1 RESTful Web Service Fundamentals

In this chapter, you will go through the fundamentals of RESTful APIs, or REST APIs for short, and their design paradigms. We will take a brief look at the history of REST, learn how resources are formed, and understand methods and status codes before we move on to explore **Hypermedia As The Engine Of Application State** (**HATEOAS**). These basics should provide a solid platform for you to develop a RESTful web service. You will also learn the best practices for designing **Application Programming Interfaces** (**APIs**).

This chapter will also introduce a sample e-commerce app, which will be used throughout the book as you learn about the different aspects of API development. In this chapter, we will cover the following topics:

- Introducing REST APIs
- Handling resources and **Uniform Resource Identifiers** (**URIs**)
- Exploring **Hypertext Transfer Protocol** (**HTTP**) methods and status codes
- Learning HATEOAS
- Best practices for designing REST APIs
- Overview of an e-commerce app (our sample app)

Technical requirements

This chapter does not require any specific software. However, knowledge of HTTP is necessary.

Introducing REST APIs

An API is the means by which a piece of code communicates with another piece of code. You might have already written an API for your code or used one in your programs; for example, in Java libraries for collection, input/output, or streams that provide a variety of APIs to perform specific tasks.

Java's SDK APIs allow one part of a program to communicate with another part of a program. You can write a function and then expose it with public access modifiers so that other classes can use it. That function signature is an API for that class. However, APIs that are exposed using these classes or libraries only allow internal communication inside a single application or individual service. So, what happens when two or more applications (or services) want to communicate with each other? In other words, you would like to integrate two or more services. This is where system-wide APIs help us.

Historically, there were different ways to integrate one application with another – RPC, **Simple Object Access Protocol** (**SOAP**)-based services, and more. The integration of apps has become an integral part of software architectures, especially after the boom of the cloud and mobile phones. You now have social logins, such as Facebook, Google, and GitHub, which means you can develop your application even without writing an independent login module and get around security issues such as storing passwords in a secure way.

These social logins provide APIs using REST and GraphQL. Currently, REST is the most widely used, and it has become a standard for writing APIs for integration and web app consumption. We'll also discuss GraphQL in detail in the final chapters of this book (in *Chapter 13*, *GraphQL Fundamentals*, and *Chapter 14*, *GraphQL Development and Testing*).

REST stands for REpresentational State Transfer, which is a style of software architecture. Web services that adhere to the REST style are called RESTful web services. In the following sections, we will take a quick look at the history of REST to understand its fundamentals.

REST history

Before REST adoption, when the internet was just starting to become widely known and Yahoo and Hotmail were the popular mail and social messaging apps, there was no standard software architecture that offered a homogenous way to integrate with web applications. People were using SOAP-based web services, which, ironically, were not simple at all.

Then came the light. Roy Fielding, in his doctoral research, *Architectural Styles and the Design of Network-Based Software Architectures* (`https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`), came up with REST in 2000. REST's architecture style allowed any server to communicate with any other server over the network. It simplified communication and made integration easier. REST was made to work on top of HTTP, which enables it to be used all over the web and in internal networks.

eBay was the first to exploit REST-based APIs. It introduced the REST API with selected partners in November 2000. Later, Amazon, Delicious (a site-bookmarking web app), and Flickr (the photo-sharing app) started providing REST-based APIs. In fact, **Amazon Web Services** (**AWS**) took advantage of Web 2.0 (with the invention of REST) and provided REST APIs to developers for AWS cloud consumption in 2006.

Later Facebook, Twitter, Google, and other companies started using it. Nowadays (in 2021), you will hardly find any web application developed without a REST API. Although, the GraphQL-based API for mobile apps is getting pretty close in terms of popularity.

REST fundamentals

REST works on top of the HTTP protocol. Each URI works as an API resource. Therefore, we should use nouns as endpoints instead of verbs. RPC-style endpoints use verbs, for example, `api/v1/getPersons`. In comparison, in REST, this endpoint could be simply written as `api/v1/persons`. You must be wondering, then, how can we differentiate between the different actions performed on a REST resource? This is where HTTP methods help us. We can make our HTTP methods act as a verb, for example, GET, DELETE, POST (for creating), PUT (for modifying), and PATCH (for partial updating). We'll discuss this in more detail later. For now, the `getPerson` RPC-style endpoint is translated into `GET api/v1/persons` in REST.

