Please ensure that you successfully complete Module 8 Example before starting this quiz.

Understanding the Basics

[**OSI Model**] The Open Systems Interconnection (OSI) model defines the interconnections and interactions among different levels of network protocols. The OSI model consists of seven layers, as shown in Table 1. The fundamental component of layer 7, also known as the application layer, is HTTP. This protocol operates on a request-response model, wherein a client generates requests to start an HTTP dialogue, and a server returns responses to the client. A client may manifest as a web browser, a frontend platform, or a web app. HTTP facilitates the transfer of hypermedia, encompassing various forms of multimedia content, including but not limited to images, videos, text, and audio. HTTP was extensively utilized throughout this course, namely in the development of major components of web-based apps.

Table 1: OSI Model

#	Layer	Description
7	Application	Network services to apps (e.g. HTTP)
6	Presentation	Data Representation
5	Session	Synch & Send to ports
4	Transport	End-to-End Connection, Flow control
3	Network	Packets & Logical Addressing
2	Data Link	Frames & Physical Addressing
1	Physical	Physical Structure (e.g., media, signal)

[**Knowledge Acquired up to this Point**] In Modules 1-4, we learned fundamental concepts about HTML, CSS and JavaScript. In Modules 5 and 6, we acquired knowledge regarding the principles of design and architecture pertaining to using HTTP, a vital protocol for the OSI's model application layer, for building RESTful web services. Additionally, we explored the utilization of CRUD operations, which can effectively be correlated with methods such as POST, GET, PUT, and DELETE. In addition, we acquired knowledge regarding different HTTP status codes and their usefulness in generating HTTP responses when processing of HTTP requests. In Modules 7 and 8, the course covered the relationship between resources and identifiers, as well as the prevalent media types such as 'application/json'. In addition, we acquired knowledge regarding HTTP headers and the process of developing a web service API that conforms to the principles of REST architecture.

Table 2: Topics Covered in TCSS 460

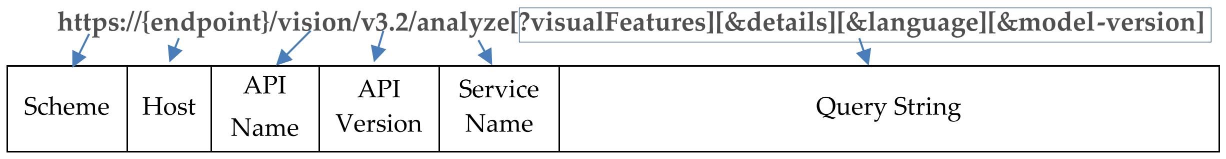
Module #	Topics
1	HTML and CSS
2, 3, 4	JavaScript, jQuery and React
5	HTTP, REST, Node.js
6	REST APIs & Data Serialization
7	REST APIs & Server Programming
8	Web Service API Styles

[**HTTP Parameterization**] One of the fundamental principles of the REST architectural style, as discussed in Lecture 13, is the utilization of unique Uniform Resource Identifiers (URIs) to identify resources. Additionally, REST enforces a uniform interface that supports a limited number of operations, such as GET, POST, PUT, and DELETE. In Lecture 18, we also discussed that <u>not all resource APIs can be classified as RESTful</u>. The method of passing parameters through the endpoint is very prevalent when submitting a HTTP request in REST APIs.

Generally, there are three main types of parameters including: (a) header parameters, (b) path parameters, and (c) query string parameters. Header parameters are components of the request and response headers, whereas path parameters are integral parts of the endpoint structure itself. Additionally, query string parameters are located after the '?' symbol within a URI. After the question mark, parameters and their corresponding values are shown and delimited by an ampersand (&). The relative order of query string parameters does not impact the functionality of the query.

[**Illustrative Example**] In the example provided in Module 8, we utilized the Azure Computer Vision API, specifically focusing on the Analyze Image web service. This service was used to classify the content of images, provide descriptions for images, and identify faces within images, among other visual features.

