Host* header:

```
Host: http://www.foo.com
```

Location (Server Response Header)

The *`Location`* response header is used by the server to redirect the recipient to a URI other than the request URI for completion. This header is returned by the server in the following two scenarios.

- When a new resource is created after the successful execution of a POST or a PUT request. In this scenario, the `Location` header contains the location information of the newly created resource and the HTTP response status code should be *201 Created*.
- When the resource has moved temporarily or permanently, or is the result of a request execution is available at a different location. In this scenario, the `Location` header contains the redirected URI and the HTTP response status code should be 3xx. The `Location` information is then used by the browser to load a different web page, as specified in the header, thus helping in automatic redirection.

The following is an example of *Location* header:

```
Location: http://www.foo.com/http/index.htm
```

ETag (Server Response Header)

The *ETag* (entity tag) response header provides a mechanism for the server to send information about the current state of the entity. It is an alphanumeric string that uniquely identifies a specific version of the resource. If the resource has changed, the ETag value changes. Hence, the ETag value can be compared to determine if the cached resource entity on the client side matches that on the server.

It is a mechanism used for web cache validation that allows a client to make conditional requests. It makes caches more efficient and saves bandwidth because the server does not need to send the full response if the content has not changed.

The following is an example of *ETag* header:

```
ETag: "686897696a7c876b7e"
```

Cache-Control (General Header)

The *Cache-Control* general header field is specifies instructions on caching response information by the client and/or any intermediary along the request/response chain. Directives contained in this header provide information about the cache-ability of the response. It specifies if the response can be cached or not. If yes, can it be cached in public or private cache? It also specifies if the cache can be archived and stored. This header also contains information about the maximum duration for which the response can be cached.

The following is an example of *Cache-Control* header:

```
cache-control: private, max-age=300, no-cache
```

Content-Type (General Header)

The *Content-Type* header specifies the media type of the payload included in the message.

The following is an example of *Content-type* header:

```
Content-Type: text/html; charset=ISO-8859-4
```

Header Naming Conventions

Earlier sections looked at the best practices for naming resources and URIs. For good API design, even the HTTP headers should be named according to a convention. This section looks at some of the recommended best practices for naming headers.

HTTP specifications provide names for all standard HTTP headers and their syntax. It also provides extension mechanisms to include custom headers, if required. The following conventions are recommended for naming custom HTTP headers.

- Historically, `X-` has been used as a prefix for naming non-standard custom headers. RFC 6648 has deprecated the use of this convention because it causes more problems than it solves. Hence, do not prefix custom header names with `X-` or similar constructs.
- Name custom headers meaningfully and with the assumption that all custom headers may become standardized, public, commonly deployed, or usable across multiple implementations.
- Use hyphens in header names if required; for example, `My-Header-Name`.
- Do not use spaces in header names.

Versioning

Versioning is one of the most important considerations for web API design. Regardless of the approach followed, REST APIs should always be versioned. It helps to develop APIs in an iterative approach.

There are multiple approaches for versioning an API. The following are some questions to ask when thinking about API versioning.

- Which versioning approach should be used?
- When should a new version of the API be created?
- How and where to indicate the version of the API?
- How many versions should be maintained?
- How long should the older versions of the API be maintained?
- What are the deprecation mechanisms for older versions?

This and many other considerations and approaches for API versioning are discussed later in this book.

Querying, Filtering, and Pagination

Enterprises use REST APIs to expose their data and services. The resource collection returned by REST API may be huge. Transmitting the entire payload over the network is heavy on the bandwidth. Additionally, processing an entire collection on the client side would be processor intensive. Since a UI can display only a limited amount of data, this becomes important from UI processing standpoint as well; for example, 20 results per page. Hence, the need arises to be able to query, filter, and paginate the response. The API should provide a mechanism for the consumer to specify the query parameters and filter criteria. They should also be able to specify a range of data to be returned in the response. The range can be in terms of the number of elements, a date and time range, or in terms of offset and a limit.

It is important to note that it is not mandatory to provide support for querying, filtering, and pagination for all REST APIs. This is a resource-specific requirement and by default is not required to be supported on all resources. Consider designing the API to support filtration and pagination only if the number of entities in the resource collection that can be returned by default is high. The API documentation should specify if these complex functionalities are available for any specific service.

Limiting via Query-String Parameters

Filtering and pagination for an API is best implemented by designing the API interface with `offset` and `limit` query-string parameters. The `offset` parameter indicates the beginning item number in a collection and the `limit` specifies the maximum number of items to return.