Note

The REST endpoint is a unique URI that represents a REST resource. For example, `https://demo.app/api/v1/persons` is a REST endpoint. Additionally, `/api/v1/persons` is the endpoint path and `persons` is the REST resource.

Here, there is client and server communication. Therefore, REST is based on the **Client-Server** concept. The client calls the REST API and the server responds with a response. REST allows a client (that is, a program, web service, or UI app) to talk to a remotely (or locally) running server (or web service) using HTTP requests and responses. The client sends to the web service with an API command wrapped in an HTTP request to the web. This HTTP request may contain a payload (or input) in the form of query parameters, headers, or request bodies. The called web service responds with a success/failure indicator and the response data wrapped inside the HTTP response. The HTTP status code normally denotes the status, and the response body contains the response data. For example, an HTTP status code of *200 OK* normally represents success.

From a REST perspective, an HTTP request is self-descriptive and has enough context for the server to process it. Therefore, REST calls are **stateless**. States are either managed on the client side or on the server side. A REST API does not maintain its state. It only transfers states from the server to the client or vice versa. Therefore, it is called REpresentation State Transfer, or REST for short.

It also makes use of HTTP cache control, which makes REST APIs **cacheable**. Therefore, the client can also cache the representation (that is, the HTTP response) because every representation is self-descriptive.

The following is a list of key concepts in REST:

- Resources and URIs
- HTTP methods

A sample REST call in plain text looks similar to the following:

```
GET /licenses HTTP/1.1
Host: api.github.com
```

Here, the `/licenses` path denotes the licenses resource. `GET` is an HTTP method. `1.1` at the end of the first line denotes the HTTP protocol version. The second line shares the host to call.

GitHub responds with a JSON object. The status is *200 OK* and the JSON object is wrapped in a response body, as follows:

```
HTTP/1.1 200 OK
date: Sun, 22 Sep 2020 18:01:22 GMT
content-type: application/json; charset=utf-8
server: GitHub.com
status: 200 OK
cache-control: public, max-age=60, s-maxage=60
vary: Accept, Accept-Encoding, Accept, X-Requested-With,
    Accept-Encoding
etag: W/"3cbb5a2e38ac6fc92b3d798667e828c7e3584af278aa3
      14f6eb1857bbf2593ba"
 … <bunch of other headers>
Accept-Ranges: bytes
Content-Length: 2507
X-GitHub-Request-Id: 1C03:5C22:640347:81F9C5:5F70D372
[
    {
         "key": "agpl-3.0",
         "name": "GNU Affero General Public License v3.0",
         "spdx_id": "AGPL-3.0",
        "url": "https://api.github.com/licenses/agpl-3.0",
        "node_id": "MDc6TGljZW5zZTE="
    },
    {
        "key": "apache-2.0",
        "name": "Apache License 2.0",
        "spdx_id": "Apache-2.0",
        "url": "https://api.github.com/licenses/
                apache-2.0",
        "node_id": "MDc6TGljZW5zZTI="
    },
    …
]
```

If you take note of the third line in this response, it tells you the value of the content type. It is good practice to have JSON as a content type for both the request and the response.

Handling resources and URIs

Every document on the **World Wide Web** (**WWW**) is represented as a resource in terms of HTTP. This resource is represented as a URI, which is an endpoint that represents a unique resource on a server.

Roy Fielding states that a URI is known by many names – a WWW address, a **Universal Document Identifier** (**UDI**), a URI, a **Uniform Resource Locator** (**URL**), and a **Uniform Resource Name** (**URN**).

So, what is a URI? A URI is a string (that is, a sequence of characters) that identifies a resource by its location, name, or both (in the WWW world). There are two types of URIs – URLs and URNs – as follows:

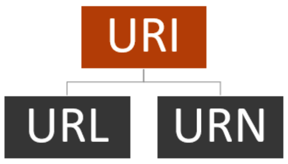

Figure 1.1 – The URI hierarchy

URLs are widely used and even known to non-developer users. URLs are not only restricted to HTTP; in fact, they are also used for many other protocols such as FTP, JDBC, and MAILTO. Therefore, a URL is an identifier that identifies the network location of a resource. We will go into more detail in the later sections.

The URI syntax

The URI syntax is as follows:

```
scheme:[//authority]path[?query][#fragment]
```

As per the syntax, the following is a list of components of a URI:

- **Scheme**: This refers to a non-empty sequence of characters followed by a colon (`:`). `scheme` starts with a letter and is followed by any combination of digits, letters, periods (`.`), hyphens (`-`), or plus characters (`+`).

