CS 4530: Fundamentals of Software Engineering

Module 10.3 Building REST APIs

Adeel Bhutta, Mitch Wand

Khoury College of Computer Sciences

Learning Goals for this Lesson

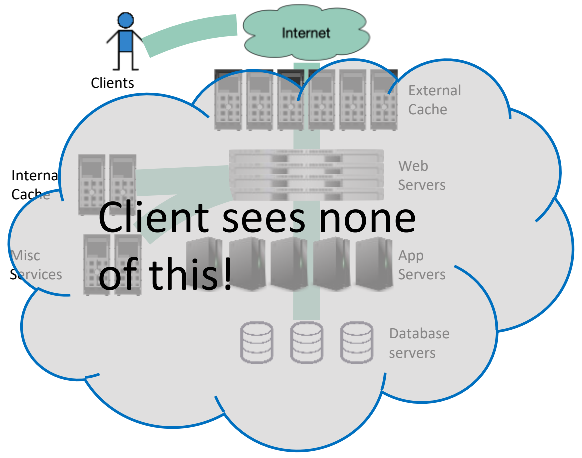
- At the end of this lesson you should be able to
  - Explain the basic principles of the REST methodology
  - Construct a simple REST server using TSOA

REST: Representational State Transfer

- A design principle for http requests
- Commonly used for APIs

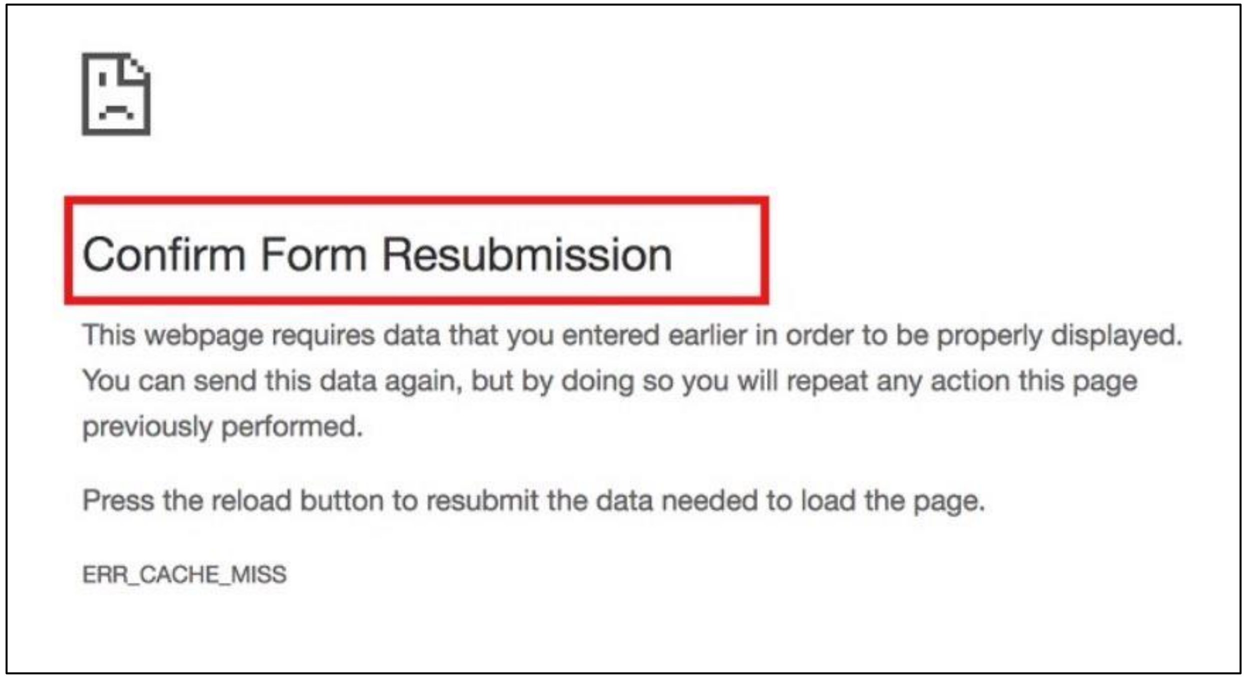
REST Principles

- Single Server - As far as the client knows, there's just one
- Stateless - Each request contains enough information that a different server could process it (if there were multiple…)
- Uniform Cacheability - Each request is identified as cacheable or not.
- Uniform Interface - Standard way to specify interface



“Not cacheable” means that it must be executed exactly once per user request.

