The OpenAPI Specification

- For our APIs to be usable, their capabilities have to be discoverable and understandable. In general terms, this means our API must be documented in some way.

- There is an argument to be made for documenting APIs in such a way as to make them understandable by both humans and machines.

- The case for human-readable API documentation should be obvious: humans will be writing client software to interact with the API, so they need to know how to use the API.

- There are many cases in which machine-readable API documentation is useful, though, too:
    - A machine could automatically generate interactive, human-readable documentation for the API.
    - A machine could generate client libraries or even server code for the API.
    - A machine could automatically generate testing code for the API.

- The OpenAPI Specification is a community-driven, standardized, programming language-agnostic, machine- and human-readable way to describe RESTful APIs. It gives us all of the benefits described above.

- There are many useful tools built around the OpenAPI Specification. One of the most popular is Swagger, which is actually a suite of tools for working with the OpenAPI Specification. Swagger includes tools for editing OpenAPI Specification, generating interactive-human readable documentation, generating server stubs and client library code, and testing APIs specified with OpenAPI.

- The combination of the OpenAPI Specification and Swagger is powerful and is increasingly becoming the standard way for developers to describe their APIs to the world, so it will be worthwhile to spend time learning how to use these tools as aides to designing, implementing, and deploying RESTful APIs.

Starting to write OpenAPI Specification with Swagger Editor

- Let's start to explore how to write an OpenAPI specification. We'll use the online version of Swagger Editor for this: https://editor.swagger.io.

- If you navigate to Swagger Editor using the URL above, you'll see an example OpenAPI specification (currently actually an older version of the OpenAPI Specification) and its corresponding automatically-generated docs. Spend a minute to peruse the specification and documentation to get a feel for the mapping between the two.

- We'll write an OpenAPI specification for the API for the Airbnb-like application we've been developing throughout the course, specifically focusing on the lodgings-related endpoints we implemented previously in our exploration of Express-based serving. As a refresher, here's a list of those API endpoints:
  - `GET /lodgings` – fetch a list of all lodgings
  - `POST /lodgings` – create a new lodging
  - `GET /lodgings/{lodgingID}` – fetch data about a single lodging
  - `PUT /lodgings/{lodgingID}` – modify a single lodging
  - `DELETE /lodgings/{lodgingID}` – delete a single lodging

- Clear out the example specification from the editor, and let's start to write the specification for our API. OpenAPI Specification can be written either in JSON or in YAML. Since YAML is less verbose and a little more human-readable, we'll use that. To start, we need to enter some basic information about our API:

```
openapi: 3.0.1
info:
  version: 1.0.0
  title: Book-a-Place API
  description: A simple API for an Airbnb-like application

paths: {}
```

- This specification should be mostly self-explanatory:

    - `openapi` – every OpenAPI Specification document needs this field to indicate the version of the OpenAPI Specification being used.
    - `info` – this required section specifies basic information about our API.
        - The `version` field is optional and indicates the version of our API.
        - The `description` field may contain CommonMark markdown.
        - There are other optional fields for specifying contact info, license, and other details.
    - `paths` – this section defines our API's individual endpoints, including the endpoint paths, the HTTP methods accepted by each endpoint, request and response bodies, and returned HTTP status codes.

- We've currently left the `paths` section empty so we can see our API's documentation automatically generated without error. We'll spend time exploring how to fill out the `paths` section just below.

- There are several other optional sections that can be included at the top level of the OpenAPI Specification document. We'll explore some of these later:
    - `servers` – specifies information about the API servers and base URLs
    - `components` – a section in which to place definitions that can be reused throughout the specification
    - `security` – a section in which API-wide security schemes (e.g. API key authorization) are described
    - `tags` – provides information about tags used to group API endpoints

Specifying API endpoints in the `paths` section

- The `paths` section of an OpenAPI Specification is where the bulk of the information about the API is specified.

- The `paths` section, as the name implies, is organized by API paths, e.g.:

```
paths:
  /lodgings:
    ...
  /lodgings/{lodgingID}:
    ...
  /reservations:
    ...
```

- Each path defined this way in the `paths` section is relative to the API server URL, which can optionally be specified in the top-level `servers` section or the specification and defaults to `/`.

- Under each path is specified information about the API endpoints associated with that path, i.e. all of the HTTP methods that can be used in requests to the path. Information about the request body, response body, HTTP status codes, etc. is specified for each method/path combination.