  Scheme examples include HTTP, HTTPS, MAILTO, FILE, FTP, and more. URI schemes must be registered with the **Internet Assigned Numbers Authority** (**IANA**).

- **Authority**: This is an optional field and is preceded by `//`. It consists of the following optional subfields:

  a. **Userinfo**: This is a subcomponent that might contain a username and a password, which are both optional.

  b. **Host**: This is a subcomponent containing either an IP address or a registered host or domain name.

  c. **Port**: This is an optional subcomponent that is followed by a colon (`:`).
- **Path**: A path contains a sequence of segments separated by slash characters (`/`). In the preceding GitHub REST API example, `/licenses` is the path.
- **Query**: This is an optional component and is preceded by a question mark (`?`). The query component contains a query string of non-hierarchical data. Each parameter is separated by an ampersand (`&`) in the query component and parameter values are assigned using an equals (`=`) operator.
- **Fragment**: This is an optional field and is preceded by a hash (`#`). The fragment component includes a fragment identifier that gives direction to a secondary resource.

The following list contains examples of URIs:

- `www.packt.com`: This doesn't contain the scheme. It just contains the domain name. There is no port either, which means it points to the default port.
- `index.html`: This contains no scheme nor authority. It only contains the path.
- `https://www.packt.com/index.html`: This contains the scheme, authority, and path.

Here are some examples of different scheme URIs:

- `mailto:support@packt.com`
- `telnet://192.168.0.1:23/`
- `ldap://[2020:ab9::9]/c=AB?objectClass?obj`

From a REST perspective, the path component of a URI is very important because it represents the resource path and your API endpoint paths are formed based on it. For example, take a look at the following:

```
GET https://www.domain.com/api/v1/order/1
```

Here, `/api/v1/order/1` represents the path, and `GET` represents the HTTP method.

URLs

If you look closely, most of the URI examples mentioned earlier can also be called URLs. A URI is an identifier; on the other hand, a URL is not only an identifier, but it also tells you how to get to it.

As per **Request for Comments (RFC)**-3986 on URIs (`https://xml2rfc.tools.ietf.org/public/rfc/html/rfc3986.html`), the term URL refers to the subset of URIs that, in addition to identifying a resource, provide a means of locating the resource by describing its primary access mechanism (for example, its network "location").

A URL represents the full web address of a resource, including the protocol name (the scheme), the hostname port (in case the HTTP port is not `80`; for HTTPS, the default port is `443`), part of the authority component, the path, and optional query and fragment subcomponents.

URNs

URNs are not commonly used. They are also a type of URI that starts with a scheme – **urn**. The following URN example is directly taken from RFC-3986 for URIs (`https://xml2rfc.tools.ietf.org/public/rfc/html/rfc3986.html`):

```
urn:oasis:names:specification:docbook:dtd:xml:4.1.2
```

This example follows the `"urn:" <NID> ":" <NSS>` syntax, where `<NID>` is the NAMESPACE IDENTIFIER, and `<NSS>` is the Namespace-specific String. We are not going to use URNs in our REST implementation. However, you can read more about them at RFC-2141 (`https://tools.ietf.org/html/rfc2141`).

As per RFC-3986 on URIs (`https://xml2rfc.tools.ietf.org/public/rfc/html/rfc3986.html`): The term URN has been used historically to refer to both URIs under the "urn" scheme RFC-2141, which are required to remain globally unique and persistent even when the resource ceases to exist or becomes unavailable, and to any other URI with the properties of a name.

Exploring HTTP methods and status codes

HTTP provides various HTTP methods. However, you are primarily going to use only five of them. To begin with, you want to have **Create, Read, Update, and Delete** (**CRUD**) operations associated with HTTP methods:

- POST: Create or search
- GET: Read

- PUT: Update
- DELETE: Delete
- PATCH: Partial update

Some organizations also provide the `HEAD` method for scenarios where you just want to retrieve the header responses from the REST endpoints. You can hit any GitHub API with the `HEAD` operation to retrieve only headers; for example, `curl --head https://api.github.com/users`.

> **Note**
> REST has no such requirement that specifies which method should be used for which operation. However, widely used industry guidelines and practices suggest following certain rules.

Let's discuss each method in detail in the following sections.