- For example, POST is typically not cacheable



Confirm Form Resubmission

This webpage requires data that you entered earlier in order to be properly displayed. You can send this data again, but by doing so you will repeat any action this page previously performed.

Press the reload button to resubmit the data needed to load the page.

ERR_CACHE_MISS

Uniform Interface: URIs are nouns

- In a RESTful system, the server is visualized as a store of named resources (nouns), each of which has some data associated with it.
- A URI is a name for such a resource.

Examples

- Examples:
  - `/cities/losangeles`
  - `/transcripts/00345/graduate` (student 00345 has several transcripts in the system; this is the graduate one)
- Anti-examples:
  - `/getCity/losangeles`
  - `/getCitybyID/50654`
  - `/Cities.php?id=50654`

We prefer plural nouns for toplevel resources, as you see here.

Useful heuristic: if you were keeping this data in a bunch of files, what would the directory structure look like?
But you don't have to actually keep the data in that way.

Path parameters specify portions of the path to the resource

For example, your REST protocol might allow a path like

```
/transcripts/00345/graduate
```

In a REST protocol, this API might be described as

```
/transcripts/:studentid/graduate
```

`:studentid` is a path parameter, which is replaced by the value of the parameter

Query parameters allow named parameters

Example:

`/transcripts/graduate?lastname=covey&firstname=avery`

These are typically used to specify more flexible queries, or to embed information about the sender's state, eg

https://calendar.google.com/calendar/u/0/r/month/2023/2/1?tab=mc&pli=1

This URI combines path parameters for the month and date, and query parameters for the format (`tab` and `pli`).

You can also put parameters in the body.

- You can put additional parameters or information in the body, using any coding that you like. (We’ll usually use JSON)
- You can also put parameters in the headers.
- TSOA gives tools for extracting all of these parameters

Uniform Interface: Verbs are represented as http methods

- In REST, there are exactly four things you can do with a resource
- POST: requests that the server create a resource with a given value.
- GET: requests that the server respond with a representation of the resource
- PUT: requests that the server replace the value of the resource by the given value
- DELETE: requests that the server delete the resource

Example interface #1: a todo-list manager

- Resource: /todos
  - GET /todos - get list all of my todo items
  - POST /todos - create a new todo item (data in body; returns ID number of the new item)
- Resource: /todos/:todoItemID
  - :todoItemID is a path parameter
  - GET /todos/:todoItemID - fetch a single item by id
  - PUT /todos/:todoItemID - update a single item (new data in body)
  - DELETE /todos/:todoItemID - delete a single item

Example interface #2: the transcript database

```
POST /transcripts
 -- adds a new student to the database,
 -- returns an ID for this student.
 -- requires a body parameter 'name', url-encoded (eg name=avery)
 -- Multiple students may have the same name.
GET  /transcripts/:ID
 -- returns transcript for student with given ID.  Fails if no such student
DELETE /transcripts/:ID
 -- deletes transcript for student with the given ID, fails if no such student
POST /transcripts/:studentID/:courseNumber
 -- adds an entry in this student's transcript with given name and course.
 -- Requires a body parameter 'grade'.
 -- Fails if there is already an entry for this course in the student's transcript
GET  /transcripts/:studentID/:courseNumber
 -- returns the student's grade in the specified course.
 -- Fails if student or course is missing.
GET  /studentids?name=string
 -- returns list of IDs for student with the given name
```

Remember the heuristic: if you were keeping this data in a bunch of files, what would the directory structure look like?

