Learning Objectives: Various API Topics

Learners will be able to…

- **use good naming practices for endpoints**
- **deploy safe strategies for upgrading APIs**
- **provide API users with ways to narrow down results of collection requests**

info

Make Sure You Know

This assignment assumes a basic knowledge of JavaScript, HTML and CSS.

Limitations

We will not explain simple programming concepts, it is assumed that the user is familiar with these concepts.

Naming Conventions and Documentation

Use consistent naming conventions for API endpoints

If you follow proper naming conventions for your API, it makes it easier for people who are accessing them.

- Use nouns to represent resource endpoints, not verbs.
- Do not include rest verbs (`get`, `post` etc) in your endpoint names.
- Use plural nouns for collections (e.g. `/pets/123` rather than `/pet/123`).
- Use American English spelling.
- Use consistent word separation scheme for the entities (for example, `kebab-case`).
- Use consistent naming in the query parameters (for example, `camelCase`).
- Nest related endpoints (for example `users/:username/appointments/:apptId`)

Documenting your API

Clients that wish to access an API need to know about the endpoints and the parameters required by those endpoints. API providers should provide all the information that is required in a clear format, so that API users do not need access to the source code to discover what API calls are available and how to use them.
* The OpenAPI 3 specification defines a way to describe the HTTP API interfaces for a service that is language agnostic.
* Swagger UI turns an OpenAPI specification into an interactive page where you can explore an API.

API Documentation

This example shows you an auto-generated swagger UI that we will edit to give more detail.

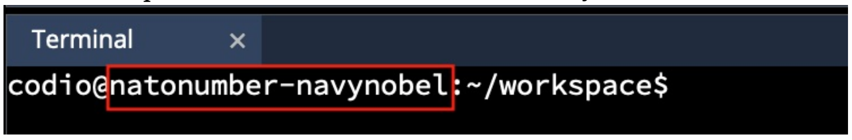
important

Installing the node modules

The first time you run something in each of the assignments in this course you need to make sure that you have all the proper node modules installed. To do that run the commands below in the terminal.

```
cd examples
npm i
```

Starting the App

1. Edit the file`swagger_output.json` and replace `nevercricket-viennahazard` with the two words you see in your terminal between the `@` and `:`.
   In the example below it would be `natonumber-navynobel`.
   

   ```
   codio@natonumber-navynobel:~/workspace$
   ```

2. Start the app by entering the following in the terminal. To stop the app use Ctrl-C in the terminal. You must be in the `examples` directory to run the command.

```
nodejs app.js
```

Select **Doc** from the Codio menu inside your browser pane to view the generated documentation

The file `swagger_output.json` was auto-generated by swagger and just contains the endpoints and parameters. This would be more helpful if the **description** fields were filled in.

1. Stop the app by typing Ctrl-C in the terminal window.

2. Add comments to the file `swagger_output.json`.

- Add a description of the API itself - “This API provides information about pets and checkups for a veterinarian office”.
- Describe the action GET for the endpoint `/pets` - “The GET action on this endpoint provides a list of all the pets clients, it takes no parameters.”.
- Describe the action POST for the endpoint `/pets` - “The POST action on this endpoint allows you to add a pet. The required parameters are ‘kind’ and ‘name’.”

3. View the API documentation again, with the comments. Enter `nodejs app.js` and click on **Doc** from the menu.

Backwards Compatibility and API Versions

You may need or want to change your web APIs at some point. It’s important that when you do this, you do not break client apps that depend on the old APIs.

API Evolution is the concept of never breaking your specification until you absolutely have to, and when you do, you manage that change with ample warnings to clients. The best practice is to add new fields, or add a new resource, and if you absolutely must you can deprecate you try remove APIs that are no longer being used.

Backwards compatibility

An API is considered **backwards compatible** if the applications that access it continue to run without changes when there is a new version.

Required elements for compatibility:
- Resources may be found at the same URIs at which they were previously found
- The error code values returned by existing APIs should remain the same
- If query parameters are added to existing APIs, they should not be mandatory
- Available actions on a resource should not be taken away. For example if a user could delete specific resources in the past, they should continue to be able to do so.
- Documented available media types for a URI should not be taken away and the default media type should not be changed.
- Parameters that have been added in the new API should be optional.
- Parameters should not be removed as that will break the calling application.

API Versioning Schemes

Versions are one way to ensure backwards compatibility. They allow you to maintain multiple versions of an API.

- URI Versioning
The advantage of this is that you can leave the existing APIs as they are and create new versions of an API by pre-pending a version number to the endpoint (e.g. `myapi.example.com/v1/pets` and `myapi.example.com/v2/pets`). The disadvantage of URI versioning is that it can lead to a very large API footprint. If you version one resource you have to version them all. You

cannot simply delete old versions when you come out with a new version because that would break any clients who access the old API. You have to have a strategy for getting rid of old versions, see **API Deprecation** below. Having multiple versions can also reduce efficiency with respect to HTTP caching, the cache will need to keep multiple versions of a resource.