POST

The HTTP POST method is normally what you want to associate with creating resource operations. However, there are certain exceptions when you might want to use the POST method for read operations. However, it should be put into practice after a well-thought-out process. One such exception is the search operation where filter criteria have too many parameters that might cross the GET call's length limit.

A GET query string has a limit of 256 characters. Additionally, the GET HTTP method is limited to a maximum of 2,048 characters minus the number of characters in the actual path. On the other hand, the POST method is not limited by the size of the URL for submitting name and value pairs.

You may also want to use the POST method with HTTPS for a read call if the submitted input parameters contain any private or secure information.

For successful create operations, you can respond with the *201 Created* status, and for successful search or read operations, you should use the *200 OK* or *204 No Content* status codes, although the call is made using the POST HTTP method.

For failed operations, REST responses may have different error status codes based on the error type, which we will look at later in this section.

GET

The HTTP GET method is what you usually want to associate with read resource operations. Similarly, you must have observed the GitHub GET `/licenses` call that returns the available licenses in the GitHub system. Additionally, successful GET operations should be associated with the *200 OK* status code if the response contains data, or *204 No Content* if the response contains no data.

PUT

The HTTP PUT method is what you usually want to associate with update resource operations. Additionally, successful update operations should be associated with a *200 OK* status code if the response contains data, or *204 No Content* if the response contains no data. Some developers use the PUT HTTP method to replace existing resources. For example, GitHub API v3 uses PUT to replace the existing resource.

DELETE

The HTTP DELETE method is what you want to associate with delete resource operations. GitHub does not provide the DELETE operation on the licenses resource. However, if you assume it exists, it will certainly look very similar to `DELETE /licenses/agpl-3.0`. A successful delete call should delete the resource associate with the `agpl-3.0` key. Additionally, successful DELETE operations should be associated with the *204 No Content* status code.

PATCH

The HTTP PATCH method is what you want to associate with partial update resource operations. Additionally, successful PATCH operations should be associated with a *200 OK* status code. PATCH is relatively new as compared to other HTTP operations. In fact, a few years ago, Spring did not have state-of-the-art support for this method for REST implementation due to the old Java HTTP library. However, currently, Spring provides built-in support for the PATCH method in REST implementation.

HTTP status codes

There are five categories of HTTP status codes, as follows:

- Informational responses (100–199)
- Successful responses (200–299)
- Redirects (300–399)

- Client errors (400–499)
- Server errors (500–599)

You can view a complete list of status codes at MDN Web Docs (`https://developer.mozilla.org/en-US/docs/Web/HTTP/Status`) or RFC-7231 (`https://tools.ietf.org/html/rfc7231`). However, you can find the most commonly used REST response status codes in the following table:

HTTP status code	Description
200 OK	For successful requests other than those already created.
201 Created	For successful creation requests.
202 Accepted	The request has been received but not yet acted upon. This is used when the server accepts the request, but the response cannot be sent immediately, for example, in batch processing.
204 No Content	For a successful operation that contains no data.
304 Not Modified	This is used for caching. The server responds to the client that the resource is not modified; therefore, the same cache resource can be used.
400 Bad Request	This is for a failed operation when input parameters are either incorrect or missing or the request itself is incomplete.
401 Unauthorized	This is for a failed operation due to unauthenticated requests. The specification says it's unauthorized, but semantically, it means unauthenticated.
403 Forbidden	This is for a failed operation when the invoker is not authorized to perform.
404 Not Found	This is for a failed operation when the requested resource doesn't exist.
405 Method Not Allowed	This is for a failed operation when the method is not allowed for the requested resource.
406 Not Acceptable	This is for a failed operation when the `Accept` header doesn't match. You can also use it when dependent operations are a barrier to the request process. For example, an order cannot be placed when a payment is rejected.
409 Conflict	This is for a failed operation when an attempt is made for a duplicate create operation.
429 Too Many Requests	This is for a failed operation when a user sends too many requests in a given amount of time ("rate limiting").
500 Internal Server Error	This is a failed operation due to a server error. This is a generic error.
502 Bad Gateway	This is for failed operations when the upstream server calls fail. For example, when an app calls a third-party payment service but the call fails.
503 Service Unavailable	This is for a failed operation when something unexpected has happened at the server, for example, an overload or a service fails.