- Let's explore how this works by specifying information about our `/lodgings` path. There are two HTTP methods that can be used in requests to this path: GET and POST. We'll start by writing a specification for the `GET /lodgings` endpoint:

```
paths:
  /lodgings:
    get:
      ...
```

- Under the `get` section for the `/lodgings` path, we'll first add a (short) summary and (longer) description of the `GET /lodgings` endpoint:

```
summary: Fetch a list of lodgings
description: >
  Returns a paginated list of lodgings.
```

- Both of these fields are optional. The `description` field may be multi-line (a multi-line string in YAML can be indicated with the YAML "folded style" operator, `>`, as above) and supports CommonMark markdown.

- We can also assign an `operationID` to this endpoint:

```
operationId: getLodgings
```

- The `operationID` should be a string that uniquely identifies an API endpoint (i.e. no two `operationID`s may be the same). These IDs are used by some code generators to name methods corresponding to API endpoints and can also be used to define links between endpoints.

- Next, remember that our implementation of the `GET /lodgings` endpoint returned a paginated response and that the client could specify a query string parameter named `page` to request a specific page of lodgings. We can do this by adding a `parameters` section in our endpoint's specification:

```
parameters:
  - name: page
    in: query
    description: >
      Specifies a specific page of lodgings to request.
    schema:
      type: integer
      minimum: 1
      default: 1
```

- Note that the `parameters` section holds an array. In YAML, each item in an array are indicated with a dash, `-`, as above. Here's a little about the fields of the one array item specified above:
  - `name` – the name of the parameter, as specified in the query string
  - `in` – the value `query` indicates that the parameter goes in the query string
    - Parameters may also be specified as going in the URL path, in the request headers, or in a cookie.
  - `description` – a description of the parameter
  - `schema` – describes information about the value expected for this parameter. In this case, `page` should be an integer whose minimum value is 1 and whose default value (if the parameter is not specified) is also 1.

- Finally, we'll describe the possible responses from this endpoint in the `responses` section. Note that this is the *only* section that is required for all items in the `paths` section.

Endpoint responses: the `responses` section

- A given API endpoint can return many different kinds of responses, including successful responses and various responses indicating errors. Each of these different responses is indicated in the endpoint's `responses` section by its associated HTTP status code.

- Continuing the specification for our `GET /lodgings` endpoint, we have one possible response, which has a 200 HTTP status. Thus, the `responses` section for this endpoint would start out like this:

```
responses:
  200:
    ...
```

- Under the `200` section, we can specify information about the associated response. For example, we can provide a simple description:

```
description: Success
```

- In the case of our `GET /lodgings` endpoint, we'll also want to include a description of the response body, which will contain information about the returned lodgings. This will go within a `content` section underneath our `200` section.

- The OpenAPI specification allows us to indicate that an API endpoint can return responses with multiple different content type formats. For example, an API endpoint might return either JSON or XML content.

- Our API returns only JSON, so we will only include a specification for that content type, indicated by its complete media type:

```
content:
  application/json:
    ...
```

- Under the `application/json` section, we will specify the ***schema*** of the JSON response body, i.e. a data description of the response body. This will be similar to the simple schema we wrote above for our `page` query string parameter.

- At the top level, our `GET /lodgings` endpoint returns a JSON object, so we can begin our schema specification by indicating an `object` data type:

```
schema:
```

```
  type: object
  ...
```

- As a reminder, here is an example of the JSON object returned by the `GET /lodgings` endpoint (note that we're omitting the `links` property for now):

```
{
     "pageNumber": 1,
     "totalPages": 127,
     "pageSize": 10,
     "totalCount": 1264,
     "lodgings": [ ... ]
}
```

- In particular, this object contains a number of different properties, such as the page number being returned, the total number of pages, an array of lodgings, etc. We can include specifications for each of these properties within a `properties` section underneath `schema`.

- The specification of the first several properties of this response body, which describe information about pagination, is straightforward:

```
properties:
  pageNumber:
    type: integer
    description: Page number of returned lodgings.
    example: 1
  totalPages:
    type: integer
    description: Total number of pages available.
    example: 127
  pageSize:
    type: integer
    description: Number of lodgings per page.
    example: 10
  totalCount:
    type: integer
    description: Total number of lodgings.
    example: 1264
```

```
  ...
```

- Here, for each property (`pageNumber`, `totalPages`, `pageSize`, and `totalCount`), we indicate the data type and provide a description and an example value.