The following is an example:

```
GET http://www.foo.com/products?offset=0&limit=25
```

In this example, the `offset` value 0 and `limit` value 25 indicate to return the first 25 items in the list. If the number of items fetched from the back end is more than 25, only the first 25 are returned. To retrieve the next set of items, the client has to make another call with a changed value for `offset` (=25) and `limit` (=25). If the number of items in the list is less than 25, all the items are returned in the response. This approach helps implement pagination support in the API.

It is important to understand that `offset` and `limit` are query-string parameters and are not dictated by any standards or specifications. Hence, different API providers may implement the same concept by using different parameter names. `start`, `count`, `page`, and `rpp` (records per page) are other examples of query-string parameters that can be used to implement pagination. An API designer can name them anything to suite the business context.

Filtering

Filtering is an approach to restrict the results returned in the response by specifying additional search criteria. These search criteria must be met on the data returned in the result. The filtering can become complex if the API has to support a complicated set of search criteria. The filtering criteria is based on the resource attribute. The complexity increases if filtering involves a complex combination of comparison operators. However, filtration can be achieved by supporting simple criteria, such as *`starts-with`* or *`contains`*, and so forth.

The filtering criteria can be specified by using the `filter` query-string containing a delimiter-separated list of name/value pairs. The delimiters that have conventionally worked are the vertical bar (`|`) to separate individual filter phrases and a double colon (`::`) to separate the names and values. This approach supports a wide range of use cases for filtering and also makes the filter criteria user-readable. The following is an example:

```
GET http://www.foo.com/customers?filter="name::matt|city::delhi"
```

Note that the property names in the name/value pairs match the name of the properties returned by the service in the payload. Wild cards can also be included in the filter values by using the asterisk (*).

Filtering can be implemented for an API by using one of the following approaches.

- Map the filter criteria to the back-end database SQL queries and implement filters at the database layer. This would retrieve the data matching the criteria from the data store; the same can be passed to the client with minimal messaging.
- Implement filter criteria in the service implementation layer. The service accepts the filter criteria as inputs and applies them on the data fetched from the data store. This may be required when the search criteria is complex or requires some business logic to be executed on the data set returned from the data store.
- Implement filter criteria on the API's intermediary layer. In the event that there is no change to the database or service implementation layer, the filtering is done on the intermediary API node that is generally introduced for creating and exposing REST APIs. Implementation of the filter on the intermediary API node might be complex due to the limited programming support provided by these tools.

When deciding on which of these approaches to adopt, it is recommended to implement filtering as close to the resource data store as possible.

The Richardson Maturity Model

The Richardson Maturity Model defines the levels to assess the maturity of a REST API service. It defines the following four levels (0–3) based on services support for URI, HTTP verbs, and hypermedia.

- Level 0: Swamp of POX
- Level 1: Resources
- Level 2: HTTP verbs
- Level 3: Hypermedia controls

Figure 3-1 shows the three core technologies with which Richardson evaluates service maturity. Each layer builds on top of concepts and technologies of the layer below. The higher up the stack an application sits, and the more it employs the technologies in each layer, the more mature it is.

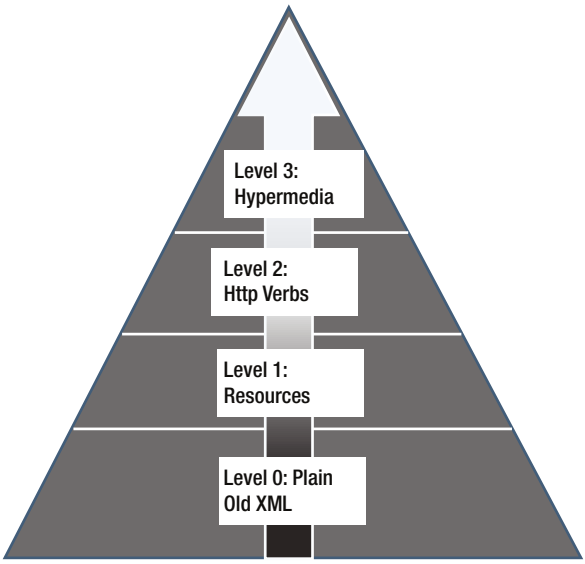

Figure 3-1. *Richardson's Maturity Model for REST APIs*

Let's look at each of these levels in detail.