Learning HATEOAS

With HATEOAS, RESTful web services provide information dynamically through hypermedia. Hypermedia is a part of the content that you receive from a REST call response. This hypermedia content contains links to different types of media such as text, images, and videos.

Hypermedia links can be contained either in HTTP headers or the response body. If you take a look at GitHub APIs, you will find that GitHub APIs provide hypermedia links in both headers and the response body. GitHub uses the header named "Link" to contain the paging-related links. Additionally, if you look at the responses of GitHub APIs, you'll also find other resource-related links with keys that have a postfix of `"url"`. Let's take a look at an example. We'll hit the `GET /users` resource and analyze the response:

```
$ curl -v https://api.github.com/users
```

This will give you the following output:

```
HTTP/1.1 200 OK
date: Mon, 28 Sep 2020 05:49:56 GMT
content-type: application/json; charset=utf-8
server: GitHub.com
status: 200 OK
cache-control: public, max-age=60, s-maxage=60
vary: Accept, Accept-Encoding, Accept, X-Requested-With,
      Accept-Encoding
etag: W/"6308a6b7274db1f1ffa377aeeb5359a015f69fa6733298938
      9453c7f20336753"
x-github-media-type: github.v3; format=json
link: <https://api.github.com/users?since=46>; rel="next",
<https://api.github.com/users{?since}>; rel="first"
… <Some other headers>
…
[
  {
    "login": "mojombo",
    "id": 1,
    "node_id": "MDQ6VXNlcjE=",
    "avatar_url": "https://avatars0.githubusercontent.com/
                 u/1?v=4",
    "gravatar_id": "",
    "url": "https://api.github.com/users/mojombo",
    "html_url": "https://github.com/mojombo",
    "followers_url": "https://api.github.com/users/mojombo/
                    followers",
      "following_url": "https://api.github.com/users/mojombo/
                        following{/other_user}",
      "gists_url": "https://api.github.com/users/mojombo/
                    gists{/gist_id}",
      "starred_url": "https://api.github.com/users/mojombo
                      /starred{/owner}{/repo}",
      "subscriptions_url": "https://api.github.com/users/
                            mojombo/subscriptions",
      "organizations_url": "https://api.github.com/
                            users/mojombo/orgs",
      "repos_url": "https://api.github.com/users/
                    mojombo/repos",
      "events_url": "https://api.github.com/users/mojombo
                     /events{/privacy}",
      "received_events_url": "https://api.github.com/users
                              /mojombo/received_events",
      "type": "User",
      "site_admin": false
    },
    {
      "login": "defunkt",
      "id": 2,
      "node_id": "MDQ6VXNlcjI=",
  …
  … <some more data>
  ]
```

In this code block, you'll find that the `"Link"` header contains the pagination information. Links to `"next"` page and `"first"` page are given as a part of the response. Additionally, you can find many URLs in the response body, such as `"avatar_url"` or `"followers_url"`, which provide links to other hypermedia.

REST clients should possess a generic understanding of hypermedia. Then, REST clients can interact with RESTful web services without having any specific knowledge of how to interact with the server. You just call any static REST API endpoint and you will receive the dynamic links as a part of the response to interact further. REST allows clients to dynamically navigate to the appropriate resource by traversing the links. It empowers machines, as REST clients can navigate to different resources in a similar way to how humans look at a web page and click on any link. Put simply, the REST client makes use of these links to navigate.

HATEOAS is a very important concept of REST. It is one of the concepts that differentiate REST from RPC. Even Roy Fielding was so concerned with certain REST API implementations that he published the following blog on his website in 2008: `REST APIs must be hypertext-driven`.

You must be wondering what the difference between hypertext and hypermedia is. Essentially, hypermedia is just an extended version of hypertext. As Roy Fielding states:

> *"When I say hypertext, I mean the simultaneous presentation of information and controls such that the information becomes the affordance through which the user (or automaton) obtains choices and selects actions. Hypermedia is just an expansion on what text means to include temporal anchors within a media stream; most researchers have dropped the distinction.*
>
> *Hypertext does not need to be HTML on a browser. Machines can follow links when they understand the data format and relationship types."*

Best practices for designing REST APIs

It is too early to talk about the best practices for implementing APIs. APIs are designed first and implemented later. Therefore, you'll find design-related best practices mentioned in the next sections. You'll also find best practices for going forward during the course of your REST API implementation.

