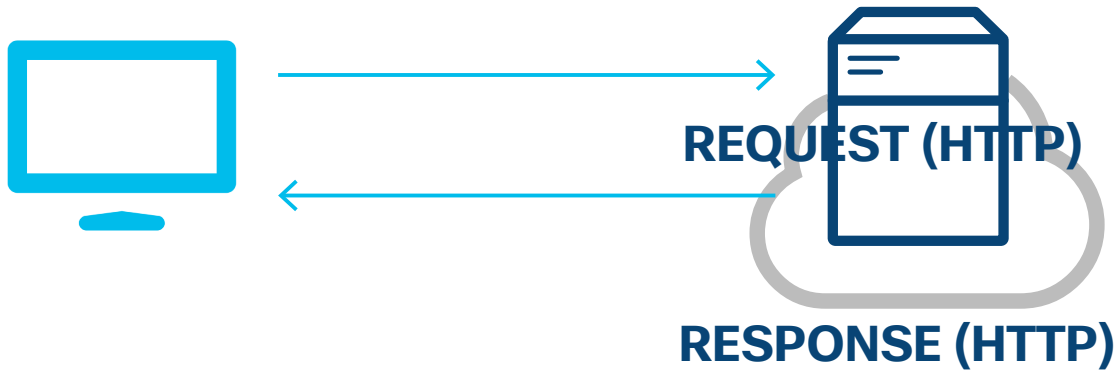
Introduction to REST APIs

4.4.1

REST Web Service APIs

REST API request/response model



A REST web service API (REST API) is a programming interface that communicates over HTTP while adhering to the principles of the REST architectural style.

To refresh your memory, the six principles of the REST architectural style are:

1. Client-Server
2. Stateless
3. Cache
4. Uniform Interface
5. Layered System
6. Code-On-Demand (Optional)

Because REST APIs communicate over HTTP, they use the same concepts as the HTTP protocol:

- HTTP requests/responses
- HTTP verbs
- HTTP status codes
- HTTP headers/body

4.4.2

REST API Requests

REST API requests

REST API requests are essentially HTTP requests that follow the REST principles. These requests are a way for an application (client) to ask the server to perform a function. Because it is an API, these functions are predefined by the server and must follow the provided specification.

REST API requests are made up of four major components:

- Uniform Resource Identifier (URI)
- HTTP Method
- Header
- Body

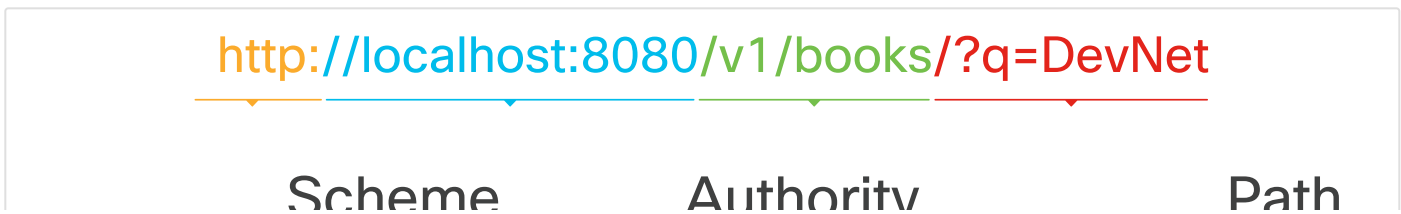
Uniform Resource Identifier (URI)

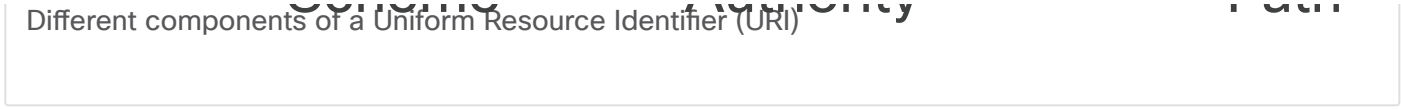
The Uniform Resource Identifier (URI) is sometimes referred to as Uniform Resource Locator (URL). The URI identifies which resource the client wants to manipulate. A REST request must identify the requested resource; the identification of resources for a REST API is usually part of the URI.