Level 0: Swamp of POX (Plain Old XML)

This is the most basic level of maturity. At this level, the service is characterized as having a single URI that acts as the entry point. HTTP is used as the transport system for remote interactions. The payload content can be described in XML, JSON, YAML, key-value pairs, or any format of your choice. Normally, the POST method is used for sending the request to the server. SOAP and XML RPC are examples of services at Level 0 maturity. Figure 3-2 below shows a client making a request to an appointment service to get the availability of slots for a given date and doctor. The search parameters are sent in plain old XML format using POST request.

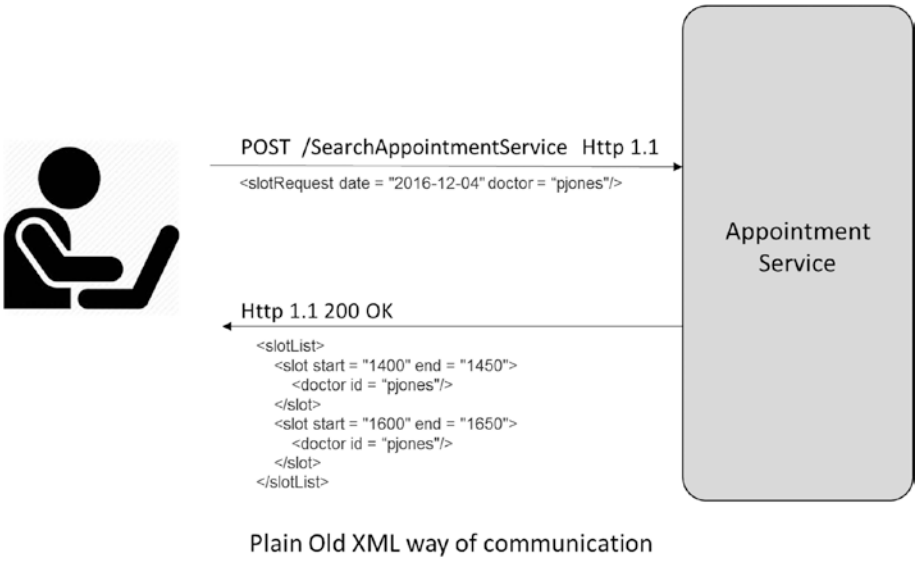

Figure 3-2. *Level 0- Plain Old XML way of communication*

Level 1: Resources

The first step toward RESTful maturity is the introduction of resources. At this level, instead of having a single URI as an endpoint for all services, you start interacting with individual resources through separate URIs. So instead of going through an endpoint like `http://www.foo.com/searchAppointmentService`, you start using resource URIs like `http://www.foo.com/api/doctors/{doctorId}`. Here `doctors` is a resource and you get access to an individual doctor's information by using `{doctorId}`. At this level, you still use POST as the only HTTP method for all of your communication. Figure 3-3 below shows a client making a request to an appointment service to get the availability of slots for a given date and doctor. The URL used to get the slot availability of the doctor is resource oriented.