Didn't seem to fit the model, sorry

It would be better to have a machine-readable specification

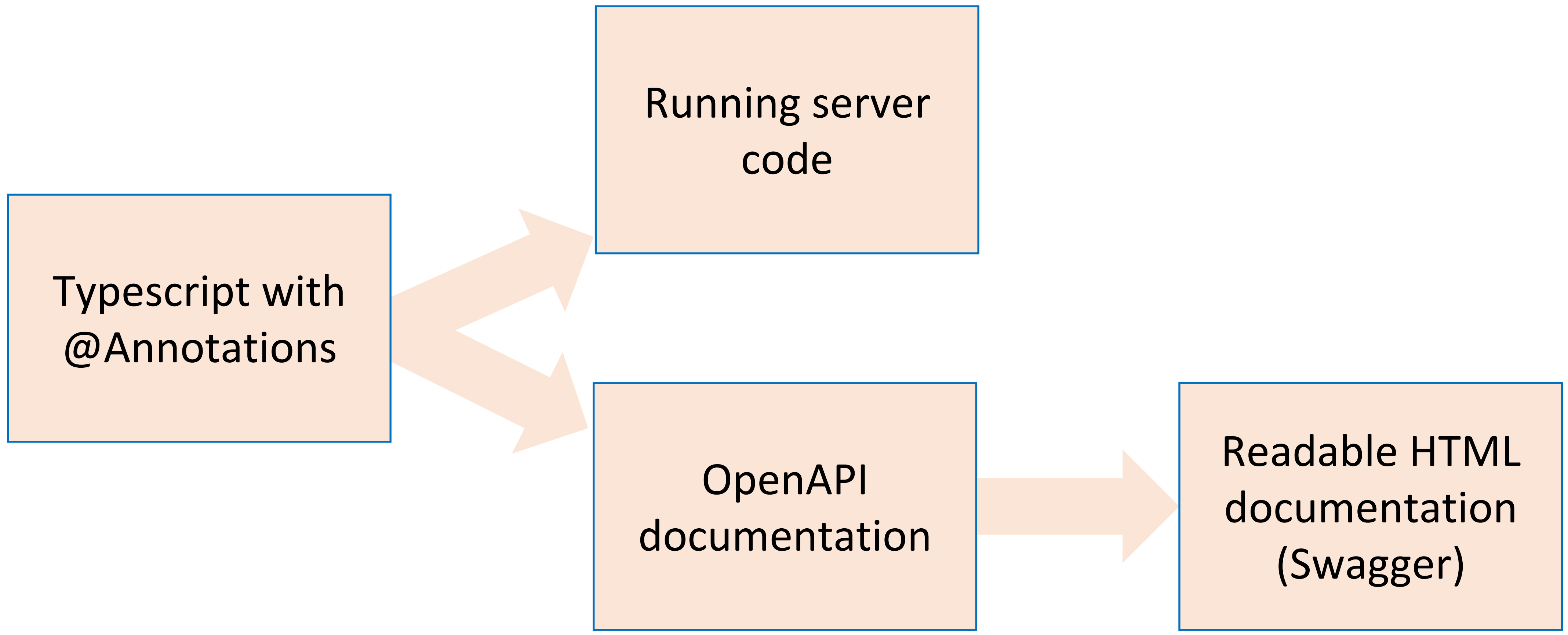
- The specification of the transcript API on the last slide is RESTful, but is not machine-readable
- A machine-readable specification is useful for:
  - Automatically generating client and server boilerplate, documentation, examples
  - Tracking how an API evolves over time
  - Ensuring that there are no misunderstandings

OpenAPI is a machine-readable specification language for REST

- Written in YAML
- Not really convenient for human use
- Better: use a tool!

```yaml
/towns/{townID}/viewingArea:
post:
  operationId: CreateViewingArea
responses:
  '204':
description: No content
'400':
description: Invalid values specified
content:
  application/json:
schema:
  $ref: '#/components/schemas/InvalidParametersError'
description: Creates a viewing area in a given town
tags:
  - towns
security: []
parameters:
  - description: ID of the town in which to create the new viewing area
in: path
name: townID
required: true
schema:
  type: string
  - description: |-
  session token of the player making the request, must
match the session token returned when the player joined the town
in: header
name: X-Session-Token
required: true
schema:
  type: string
requestBody:
  description: The new viewing area to create
required: true
content:
  application/json:
schema:
  $ref: '#/components/schemas/ViewingArea'
description: The new viewing area to create
```

TSOA uses TS annotations to generate all the needed pieces


Typescript with @Annotations
Running server code
OpenAPI documentation
Readable HTML documentation (Swagger)

Sample annotated typescript (1)