A URI is essentially the same format as the URL you use in a browser to go to a webpage. The syntax consists the following components in this particular order:

- Scheme
- Authority
- Path
- Query

When you piece the components together, a URI will look like this : `scheme:[//authority][/path][?query]`





Different components of a Uniform Resource Identifier (URI)

Scheme

The scheme specifies which HTTP protocol should be used. For a REST API, the two options are:

- http -- connection is open
- https -- connection is secure

Authority

The authority, or destination, consists of two parts that are preceded with two forward slashes (`//`):

- Host
- Port

The **host** is the hostname or IP address of the server that is providing the REST API (web service). The **port** is the communication endpoint, or the port number, that is associated to the host. The port is always preceded with a colon (`:`). Note that if the server is using the default port -- 80 for HTTP and 443 for HTTPS -- the port may be omitted from the URI.

Path

For a REST API, the path is usually known as the resource path, and represents the location of the resource, the data or object, to be manipulated on the server. The path is preceded by a slash (`/`) and can consists of multiple segments that are separated by a slash (`/`).

Query

The query, which includes the query parameters, is optional. The query provides additional details for scope, for filtering, or to clarify a request. If the query is present, it is preceded with a question mark (`?`). There isn't a specific syntax for query parameters, but it is typically defined as a set of key-value pairs that are separated by an ampersand (`&`). For example:

```
http://example.com/update/person?id=42&email=person%40example.com
```

HTTP method

REST APIs use the standard HTTP methods, also known as HTTP verbs, as a way to tell the web service which action is being requested for the given resource. There isn't a standard that defines which HTTP method is mapped to which action, but the suggested mapping looks like this:

HTTP Method	Action	Description

HTTP Method	Action	Description
POST	Create	Create a new object or resource.
GET	Read	Retrieve resource details from the system.
PUT	Update	Replace or update an existing resource.
PATCH	Partial Update	Update some details from an existing resource.
DELETE	Delete	Remove a resource from the system.

Header

REST APIs use the standard HTTP header format to communicate additional information between the client and the server, but this additional information is optional. HTTP headers are formatted as name-value pairs that are separated by a colon (:), [name]:[value]. Some standard HTTP headers are defined, but the web service accepting the REST API request can define custom headers to accept.

There are two types of headers: request headers and entity headers.

Request headers

Request headers include additional information that doesn't relate to the content of the message.

For example, here is a typical request header you may find for a REST API request:

Key	Example Value	Description
Authorization	Basic dmFncmFudDp2YWdyYW50	Provide credentials to authorize the request

Entity headers

Entity headers are additional information that describe the content of the body of the message.

Here is a typical entity header you may find for a REST API request:

Key	Example Value	Description
Content-Type	application/json	Specify the format of the data in the body

Body

The body of the REST API request contains the data pertaining to the resource that the client wants to manipulate. REST API requests that use the HTTP method POST, PUT, and PATCH typically include a body. Depending on the HTTP method, the body is optional, but if data is provided in the body, the data type

must be specified in the header using the Content-Type key. Some APIs are built to accept multiple data types in the request.

4.4.3

REST API Responses

REST API responses are essentially HTTP responses. These responses communicate the results of a client's HTTP request. The response may contain the data that was requested, signify that the server has received its request, or even inform the client that there was a problem with their request.

REST API responses are similar to the requests, but are made up of three major components:

- HTTP Status
- Header
- Body

HTTP status

REST APIs use the standard HTTP status codes in the response to inform the client whether the request was successful or unsuccessful. The HTTP status code itself can help the client determine the reason for the error and can sometimes provide suggestions for fixing the problem.

HTTP status codes are always three digits. The first digit is the category of the response. The other two digits do not have meaning, but are typically assigned in numerical order. There are five different categories:

- **1xx** - Informational
- **2xx** - Success
- **3xx** - Redirection
- **4xx** - Client Error
- **5xx** - Server Error

1xx - informational

Responses with a 1xx code are for informational purposes, indicating that the server received the request but is not done processing it. The client should expect a full response later. These responses typically do not contain a body.