The endpoint of the Azure Computer Vision API utilizes the query string parameter method, as demonstrated in the provided request URL.


https://{endpoint}/vision/v3.2/analyze[?visualFeatures][&details][&language][&model-version]

[**RESTful Design**] In contrast to query strings, path parameters are deemed essential and are integral to the backend service design, hence constituting a fundamental part of the resource identifiers. An illustration can be provided by considering the endpoint request URL that includes the parameters {**courseId**} and {**sectionId**}, which are utilized as path parameters. The order in which the path parameters are arranged has significant impact on the execution of the functionality. This is because the resources progressively become more specialized as additional identifier levels are incorporated within the URL.

https://{endpoint}/service/myresource/course/{courseId}/section/{sectionId}

The utilization of the path parameter, in accordance with the principles of RESTful design, facilitates the assignment of distinct IDs to resources, thereby aligning with one of the fundamental attributes of REST architecture. For instance, the URL stated above can be associated with distinct resources, therefore establishing a logical framework for constructing responses. The HTTP GET method is assumed for the URLs that follow in the example below.

/service/course	Retrieve course-related information for all available courses
/service/course/{courseId}	Retrieve course-related information for a specific courseId
/service/course/{courseId}/section	Retrieve all section-related information for a specific courseId
/service/course/{courseId}/section/{sectionId}	Retrieve specific section-related information for a specific course
/service/course/{courseId}/section/{sectionId}/student	Retrieve all student information for a specific section of a specific course
/service/course/{courseId}/section/{sectionId}/student/{studentid}	Retrieve specific student-related information for a specific section of a specific course

The REST architecture enables one to select the association of HTTP methods, including POST, GET, PUT, and DELETE, with any URL. In summary, when it comes to RESTful design, it is recommended to avoid utilizing the subsequent format or approach:

GET /service/get_section?sectionId=24564
POST /service/delete_section → pass sectionId=24564 in the HTTP body

In the context of RESTful design, it is recommended to perform the following measures:

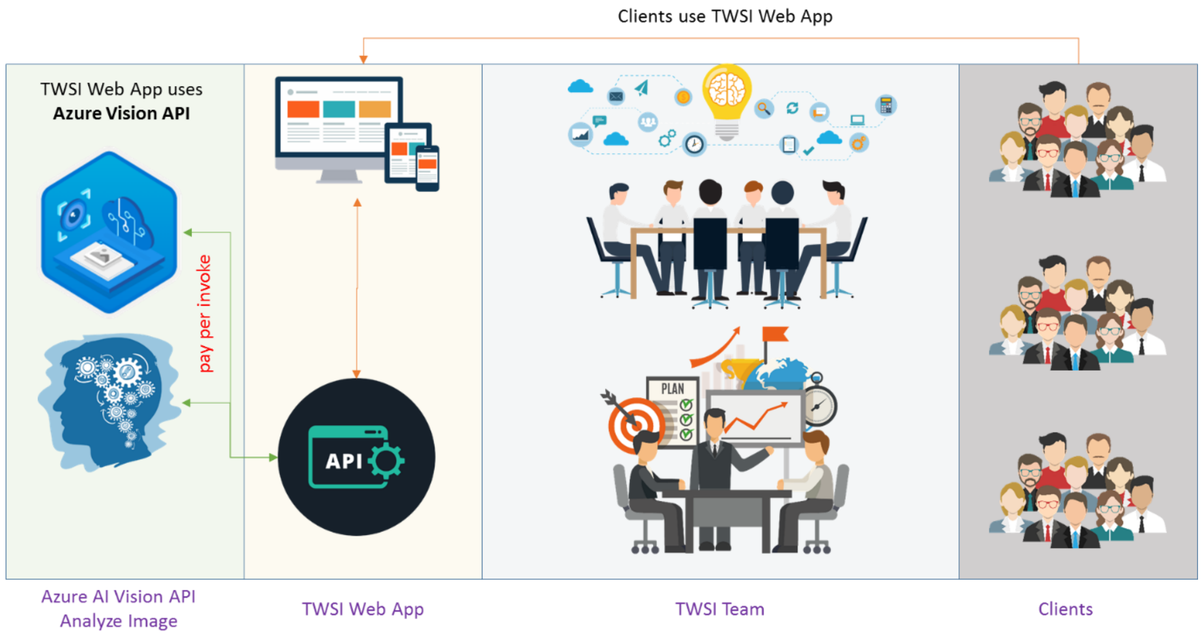
GET /service/sectionId/1
DELETE /service/sectionId/1

Task

[Synopsis] TCSS 460 Web Services Inc. (TWSI), a startup company specializing in offering consultation services pertaining to Artificial Intelligence (AI) applications such as image annotation, object recognition in images, and image classification, has recently experienced a significant expansion in its client base. To this extent, TWSI seeks to develop a web application composed of client and server programming which will enable clients to gain knowledge about the advantages and possible drawbacks associated with the utilization of computer vision and image processing. TWSI intends to strategically rebrand Azure's Computer Vision API, specifically the Analyze Image service, as a reseller marketing tactic in order to cultivate a distinct and novel perception among its clients. Figure 1 presents an overview of TWSI's business model.