This class defines methods that can be invoked on the base route /towns

```
@Route('towns')
export class TownsController exter


/**
 * Creates a viewing area in a given town
 *
 * @param townID ID of the town in whi
 * @param sessionToken session token o
 *        match the session token retu
 * @param requestBody The new viewing
 *
 * @throws InvalidParametersError if t
 *          viewing area could not be
 */
@Post('{townID}/viewingArea')
@Response<InvalidParametersError>(400, 'Invalid values specified')
public async createViewingArea(
  @Path() townID: string,
  @Header('X-Session-Token') sessionToken: string,
  @Body() requestBody: ViewingArea,
){ /** method body goes here */ }
```

This method can be invoked by making a POST request to /towns/{townID}/viewingArea - where /towns was the base route for the class. {townID} is a path parameter

In the event of an InvalidParametersError, the HTTP response will have the error status code “400”

Sample annotated typescript (2)

```
@Route('towns')
export class TownsController exter


/**
 * Creates a viewing area in a given town
 *
 * @param townID ID of the town in whi
 * @param sessionToken session token c
 *        match the session token retu
 * @param requestBody The new viewing
 *
 * @throws InvalidParametersError if t
 *          viewing area could not be
 */
@Post('{townID}/viewingArea')
@Response<InvalidParametersError>(400, 'Invalid values specified')
public async createViewingArea(
  @Path() townID: string,
  @Header('X-Session-Token') sessionToken: string,
  @Body() requestBody: ViewingArea,
){ /** method body goes here */ }
```

This class defines methods that can be invoked on the base route /towns

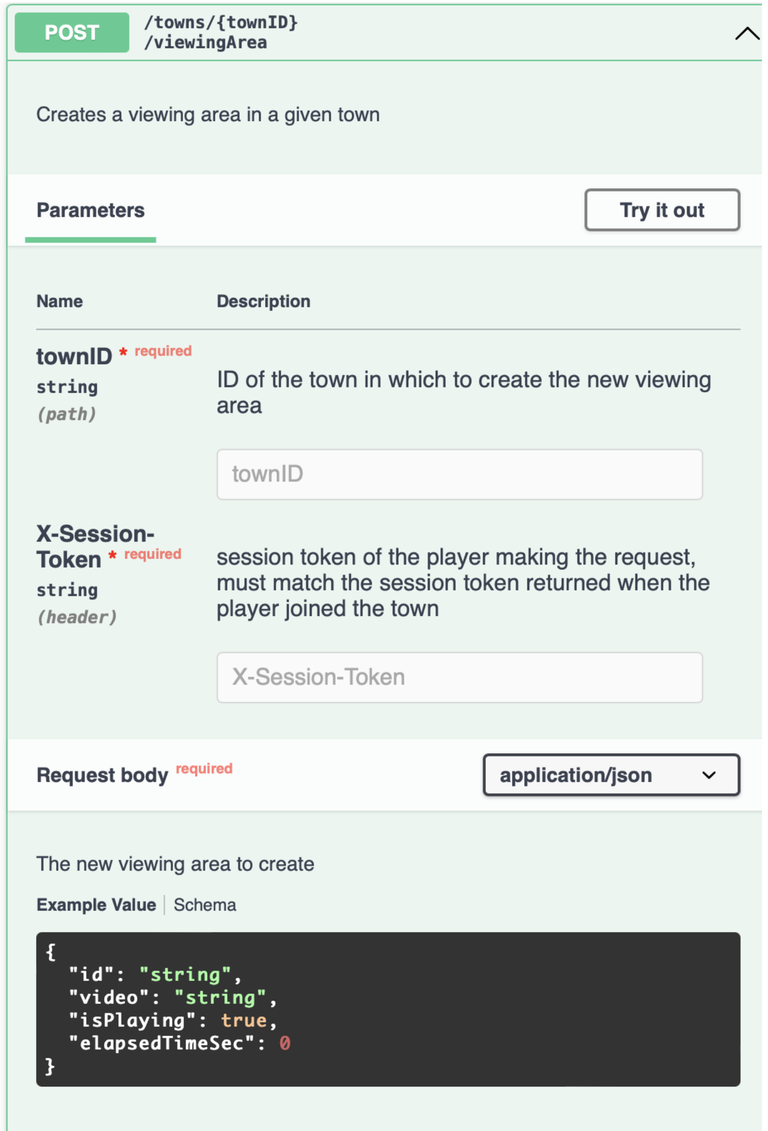
This method can be invoked by making a POST request to /towns/{townID}/viewingArea - where /towns was the base route for th {townID} is a path paramet

The townID parameter to the method will come from the corresponding Path parameter of the URI.