2xx - success

Responses with a 2xx code mean that the server received and accepted the request. For synchronous APIs, these responses contain the requested data in the body (if applicable). For asynchronous APIs, the

responses typically do not contain a body and the 2xx status code is a confirmation that the request was received but still needs to be fulfilled.

3xx - redirection

Responses with a 3xx code mean that the client has an additional action to take in order for the request to be completed. Most of the time a different URL needs to be used. Depending on how the REST API was invoked, the user might be automatically redirected without any manual action.

4xx - client error

Responses with a 4xx code means that the request contains an error, such as bad syntax or invalid input, which prevents the request from being completed. The client must take action to fix these issues before resending the request.

5xx - server error

Responses with a 5xx code means that the server is unable to fulfill the request even though the request itself is valid. Depending on which particular 5xx status code it is, the client may want to retry the request at a later time.

Common HTTP status codes

HTTP Status Code	Status Message	Description
200	OK	Request was successful and typically contains a payload (body)
201	Created	Request was fulfilled and the requested resource was created
202	Accepted	Request has been accepted for processing and is in process
400	Bad Request	Request will not be processed due to an error with the request
401	Unauthorized	Request does not have valid authentication credentials to perform the request
403	Forbidden	Request was understood but has been rejected by the server
404	Not Found	Request cannot be fulfilled because the resource path of the request was not found on the server
500	Internal Server Error	Request cannot be fulfilled due to a server error
503	Service Unavailable	Request cannot be fulfilled because currently the server cannot handle the request

You can get details about each HTTP status code from the official registry of HTTP status codes, which is maintained by the Internet Assigned Numbers Authority (IANA). The registry also indicates which values are unassigned.

Header

Just like the request, the response's header also uses the standard HTTP header format and is also optional. The header in the response is to provide additional information between the server and the client in name-value pair format that is separated by a colon (`:`), [name]:[value].

There are two types of headers: response headers and entity headers.

Response headers

Response headers contain additional information that doesn't relate to the content of the message.

Some typical response headers you may find for a REST API request include:

Key	Example Value	Description
Set-Cookie	JSESSIONID=30A9DN810FQ428P; Path=/	Used to send cookies from the server
Cache-Control	Cache-Control: max-age=3600, public	Specify directives which MUST be obeyed by all caching mechanisms

Entity headers

Entity headers are additional information that describes the content of the body of the message.

One common entity header specifies the type of data being returned:

Key	Example Value	Description
Content-Type	application/json	Specify the format of the data in the body

Body

The body of the REST API response is the data that the client requested in the REST API request. The body is optional, but if data is provided in the body, the data type is specified in the header using the `Content-Type` key. If the REST API request was unsuccessful, the body may provide additional information about the issue or an action that needs to be taken for the request to be successful.

Response pagination

Some APIs, such as a search API, may need to send a huge amount of data in the response. To reduce the bandwidth usage on the network, these APIs will paginate the response data.

Response pagination enables the data to be broken up into chunks. Most APIs that implement pagination will enable the requester to specify how many items they want in the response. Because there are multiple chunks, the API also has to allow the requester to specify which chunk it wants. There isn't a standard way for an API to implement pagination, but most implementations use the query parameter to specify which page to return in the response. Take a look at the API's documentation to get the pagination details for the specific API you're using.

Compressed response data

When the server needs to send very large amounts of data that cannot be paginated, compressed data is another way to reduce the bandwidth.

This data compression can be requested by the client through the API request itself. To request a data compression, the request must add the `Accept-Encoding` field to the request header. The accepted values are:

- gzip
- compress
- deflate
- br
- identity
- *

If the server cannot provide any of the requested compression types, it will send a response back with a status code of `406 -- Not acceptable`.

If the server fulfills the compression, it will send the response back with the compressed data and add the `Content-Encoding` field to the response header. The value of the `Content-Encoding` is the type of compression that was used, enabling the client to decompress the data appropriately.