- Media Type Versioning
In this versioning scheme, the media types of a resource are versioned and the types are custom names rather than generic ones such as application/json. The advantages of this scheme is that you can version at the resource level, you do not need to create a new version of the entire API. Some of the disadvantages are that the client will need to know the media type for each resource and be consistent about using that media type.

Example:
`tex -hide-clipboard HTTP/1.1 200 OK X-Powered-By: Express Content-Type: application/name.v1+json; charset=utf-8 Content-Length: 240 ETag: W/"f0-7Up1438Iu3S5DGObSecF2CXtaWs" Date: Wed, 08 Mar 2023 21:51:48 GMT Connection: keep-alive Keep-Alive: timeout=5`

- Custom Request Header Versioning
The version is part of the request header `API-Version: 2`. The advantage is you do not need to create a new set of URIs with this method.

- Domain (or Hostname) Versioning
This is similar to URI versioning in that it changes the URI `myapiv1.example.com/pets`. It has the same disadvantages as URI Versioning with the additional one that clients may need to get access to the new domain approved.

- Request Parameter Versioning
A parameter in the request is used to provide the version. The advantage of this scheme is that the parameter can be optional and the default can be the latest version `myapi.example.com/pets?version=2`.

- Date (or Dynamic Date Versioning
The timestamp of the first time a user accesses an API is saved with their other account data and that becomes the version they use when they make requests. This scheme doesn’t require changing endpoints and allows the API creator visibility into which versions of the API clients are using. This scheme can be difficult to implement.

API Deprecation

Sometimes an API endpoint, an API feature or an API version needs to be deprecated. Deprecation can be very disruptive to API users and should be avoided whenever possible but often it is necessary. It is asking a lot for a company to support every version of a product they have produced forever. You can see that with computer hardware, operating systems are always being upgraded and eventually they become too sophisticated for the hardware and the users of that version of the computer are told that new operating systems and software packages will no longer run on their system.

- Best practices for API deprecation
    - Announce the deprecation well in advance to give existing users time to update their applications.
    - Use the Deprecation HTTP Header Field to inform clients about when an API will be deprecated.
    - Inform your customers using email and/or blog posts about the upcoming deprecation.
    - Supply your customers with potential migration plans, for example to a new version.
    - Deprecated items should be documented in the API specification.

Filtering, Sorting and Paging

For API endpoint collections that are very large - **filtering**, **sorting** and **paging** capabilities are crucial.

Filtering Methods

Filtering can be added to query parameters, it allows the client to request a range of values from a collection, rather than an exact match.

- **LHS Brackets**
  One method of filtering output is the use of **Left Hand Side (LHS)** brackets, the **LHS** refers to the fact that they are placed on the left of the = operator. Operations such as [lte] - less than or equal and [gte] - greater than or equal can be used.

Example:

```
GET /users?age[gte]=18&price[lte]=22
```

This would return a list of all your users aged 18 to 22.

- **RHS Colon**
  Another method is **Right Hand Side (LHS)** colon, which is similar to the above method but the conditional is placed on the right of the = operator.

Example:

```
GET /users?age=gte:25&age=lte:30
```

This would return a list of all your users aged 25 to 30.

Pagination

Paging is useful for endpoints that return a long list of resources.
- Paging requires defining a consistent ordering scheme. It could be the item’s unique identifier or a different ordered field such as the creation date.
- Common pagination schemes:
1. **Offset - Limit Pagination** where offset is the starting point of the data and limit is the number of entries. This works very well with SQL as it has built in `OFFSET` and `LIMIT` keywords for that can be used with `SELECT` statements. The downside of this scheme is that if items have been added to the database between pagination calls, the pagination scheme can be thrown off. It can also be very inefficient with large databases because it is

fetching the rows that it is offsetting by even though it isn’t providing them.
1. **Cursor or Keyset Pagination** It uses the results of the previously returned page to set the start range for the next page and it provides a limit of entries per page. For examples if it is using a descending list of creation dates as the ordering scheme, it is uses the minimum creation date as the starting point for the next range.

On the left is an example of **Cursor Pagination** in `app.js` and a client that is accessing the server in `client.js`. This form of **Pagination** has the advantage that it provides **infinite scroll**. The disadvantage of this type of scrolling is that you can’t go directly to a specific page by number.

info

Starting the App

1. Install the node modules for the keyset-pagination app.

```
cd examples/keyset-pagination
npm i
```

2. Start the app by entering the following in the terminal. To stop the app use Ctrl-C in the terminal.

```
nodejs app.js
```

Try it out

3. Open another terminal window **Tools**>**Terminal** and try the client. You'll have to install the appropriate node modules in that directory as well.

```
cd examples/keyset-pagination-client
npm i
```

Run the client

```
nodejs client.js
```

The code will also filter for **kind** of pet so you can also try `nodejs client.js cat`.

In the lower left window you can see the SQL call for each page it displays.

Sorting

Sorting is another important feature for an endpoint that returns a lot of data. It is usually implemented by adding a `sort` or `sort_by` parameter to the URL and the order you are looking for ascending or descending.

```
GET /users?sort_by=last_name&order=asc
```