Therefore, TWSI aims to develop a web app that integrates Azure's Analyze Image service into its own platform. This application will be accessible to clients through a backend REST web service API that TWSI plans to develop, and a frontend user interface will be provided to allow TWSI clients to explore the services delivered through the TWSI backend web API. The clients of TWSI are clueless of the utilization of the Azure Computer Vision API by TWSI, and TWSI incurs costs to Microsoft for each instance of service invocation performed by a client. In such cases, the backend service API of the web app acts as a wrapper for the features and functionality provided by Azure Computer Vision. This allows TWSI to offer computer vision services to clients under a distinct and unique brand that matches its identity offering computer vision consulting services driven by Azure's Computer Vision API. As a design requirement, TWSI plans to use **Node.js** to develop the backend API and **JavaScript** technologies to develop the frontend (jQuery or React).

Figure 1: TWSI Web App Business Model

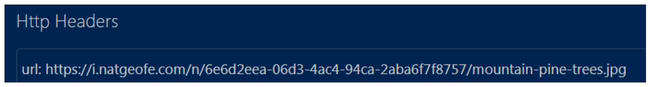


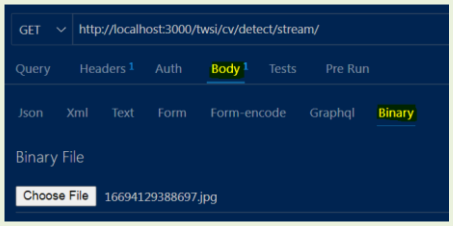
TWSI has employed the expertise of a software designer who has identified a collection of design prerequisites for developing the client frontend, and server backend web service API implementation. The designer seeks to adhere to the REST architectural approach when developing the TWSI web service API. The REST Web Service API offered by TWSI aims to deliver image analysis services to customers in a RESTful manner. These services are essentially the same as those provided by the Azure Computer Vision API. However, Azure's Computer Vision API does not fully adhere to the REST architecture. Hence, the REST Web Service API offered by TWSI acts as a **service wrapper** or **façade** that defines the characteristics of a service layer and establishes a boundary with a predefined set of operations derived from Azure's Computer Vision API (specifically the Analyze Image service). The frontend is responsible for enabling the interaction between the clients of the TWSI and the web API. TWSI has highlighted the subsequent design requirements in light of the aforementioned context. You are required to develop the following items for this activity.

A) Service Wrapper: Backend REST API Design Requirements → folder 'backend'

You are required to develop a REST web service API for TWSI that supports the multiple features offered by the Azure Computer Vision API (Analyze Image service). A sample list of URLs and corresponding description is provided below as a guideline or reference. A description for each URL is provided along with the intended feature that is to be mapped to that of Azure's Computer Vision API (Analyze Image). For all HTTP requests to Azure's Computer Vision API, the 'Content-Type' and 'Ocp-Apim-Subscription-Key' must be included with the respective values. The backend API will utilize Azure's API to execute tasks, specifically application-to-application programming. Subsequently, clients will employ a frontend interface to invoke your service wrapper service using a client interface.

URI	HTTP Method	Purpose
/twsi	GET	Returns a general purpose JSON message containing items like (a) company name, (b) address, (c) email, and (d) telephone number (this can be arbitrarily data).
/twsi/cv	GET	Returns a general purpose JSON message containing a phrase like "TWSI Computer Vision 1.0"
/twsi/cv/detect/	GET	Returns a general purpose JSON message containing a phrase like "TWSI Image Detection"

/twsi/cv/detect/url	POST	Processes an image at a given **URL** string. The URL string must be passed through a **HTTP header attribute** called 'url'.
		Sample HTTP Body:
		`{`
		`  "visualFeatures" : "Categories,Color,Description,Faces,Objects,Tags",`
		`  "details" : "Celebrities,Landmarks",`
		`  "language" : "en",`
		`  "model-version": "latest"`
		`}`
		Sample HTTP Header:
		
		Http Headers
		url: https://i.natgeofe.com/n/6e6d2eea-06d3-4ac4-94ca-2aba6f7f8757/mountain-pine-trees.jpg