4.4.4

Using Sequence Diagrams with REST API

Sequence diagrams are used to explain a sequence of exchanges or events. They provide a scenario of an ordered set of events. They are also referred to as event diagrams. While a single REST API request may serve to obtain information or to initiate a change to a system, more commonly, interaction with a particular REST API service will be a sequence of requests. For that reason, sequence diagrams are frequently used to explain REST API request/response and asynchronous activity.

Formalized sequence diagrams are closely linked to and are considered a subset of a standardized modeling system known as Unified Modeling Language (UML). UML includes standardized approaches to other aspects of static resources and process, including standardized ways to diagram and explain user interfaces, class definitions and objects, and interaction behavior. Sequence diagrams are one way to diagram interaction behavior.

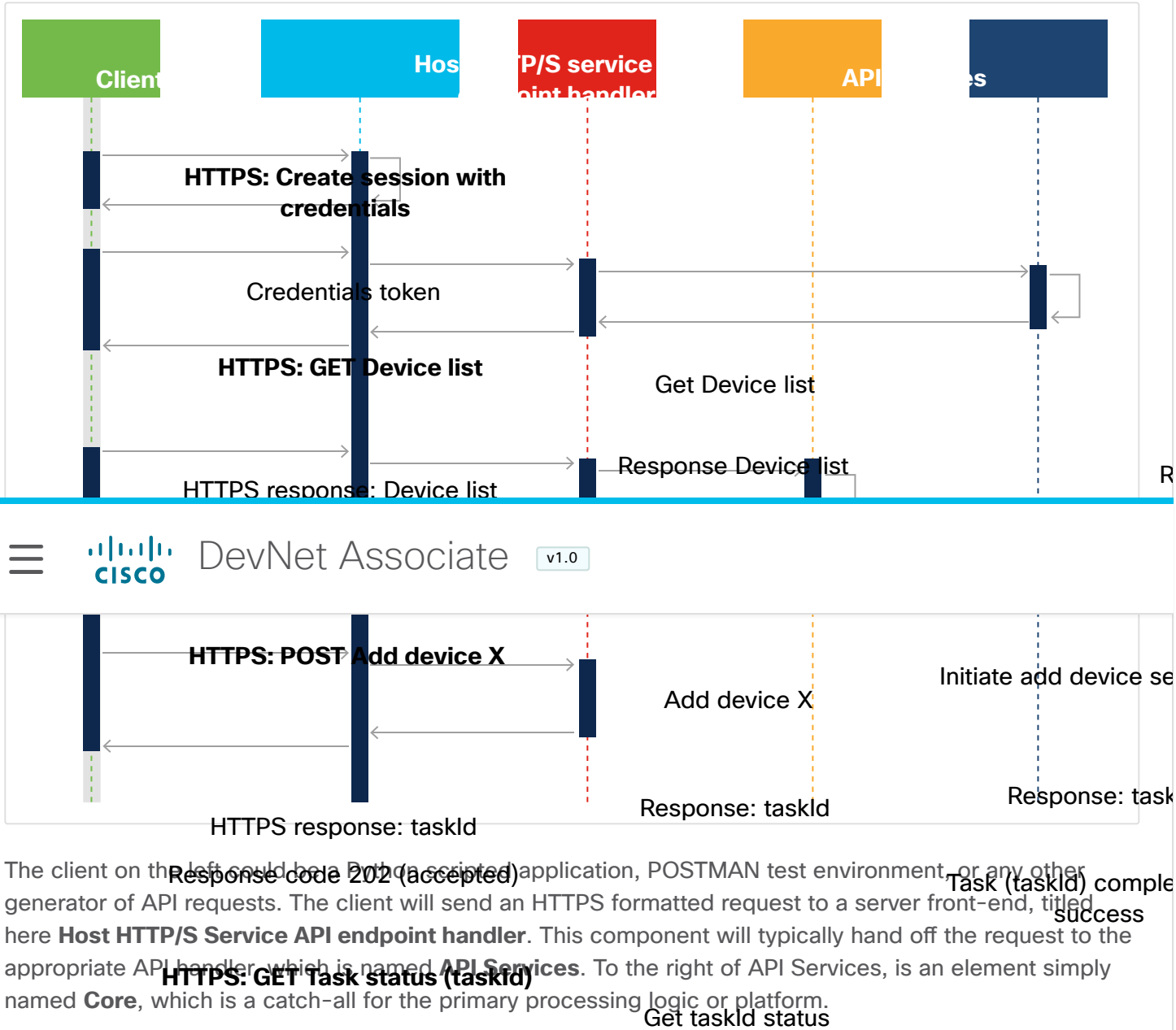
In a standard sequence diagram, such as the one below, the Y-axis is ordered and unscaled time, with zero time (*t*\=0) at the top and time increasing towards the bottom. If an arrow or an exchange is lower down in the diagram, it occurs after those that are above.