1. Use nouns and not verbs when naming a resource in the endpoint path

We previously discussed HTTP methods. HTTP methods use verbs. Therefore, it would be redundant to use verbs yourself, and it would make your call look like an RPC endpoint, for example, `GET /getlicenses`. In REST, we should always use the resource name because, according to REST, you transfer the states and not the instructions.

For example, let's take another look at the GitHub license API that retrieves the available licenses. It is `GET /licenses`. That is perfect. Let's assume that if you use verbs for this endpoint, then it will be `GET /getlicenses`. It will still work, but semantically, it doesn't follow REST because it conveys the processing instruction rather than state transfer. Therefore, only use resource names.

However, GitHub's public API only offers read operations on the licenses resource, out of all the CRUD operations. If we need to design the rest of the operations, their paths should look like following:

- `POST /licenses`: This is for creating a new license.
- `PATCH /licenses/{license_key}`: This is for partial updates. Here, the path has a parameter (that is, an identifier), which makes the path dynamic. Here, the license key is a unique value in the license collection and is being used as an identifier. Each license will have a unique key. This call should make the update in the given license. Please remember that GitHub uses PUT for the replacement of the resource.
- `DELETE /licenses/{license_key}`: This is for retrieving license information. You can try this with any license that you receive in the response of the `GET /licenses` call. One example is `GET /licenses/agpl-3.0`.

You can see how having a noun in the resource path with the HTTP methods sorts out any ambiguity.

2. Use the plural form for naming the collection resource in the endpoint path

If you observe the GitHub license API, you might find that a resource name is given in the plural form. It is a good practice to use the plural form if the resource represents a collection. Therefore, we can use `/licenses` instead of `/license`. A `GET` call returns the collection of licenses. A `style` call creates a new license in the existing license collection. For `delete` and `patch` calls, a license key is used to identify the specific license.

3. Use hypermedia (HATEOAS)

Hypermedia (that is, links to other resources) makes the REST client's job easier. There are two advantages if you provide explicit URL links in a response. First, the REST client is not required to construct the REST URLs on their own. Second, any upgrade in the endpoint path will be taken care of automatically and this, therefore, makes upgrades easier for clients and developers.

4. Always version your APIs

The versioning of APIs is key for future upgrades. Over time, APIs keep changing, and you may have customers who are still using an older version. Therefore, you need to support multiple versions of APIs.

There are different ways you can version your APIs, as follows:

- **Using headers**: The GitHub API uses this approach. You can add an `Accept` header that tells you which API version should serve the request; for example, consider the following:

  ```
  Accept: application/vnd.github.v3+json
  ```

  This approach gives you the advantage of setting the default version. If there is no `Accept` header, it should lead to the default version. However, if a REST client that uses a versioning header is not changed after a recent upgrade of APIs, it may lead to a functionality break. Therefore, it is recommended that you use a versioning header.
- **Using an endpoint path**: In this approach, you add a version in the endpoint path itself; for example, `https://demo.app/api/v1/persons`. Here, `v1` denotes that version 1 is being added to the path itself.

  You cannot set default versioning out of the box. However, you can overcome this limitation by using other methods, such as request forwarding. Clients always use the intended versions of the APIs in this approach.

Based on your preferences and views, you can choose either of the preceding approaches for versioning. However, the important point is that you should always use versioning.

5. Nested resources

Consider this very interesting question: how are you going to construct the endpoint for resources that are nested or have a certain relationship? Let's take a look at some examples of customer resources from an e-commerce perspective:

- `GET /customers/1/addresses`: This returns the collection of addresses for customer 1.
- `GET /customers/1/addresses/2`: This returns the second address of customer 1.
- `POST /customers/1/addresses`: This adds a new address to customer 1's addresses.

- `PUT /customers/1/address/2`: This replaces the second address of customer 1.
- `PATCH /customers/1/address/2`: This partially updates the second address of customer 1.
- `DELETE /customers/1/address/2`: This deletes the second address of customer 1.

So far so good. Now, can we can have an altogether separate addresses resource endpoint (`GET /addresses/2`)? It makes sense, and you can do that if there exists a relationship that requires it; for example, orders and payments. Instead of `/orders/1/payments/1`, you might prefer a separate `/payments/1` endpoint. In the microservice world, this makes more sense; for instance, you would have two separate RESTful web services for both orders and payments.

Now, if you combine this approach with hypermedia, it makes things easier. When you make a REST API request to customer 1, it will provide the customer 1 data and address links as hypermedia (that is, links). The same applies to orders. For orders, the payment link will be available as hypermedia.