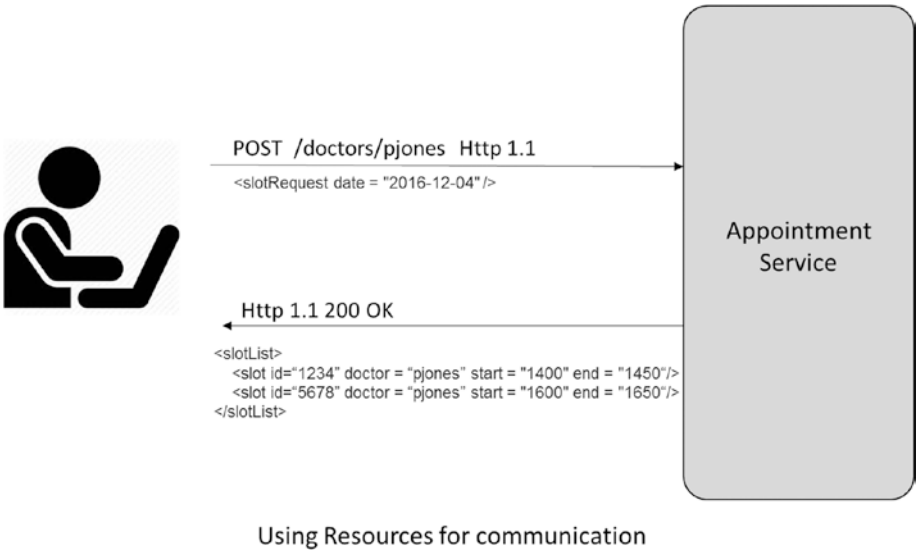

Figure 3-3. *Level 1- Using resources for communication*

Level 2: HTTP Verbs

At Level 0 and 1, the applications use the POST method for all communication. Level 2 maturity moves toward using the HTTP verbs more closely to how they are used in HTTP itself. To fetch the slot availability of a particular doctor, it should be using the HTTP verb GET at this level. As you've seen, the GET verb is safe because it is read-only and does not make any significant changes to the state of the resource. Hence, you can use the GET verb any number of times, in any order, and still get the same result every time, unless the resource has been modified using a different method. If you have to create a new appointment, you can use the POST method. If you want to update an existing appointment, you may use the PUT method.

In addition to the use of HTTP verbs, Level 2 also introduces the use of HTTP response codes to indicate the status of an operation on a resource. If a resource was successfully created, the service returns with HTTP response code 201. If the operation on a resource was successful, the 200 status code is used in the response. If the operation on a resource resulted in an error, an appropriate 4xx or 5xx response code should be used in the response. Figure 3-4 below shows a client making a request to an appointment service to get the availability of slots for a given date and doctor. 'GET' Http verb is used to access the resource oriented URL to get the appointment slots of the doctor. Http response code 200 OK is returned to indicate successful response.

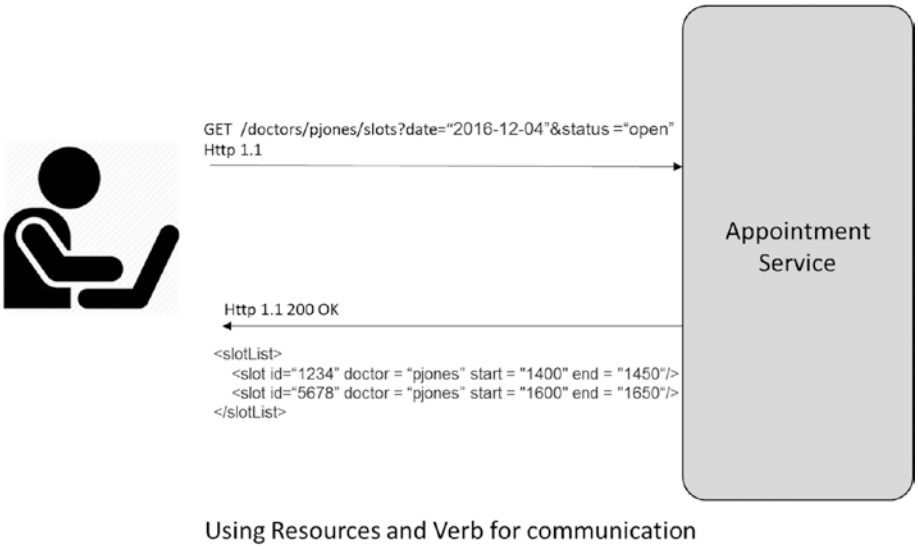

Figure 3-4. *Level 2- Using resources and verb for communication*

Level 3: Hypermedia Controls

This is the final level for REST maturity and it is where HATEOS enters the picture. It addresses the question of what to do next. After receiving the response for a service invocation, what are the next logical steps for the client? At a given node, what are the possible branches for traversal in a tree? This helps the client to be more intelligent and decide or prompt the user for the necessary possible actions.

At Level 3 maturity, the response of a REST service may contain a list of URIs. These URIs are the resources that the client wants to act upon as the next course of action. So rather than the client having to know where to post the next request, the hypermedia controls in the response tells how to do it. Figure 3-5 below shows a client making a request to an appointment service to get the availability of slots for a given date and doctor. The response returned for the GET request contains hyperlinks for the next possible actions that the client can do to book a slot.