- Specifying the `lodgings` property is more interesting. We'll begin similar to the way we specified the properties above, specifying the data type and providing a description:

```
lodgings:
  type: array
  description: The returned lodgings.
  ...
```

- We want now to provide a specification of the *items* in this array, each of which represents an individual lodging. Because many of our API's endpoints will deal in one way or another with a representation of an individual lodging, we're going to write a single, *reusable* specification of this representation.

- Remember from above that the OpenAPI Specification provides a top-level section called `components` in which we can write specifications that can be reused throughout a specification document. Our reusable specification for the representation of an individual lodging will go in this section, specifically under a subsection called `schemas`.

- To finish up the specification of the `lodgings` property in the response of our API's `GET /lodgings` endpoint, we'll assume our reusable lodging specification is already written, and we'll make a reference to it to describe the individual items in the `lodgings` array:

```
items:
  $ref: '#/components/schemas/Lodging'
```

- This completes our specification of the `GET /lodgings` endpoint. Once we have the reusable specification for an individual lodging written, we'll be able to see the complete documentation for this endpoint in the Swagger Editor.

Writing reusable specifications in the `components` section

- The `components` section of an OpenAPI Specification document contains specifications that can be reused throughout the document.

- There are many different kinds of specifications that can be included in the `components` section. For now, we'll simply see how to write a reusable schema for an object representing an individual lodging. We'll call this specification `Lodging`, and it will go within the `schemas` section of the top-level components section of our OpenAPI Specification document:

```
components:
  schemas:
    Lodging:
      ...
```

    - Note that this hierarchy exactly matches the value of the `$ref` reference we used above.

- Underneath this `Lodging` section, we can simply write a schema to provide a data description of an individual lodging, similar to the way we wrote schemas above. We can start by giving our representation a description:

```
description: >
  An object representing information about a
  single lodging.
```

- As the description implies, our lodging representation will be an object with several properties:

```
type: object
  properties:
    ...
```

- Then, under the properties section, we can simply specify the individual properties of the object:

```
name:
  type: string
  description: Name of the lodging.
  example: My Cool Condo
description:
  type: string
  description: Textual description of the lodging.
  example: A nice place to stay, downtown near the river.
price:
  type: number
  description: Price of the lodging, per night, in USD.
  example: 128.00
  minimum: 0.01
```

- Finally, we can indicate that some of these properties are required (for example, when including a `Lodging` object in the body of a POST request) by including a required section under `Lodging`, e.g.:

```
required:
  - name
  - price
```

- This completes the schema for our `Lodging` object.

Finishing the endpoints for the `/lodgings` path

- Our `/lodgings` path has one more endpoint, which handles POST requests. This endpoint creates a new lodging and adds it to the application's database. The specification for this endpoint will begin much like the specification for the GET endpoint, with a summary, a description, and an `operationID`:

```
post:
  summary: Add a new lodging
  description: >
    Creates a new lodging with specified data and adds it
    to the application's database.
  operationId: addNewLodging
  ...
```

- Importantly, requests to this endpoint must have a request body containing a JSON representation of a new lodging. We can specify the format of this request body within a `requestBody` section similar to the way we specified response bodies above. We'll reuse our `Lodging` object schema here, and we'll also indicate that the request body is required:

```
requestBody:
  required: true
  content:
    application/json:
      schema:
        $ref: '#/components/schemas/Lodging'
```

- Our POST endpoint has two different responses: one to indicate success and one to indicate an error in which the request body was incorrectly formatted.

- The endpoint's success response returns a 201 HTTP status, and its body is a JSON object that simply contains the ID of the newly-created lodging (again, omitting HATEOAS links for now), e.g.:

```
{
     "id": 12345
}
```

- We can specify this response as follows:

```
responses:
  201:
    description: New lodging successfully added
    content:
      application/json:
        schema:
          type: object
          properties:
            id:
              type: integer
              description: ID of the created lodging.
              example: 12345
```

- This endpoint's error response sends a 400 status, and its response body is a JSON object indicating an error message, e.g.:

```
{
     "err": "Request needs a JSON body with a name field"
}
```

- An error object like this will be used in other places in our API, too, so let's make a reusable schema for it:

```
components:
  schemas:
    ...
    Error:
      description: >
        An object representing an error response from
        the API.
      type: object
      properties:
        err:
          type: string
          description: A message describing the error.
```

- With that schema in place, we can specify the error response for our `POST /lodgings` endpoint:

```
responses:
  ...
  400:
    description: Incorrectly-formatted request body
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/Error'
```

Parameterized paths and the `/lodgings/{lodgingID}` endpoints

- Our next three API endpoints involve the parameterized path `/lodgings/{lodgingID}`, where `{lodgingID}` is a parameter the client must specify to indicate the ID of a specific lodging with which to operate.