However, in some cases, you might wish to have a complete response in a single request rather than using the hypermedia-provided URLs to fetch the related resource. This reduces your web hits. However, there is no thumb rule. For a flag operation, it makes sense to use the nested endpoint approach; for example, `PUT /gist/2/star` (which adds a star) and `DELETE /gist/2/star` (which undoes the star) in the case of the GitHub API.

Additionally, in some scenarios, you might not find a suitable resource name when multiple resources are involved, for example, in a search operation. In that case, you should use a `direct /search` endpoint. This is an exception.

6. Secure APIs

Securing your API is another expectation that requires diligent attention. Here are some recommendations:

- Always use HTTPS for encrypted communication.
- Always look for OWASP's top API security threats and vulnerabilities. These can be found on their website (`https://owasp.org/www-project-api-security/`) or the GitHub repository (`https://github.com/OWASP/API-Security`).
- Secure REST APIs should have authentication in place. REST APIs are stateless; therefore, REST APIs should not use cookies or sessions. Instead, these should be secure using JWT or OAuth 2.0-based tokens.

7. Documentation

Documentation should be easily accessible and up to date with the latest implementation with their respective versioning. It is always good to provide sample code and examples. It makes the developer's integration job easier.

A change log or a release log should list all of the impacted libraries, and if some APIs are deprecated, then replacement APIs or workarounds should be elaborated on inside the documentation.

8. Status codes

You might have already learned about status code in the *Exploring HTTP methods and status codes* section. Please follow the same guidelines discussed there.

9. Caching

HTTP already provides the caching mechanism. You just have to provide additional headers in the REST API response. Then, the REST client makes use of the validation to make sure whether to make a call or use the cached response. There are two ways to do it:

- **ETag**: ETag is a special header value that contains the hash or checksum value of the resource representation (that is, the response object). This value must change with respect to the response representation. It will remain the same if the resource response doesn't change.

Now, the client can send a request with another header field, called `If-None-Match`, which contains the ETag value. When the server receives this request and finds that the hash or checksum value of the resource representation value is different from `If-None-Match`, only then should it return the response with a new representation and this hash value in the ETag header. If it finds them to be equal, then the server should simply respond with a `304 (Not Modified)` status code.

- **Last-Modified**: This approach is identical to the ETag way. Instead of using the hash or checksum, it uses the timestamp value in RFC-1123 format (*Last-Modified: Wed, 21 Oct 2015 07:28:00 GMT*). It is less accurate than ETag and should only be used for the falling mechanism.

  Here, the client sends the `If-Modified-Since` header with the value received in the `Last-Modified` response header. The server compares the resource-modified timestamp value with the `If-Modified-Since` header value and sends a `304` status if there is a match; otherwise, it sends the response with a new `Last-Modified` header.

10. Rate limit

This is important if you want to prevent the overuse of APIs. The HTTP status code `429 Too Many Requests` is used when the rate limit goes over. Currently, there is no standard to communicate any warning to the client before the rate limit goes over. However, there is a popular way to communicate about it using response headers; these include the following:

- `X-Ratelimit-Limit`: The number of allowed requests in the current period
- `X-Ratelimit-Remaining`: The number of remaining requests in the current period
- `X-Ratelimit-Reset`: The number of seconds left in the current period
- `X-Ratelimit-Used`: The number of requests used in the current period

You can check the headers sent by the GitHub APIs. For example, they could look similar to the following:

- `X-Ratelimit-Limit`: 60
- `X-Ratelimit-Remaining`: 55
- `X-Ratelimit-Reset`: 1601299930
- `X-Ratelimit-Used`: 5

So far, we have discussed various concepts related to REST. Next, we will next move on to discuss our sample app.

An overview of the e-commerce app

The e-commerce app is a simple online shopping application. It provides the following features:

- A user can browse through the products.
- A user can add/remove/update the products in the cart.
- A user can place an order.
- A user can modify the shipping address.
- The application can only support a single currency.