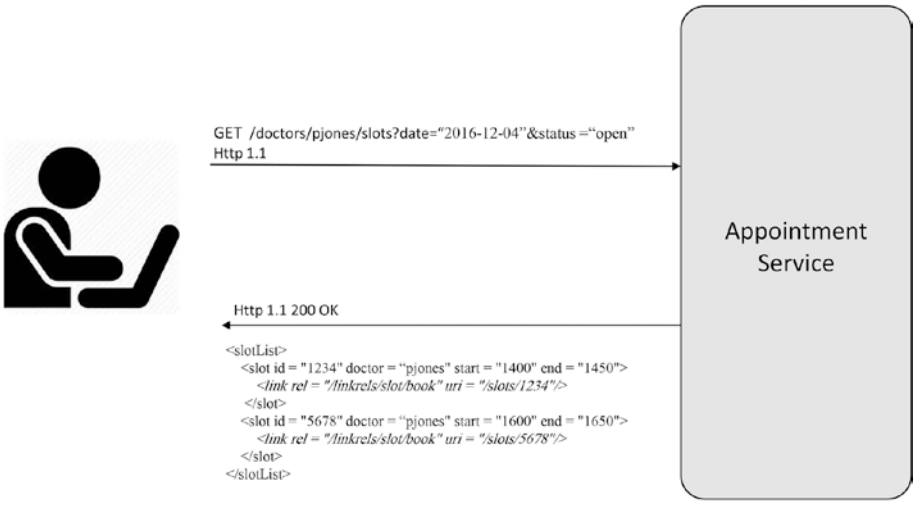

Figure 3-5. *Using Resource, Verb and HATEOAS for communication*

An obvious advantage of hypermedia controls is that it allows the server to change its URI scheme without breaking clients. It also helps client developers expose the protocol. The link gives client developers a hint on what the next possible options are. It may not provide all the information, but it at least gives developers a starting point to think about more information for the API and to look for a similar URI in the API documentation. Currently, there are no absolute standards on how to represent hypermedia controls. It is up to the service implementation team to decide how to implement HATEOS in their service.

As per Martin Fowler's article on Richardson Maturity Model, RMM provides a good way to think of the different elements of a RESTful service, but it is not a definition of levels of REST itself. Roy Fielding has made it clear that Level 3 RMM is a precondition of REST.

CHAPTER 4

API Documentation

Documenting a REST API is important for its successful adoption. APIs expose data and services that consumers want to use. An API should be designed with an interface that the consumer can understand. API documentation is key to the app developers comprehending the API. The documentation should help the developer to learn about the API functionality and enable them to start using it easily. This chapter looks at the aspects of documenting an API and some of the tools and technologies available for API documentation, including RAML, Swagger, API Blueprint, and others.

The Importance of API Documentation

As an API provider or developer, you may master your API. You have inside knowledge about its functionality, what it is supposed to do, how it is to be used, its security, limitations, error scenarios, and so forth. As an API provider, you have gradually learned everything about the API through various discussions, documentation, and references. However, this is not the case for the consumers of your API. The app developer community or API consumers look at the API's interface and wonder what the API does, how it should be used, what to expect when an error occurs, what security credentials to use, how and where to get the security credentials to use the API, and so forth. Hence, what is easy and simple to the API developer may not be intuitive to the API consumer. Good API documentation can help bridge the gap and make the API successful. API documentation communicates a vast amount of information about the API.

As enterprises move along in the digital transformation journey, there has been exponential growth in public and private APIs. In this competitive world, it is very likely that the data and services exposed by your API may also be exposed by another API provider. If the API is being monetized, it becomes more important to make it successful for your business. Good user-friendly API documentation is a key to its successful adoption. An API document is like an entrance into your API and provides a warm welcome to the API's consumers.

The API documentation should

- Get users started quickly
- Include useful and relevant information
- Provide sample code

B. De, *API Management*, DOI 10.1007/978-1-4842-1305-6_4

- Document a list of REST endpoints
- Document the message payload
- Provide Response status code and error messages

Audience for API Documentation

API documentation is used by various groups of people for various reasons. It is like a user manual for a product. Like a user manual, API documentation should have a quick-start guide, which quickly makes the first API call and lets consumers have a feel of it. At the next level, it should document the API's features, the resources and the APIs to access them, and finally, the error conditions for troubleshooting. Hence, the API documentation can be used primarily by the following types of audiences.

- **CTO**: Evaluates similar and competing APIs from a business, technology, and monetization perspective.
- **App or integration architect**: Explores the API to match the requirements for building an app or an integration solution.
- **App developer**: Wants to get started using the API with a quick-start guide and a detailed tutorial. Sample SDKs and API calls in the API documentation is of immense use to an app developer.
- **IT support specialist**: Supports the app and is interested in the error and troubleshooting information for debugging any issues with the app.