The “sessionToken” parameter will come from an HTTP header called “X-Session-Token”

The requestBody parameter will come from the body of the HTTP request

Sample generated HTML ("Swagger")



/towns/{townID}/viewingArea

Creates a viewing area in a given town

Name	Description
townID * required `string` *(path)*	ID of the town in which to create the new viewing area townID
X-Session-Token * required `string` *(header)*	session token of the player making the request, must match the session token returned when the player joined the town X-Session-Token

Request body required

The new viewing area to create

Example Value | Schema

```
{
  "id": "string",
  "video": "string",
  "isPlaying": true,
  "elapsedTimeSec": 0
}
```

Not everything can be generated ☹

- What if your API method ends with an error, like

```
throw new InvalidParametersError('Some message’)
```

- We need to transmit this information back the requester.

- We’ll need a little custom code to do this– the TSOA language doesn’t do this automatically (IIUC)

Converting JavaScript Errors to HTTP Errors

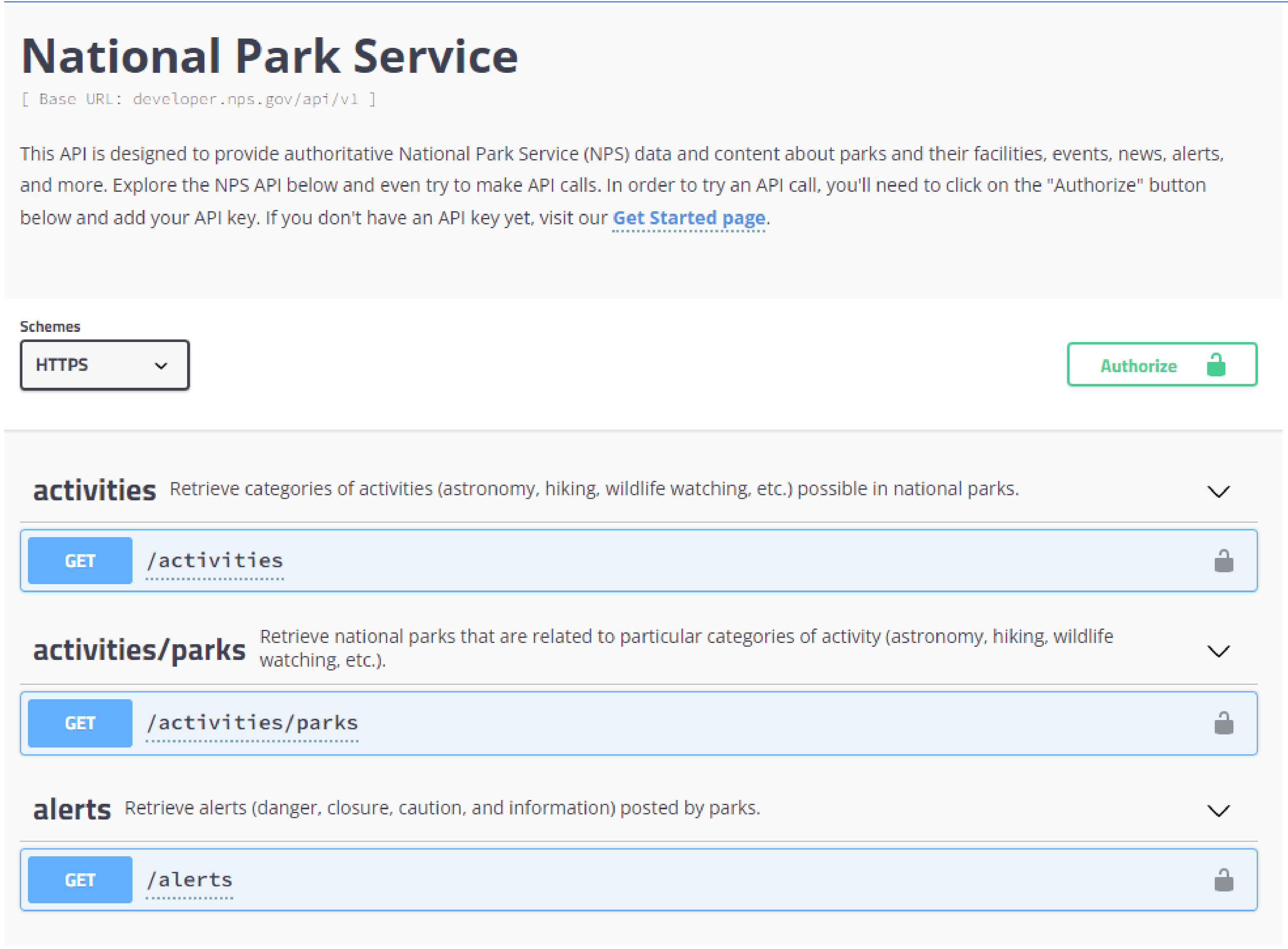
server.ts

- Under the hood, we use the popular express web server for NodeJS
- Express uses an internal pipeline architecture for processing requests
- So we wrote a custom pipeline stage to take care of this.
- This pipeline stage runs after the controller, inspects any error that might be thrown, and returns an HTTP error of 400, 422 or 500, depending on which kind of error you threw.
- Unlikely you will ever have to do this.

```