The X-axis is comprised of lifelines, represented by titled vertical lines, and exchanges or messages represented by horizontal arrows. A lifeline is any element that can interact by receiving or generating a message. By convention, front-end initiators of a sequence such as a user, a client, or a web browser are placed on the left side of the diagram. Elements such as file systems, databases, persistent storage, and so on, are placed to the right side. Intermediate services, such as a webserver or API endpoints, are arranged in the middle.

One of the useful aspects of sequence diagrams is that users of the diagrams can focus on the interaction between just two lifeline elements. For example, while the rest of a diagram might help set context, REST API users can focus on the interaction between the client and the front-end, or as shown below, the **Host HTTP/S Service API endpoint handler**.

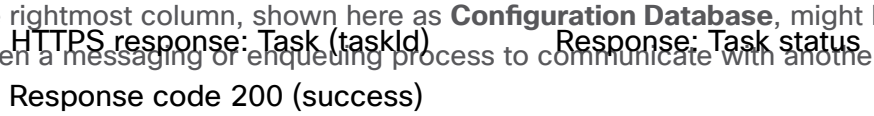
The API sequence below diagrams three simplified scenarios.

API Request/Response Sequence Diagram



The client on the left could be a Python scripted application, POSTMAN test environment, or any other generator of API requests. The client will send an HTTPS formatted request to a server front-end, titled here **Host HTTP/S Service API endpoint handler**. This component will typically hand off the request to the appropriate API handler, which is named **API Services**. To the right of API Services, is an element simply named **Core**, which is a catch-all for the primary processing logic or platform.

API Services may handle some requests directly or may interpret and then forward some or all of a request onto **Core**. The rightmost column, shown here as **Configuration Database**, might be any persistent storage – or even a messaging or enqueuing process to communicate with another system.



In this example, there are three separate sequences shown: create session, get devices, and create device.

Create Session: The starting request is labeled **HTTPS: Create Session w/credentials**. Here, the logic to create an HTTPS session is in the front-end, Host HTTP/S Service API endpoint handler. (This is shown graphically by use of the arrow that loops back.)

Get Devices: The second request from the client is to request a list of devices from the platform. In this sequence, the request is forwarded to the API Services module, which then contains the logic to query the configuration database directly, obtain a list of devices, and return the list to endpoint handler. The handler then wraps the content into an HTTPS response, and returns it to the client, along with an HTTP status code indicating Success.

These first two sequences demonstrate *synchronous* exchanges, in which a request is followed by a response, and the task is fully completed with Success or Failure.

Create Device: The third sequence starts with a POST request to create a device. In this case, the request migrates to **API Services**, which then forwards the request to the **Core** logic, which then starts to work on the request. The **API Services** wait only for a message from the **Core** acknowledging that the device creation process has begun. The **Core** provides a handle (TaskId) that identifies the work and allows follow up to see if the work was completed. The TaskId value propagates in a response back to the client. The HTTP response tells the client only the handle (TaskId) and that the request has been initiated with a response code of 202 (Accepted). This indicates that the request was accepted, and that the work is now in progress.

The **Core** logic continues to execute. It updates the **Configuration Database** and then informs the **API Services** when it is complete. At some later time, the client may choose to confirm that the task completed. The client does this with a Task Status query. **API Services** can then respond with information about the completion status, and the success or failure status for that task. Because the actual work requested was not completed prior to the response back to the client, this interaction is considered asynchronous.

4.3
API Architectural Styles

4.5
Authenticating to a REST API