Model for API Documentation

A good API document communicates all information about the usage of the API— for both humans and machines. The API document should provide all necessary information to app developers or API consumers in a human-readable format. The documentation should help them assess its suitability for use in their client app. It should provide information about its licensing policy and usage requirements-input and output parameters, message format, error messages, and more. Similarly, the API interface should be documented such that its interface can be parsed by a machine to generate client stubs and server-side skeleton code that can be further developed. To make API documentation effective, it should include the following aspects about the API:

- Title
- Endpoint
- Method
- URL parameters
- Message payload

- Header parameters
- Response code
- Error code
- A sample request and response
- Tutorials and walkthrough
- Service-level agreement

Figure 4-1 shows an example of API documentation using Swagger.

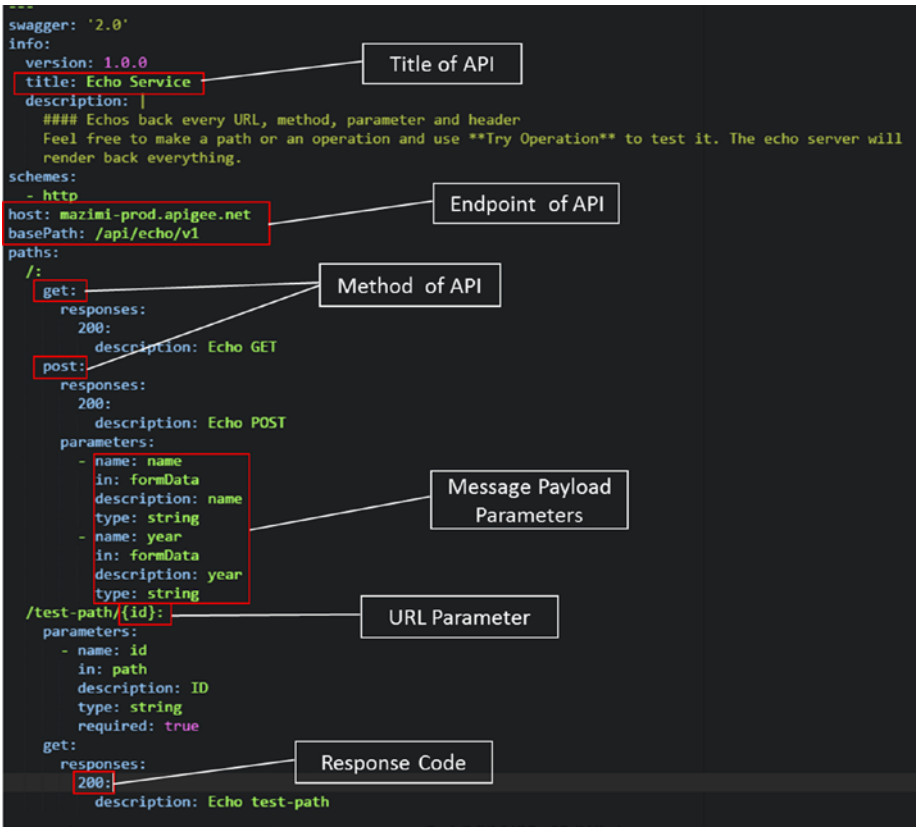

Figure 4-1. *API documentation using Swagger*

Title

The *title* should provide the name of the API, which can be used for its identification.

Endpoint

The *endpoint* is the entry point for the API. It defines the URL that clients need to use to invoke the API.

Method

The *method* defines the HTTP verbs used to access the API. GET, POST, PUT, and DELETE are the most common HTTP verbs used in a REST API. The client should specify the methods along with the URI to access the API. If an API supports multiple methods to be used for an URI, it should be specified in the API document as separate entities, as shown in Figure 4-1.

URL Parameters

The *URL parameters* define the parameter names and their format, which are used in the API call as a query string. The documentation should clearly state the purpose of each parameter, as well as which parameters are mandatory and which are optional. Any requirements for URL encode should be documented.

Message Payload

The *message payload* should specify the structure and format of the request and response message. JSON and XML are the most common formats used for a REST API. Other formats can be used as well. The message structure should specify the schema of the message payload. Any data constraints in the request payload should be documented. It is a good practice to include a table that provides the name, data type, description, and remarks, if any. Figure 4-2 shows a snippet of a Swagger format specification of an API, with the message format for a request and response payload.