app.use(
  (
    err: unknown, _req: Express.Request, res: Express.Response,
    next: Express.NextFunction,
  ): Express.Response | void => {
    if (err instanceof ValidateError) {
      return res.status(422).json({
        message: 'Validation Failed',
        details: err?.fields,
      });
    }
    if(err instanceof InvalidParametersError){
      return res.status(400).json({
        message: 'Invalid parameters',
        details: err?.message
      })
    }
    if (err instanceof Error) {
      console.trace(err);
      return res.status(500).json({
        message: 'Internal Server Error',
      });
    }

    return next();
  },
)
```

Swagger in the wild



National Park Service

[Base URL: developer.nps.gov/api/v1]

This API is designed to provide authoritative National Park Service (NPS) data and content about parks and their facilities, events, news, alerts, and more. Explore the NPS API below and even try to make API calls. In order to try an API call, you'll need to click on the "Authorize" button below and add your API key. If you don't have an API key yet, visit our Get Started page.

Schemes: HTTPS

activities Retrieve categories of activities (astronomy, hiking, wildlife watching, etc.) possible in national parks.

GET /activities

activities/parks Retrieve national parks that are related to particular categories of activity (astronomy, hiking, wildlife watching, etc.).

GET /activities/parks

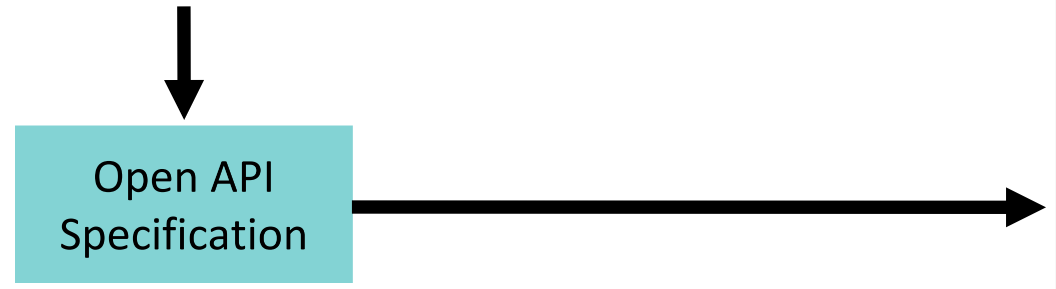
alerts Retrieve alerts (danger, closure, caution, and information) posted by parks.