- In OpenAPI Specification, path parameters are indicated with curly braces `{}`:

```
paths:
  ...
  /lodgings/{lodgingID}:
    ...
```

- For the API endpoints associated with this path, we'll want to specify information about the path parameter `{lodgingID}`. Because this parameter is common to all of the endpoints associated with this path, we can actually define it at the path level instead of at the operation level:

```
/lodgings/{lodgingID}:
  parameters:
    - name: lodgingID
      in: path
      description: Unique ID of a lodging.
      schema:
        type: integer
      example: 12345
      required: true
```

- This parameter specification looks very much like our specification for the `page` query string parameter above. There are two differences to note:
  - We use the value `path` for the `in` field to indicate that this is a path parameter.
  - We specify `required: true` to indicate that this is a *required* parameter.

- With this parameter specified, we can move on to specify the three endpoints associated with this path. None of these endpoints introduces any new concepts. For completeness, we include their specifications below without comment.

GET `/lodgings/{lodgingID}`

```
get:
  summary: Fetch data for a specific lodging.
  description: >
    Returns complete data for a the lodging specified by
    `lodgingID`.
  operationId: getLodging
  responses:
    200:
      description: Success
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Lodging'
    404:
      description: Specified `lodgingID` not found
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
```

PUT /lodgings/{lodgingID}

```
put:
  summary: Update data for a specific lodging.
  description: >
    Replaces the data for the lodging specified by `lodgingID`.
  operationId: replaceLodging
  requestBody:
    description: >
      Lodging data to replace data for the lodging specified by
      `lodgingID`.
    required: true
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/Lodging'
  responses:
    200:
      description: Success
    400:
      description: Incorrectly-formatted request body
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
    404:
      description: Specified `lodgingID` not found
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
```

DELETE `/lodgings/{lodgingID}`

```
delete:
  summary: Remove a specific lodging from the database.
  description: >
    Completely removes the data for the lodging specified by
    `lodgingID`.
  operationId: removeLodging
  responses:
    204:
      description: Success
    404:
      description: Specified `lodgingID` not found
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
```

Grouping operations with tags

- The API we're working with here is somewhat limited, with only five total endpoints over one resource type. With the OpenAPI Specification we've written so far for this API, all of these endpoints are grouped together under a tag named "default", which you can see in the automatically-generated API documentation.

- For larger APIs, it is useful to group related API endpoints together. One common approach is to group endpoints by resource type.

- This kind of grouping can be achieved by assigning tags to the individual API endpoints. For example, by adding the following section to each API endpoint's specification (i.e. underneath `get`, `post`, `put`, `delete`, etc.), we can group all of our endpoints under the "lodgings" tag:

  ```
  tags:
    - lodgings
  ```

- Optionally, we can specify a top-level `tags` section to provide information about each specified tag, e.g.:

```
tags:
  - name: lodgings
    description: >
      API endpoints related to lodgings resources
```

Serving your OpenAPI Specification file

- It is common to have your API server serve the OpenAPI Specification file for your API at the path `/openapi.yaml` (or `/openapi.json` if you have a JSON specification).

- To do this, we can set our API's Express server up to serve static files out of a specified directory and then place our OpenAPI specification file in that directory.

- In particular, Express has a built-in middleware function named `static()` that we can use to serve static files.

- For example, we can add this line to our API server code to serve files out of a subdirectory called public/ that's in the same directory as our server code:

  ```
  app.use(express.static('public'));
  ```

- By registering this middleware function, if a request comes in whose path matches the name of one of the files in `public/`, the server will respond with the contents of that file.

- For example, say our `public/` directory contains the following files:
  - `openapi.yaml`
  - `openapi.json`

- Then, requests for any of the following paths will be handled by sending back the contents of the corresponding file:
  - `/openapi.yaml`
  - `/openapi.json`

- Thus, after adding the above `express.static()` middleware specification, we can simply put our OpenAPI specification in a file named `public/openapi.yaml` to have our server serve it upon request.

Complete OpenAPI Specification

- You can find the complete OpenAPI specification we wrote above at the following location:

  https://gist.github.com/robwhess/3e7378b1f2049af51c2dffc65935fb2b