E-commerce is a very popular domain. If we look at the features, we can divide the application into the following subdomains using bounded contexts:

- **Users**: This subdomain is related to users. We'll add the `users` RESTful web service, which provides REST APIs for user management.
- **Carts**: This subdomain is related to the cart. We'll add the `carts` RESTful web service, which provides REST APIs for cart management. Users can perform CRUD operations on cart items.
- **Products**: This subdomain is related to the products catalog. We'll add the `products` RESTful web service, which provides REST APIs to search and retrieve the products.
- **Orders**: This subdomain is related to orders. We'll add the `orders` RESTful web service, which provides REST APIs for users to place orders.
- **Payment**: This subdomain is related to payments. We'll add the `payment` RESTful web service, which provides REST APIs for payment processing.
- **Shipping**: This subdomain is related to shipping. We'll add the `shipping` RESTful web service, which provides REST APIs for order tracking and shipping.

Here's a visual representation of our app's architecture:

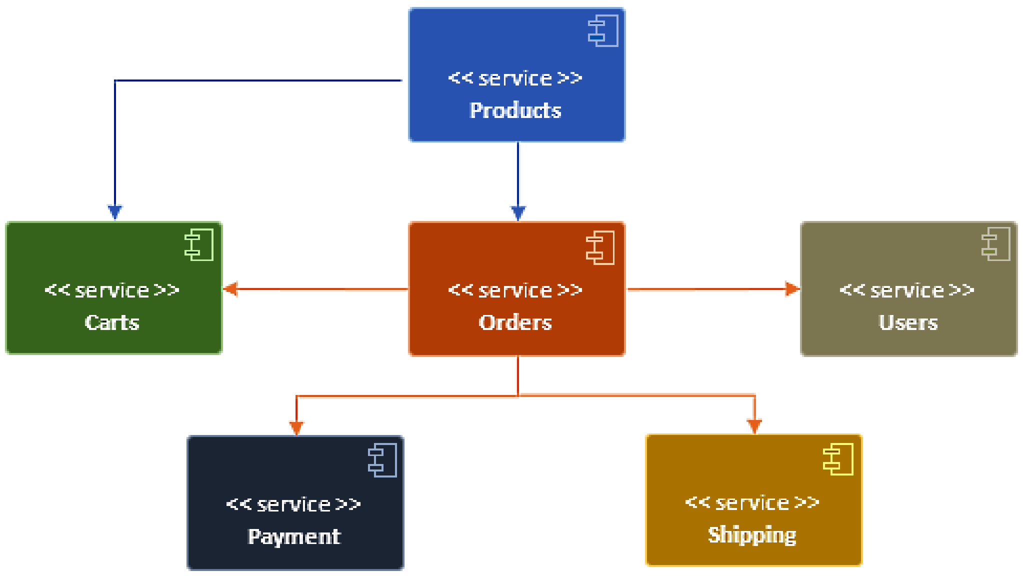

Figure 1.2 – The e-commerce app architecture

We'll implement a RESTful web service for each of the subdomains. We'll keep the implementation simple, and we will focus on learning these concepts throughout this book.

Summary

In this chapter, we learned the basic concepts of the REST architecture style. Now, you know how REST, which is based on HTTP, simplifies and makes integration easier. We also explored the different HTTP concepts that allow you to write REST APIs in a meaningful way. We also learned why HATEOAS is an integral part of REST implementation. Additionally, we learned the best practices for designing REST APIs. We also went through an overview of our e-commerce app. This sample app will be used throughout the book.

In the next chapter, you'll learn about the Spring Framework and its fundamentals.

Questions

1. Why have RESTful web services became so popular and, arguably, the industry standard?
2. What is the difference between RPC and REST?

3. How would you explain HATEOAS?
4. What error codes should be used for server-related issues?
5. Should verbs be used to form REST endpoints, and why?

Further reading

- *Architectural Styles and the Design of Network-based Software Architectures* can be found at `https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`.
- The URI Generic Syntax (RFC-3986) can be found at `https://tools.ietf.org/html/rfc3986`.
- The URN Syntax (RFC-2141) can be found at `https://tools.ietf.org/html/rfc2141`.
- HTTP Response Status Codes – RFC 7231 can be found at `https://tools.ietf.org/html/rfc7231`.
- HTTP Response Status Codes – Mozilla Developer Network can be found at `https://developer.mozilla.org/en-US/docs/Web/HTTP/Status`.
- *REST APIs must be hypertext-driven* can be found at `https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven`.
- RFC for the URI template can be found at `https://tools.ietf.org/html/rfc6570`.
- The OWASP API security project can be found at `https://owasp.org/www-project-api-security/` and `https://github.com/OWASP/API-Security`.