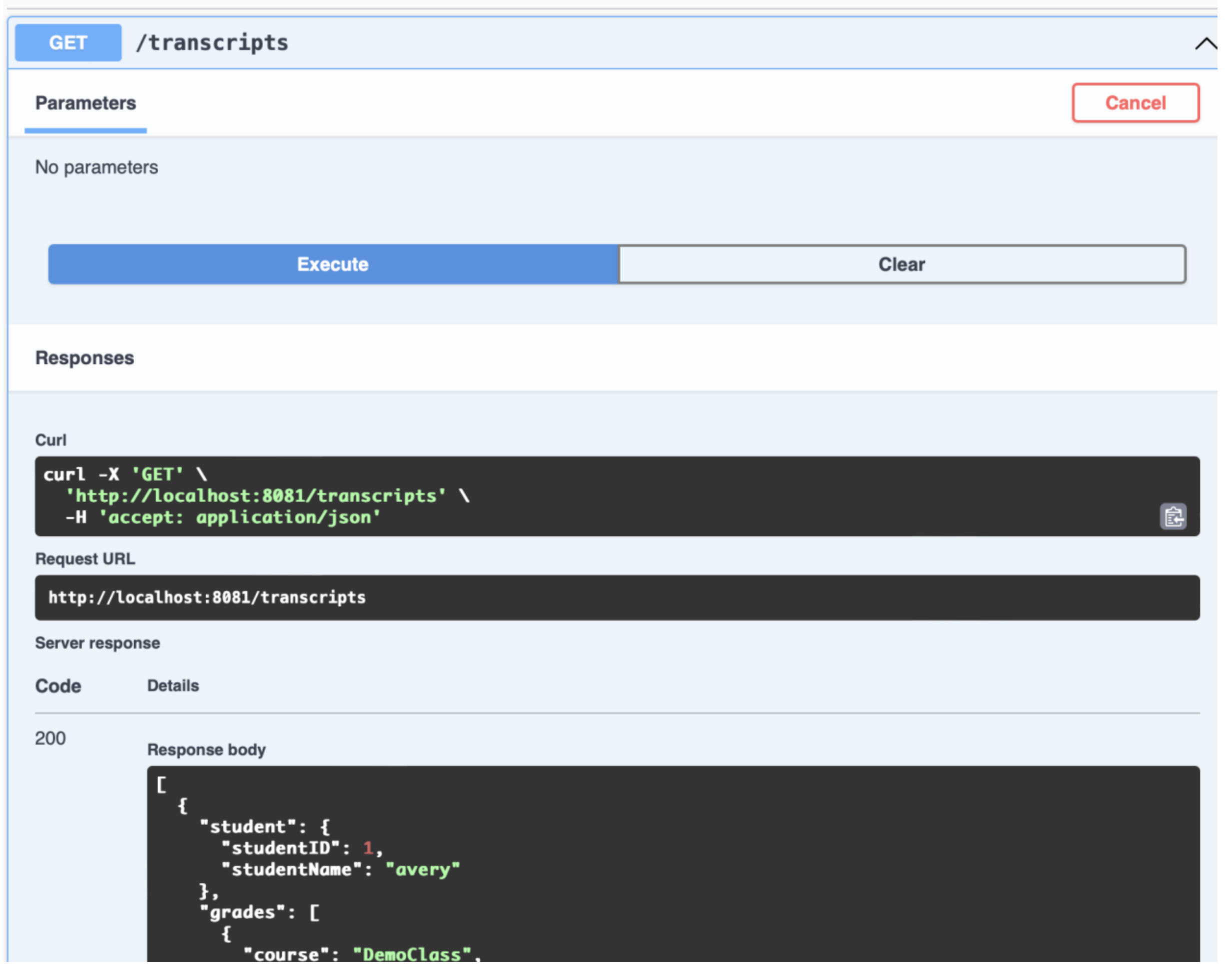
GET /alerts

Activity: Build the Transcript REST API

```
@Route('transcripts')
export class TranscriptsController extends
Controller {

    @Get()
    public getAll() {
        return db.getAll();
    }
```





/transcripts

No parameters

Responses

Curl

```
curl -X 'GET' \
  'http://localhost:8081/transcripts' \
  -H 'accept: application/json'
```

Request URL

```
http://localhost:8081/transcripts
```

Server response

Code	Details
200	Response body

```
[
  {
    "student": {
      "studentID": 1,
      "studentName": "avery"
    },
    "grades": [
      {
        "course": "DemoClass",
```