/twsi/cv/detect/url/{features}	POST	Processes an image at a given **URL** string. The URL string must be passed through a **HTTP header attribute** called 'url'. **{features}** specifies one or more visual features that are intended to be utilized within the HTTP request. The value of features does not have to include all of the visual features. For example**,** '**/twsi/cv/detect/url/Categories,Color,Tags**' is a valid set of visual features that can be employed and limited to Categories, Color and Tags to be returned. **There are no restrictions on what values to use for details and language.**
/twsi/cv/detect/url/features/{feature}	POST	Processes an image at a given **URL** string. The URL string must be passed through a **HTTP header attribute** called 'url'. {**feature**} specifies a single visual features to be employed within the HTTP request. For example, For example, '**/twsi/cv/detect/url/features/Tags**' represents a HTTP request for using only the Tags visual feature to be employed. **There are no restrictions on what values to use for details and language.**
/twsi/cv/detect/url/details	POST	Processes an image at a given **URL** string. The URL string must be passed through a **HTTP header attribute** called 'url'. **details** specifies ~~both~~ domain-specific details of ~~Celebrities and~~ Landmarks to be employed. For example, '**/twsi/cv/detect/url/details**' represents a HTTP request for using ~~both Celebrities and~~ Landmarks as an option for details. **There are no restrictions on what values to use for features (one, some or all), and language.**
/twsi/cv/detect/url/details/{detail}	POST	Processes an image at a given **URL** string. The URL string must be passed through a **HTTP header attribute** called 'url'. {**detail**} specifies a single domain-specific detail to be employed within the HTTP request. For example, '**/twsi/cv/detect/url/details/landmarks**' represents a HTTP request for using only the Landmarks details option. **There are no restrictions on what values to use for features (one, some or all), and language.** **What is the suggested approach for resolving the Azure's API Limited Access Registration Issue to the Celebrities details option?** If a client sends a request to '**/twsi/cv/detect/url/details/Celebrities**', your API should respond to the client with an appropriate message such as: "In order to mitigate the potential for abuse of face detection services, registration is necessary for this Celebrities details feature. To obtain additional information, please reach out to our sales team at sales@twsi."
/twsi/cv/detect/stream	POST	Processes an image as binary file in the HTTP body using **application/octet-stream**. This URL should be used with clients that have a form element to browse for a local file. Since the binary data will be in the body, pass the values for visualFeatures, details, language and model-version in the HTTP header instead. **There are no restrictions on what values to use for features (one, some or all), and language.** In Thunder Client, you can test for binary file upload in the HTTP body using the Binary option as shown below.  GET http://localhost:3000/twsi/cv/detect/stream/ Query Headers 1 Auth Body 1 Tests Pre Run Json Xml Text Form Form-encode Graphql Binary Binary File Choose File 16694129388697.jpg

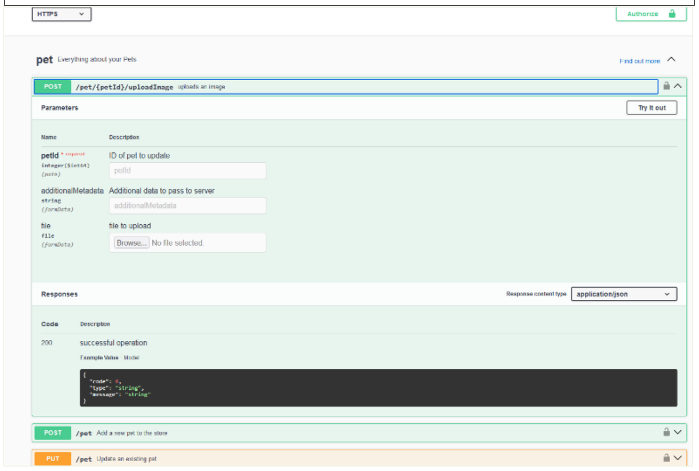
While the aforementioned design requirements serve as an initial guideline, as a web application developer, **you have the flexibility to make appropriate adjustments or modifications regarding data exchange, parameter handling, and route definitions**. The aforementioned serves solely as an illustrative example. You are free to make any necessary or appropriate alterations or modifications to the above design and/or requirements. Further, you may expand on the features listed above as deemed appropriate. However, the level and complexity of the Web Service API should be somewhat similar to the above design guidelines. This frontend should be implemented in a separate project or folder from that of REST API in Part A (e.g., Part A: "backend", Part B: "frontend"), particularly if using React in Node.js.

B) Frontend: Frontend Client for Service Wrapper in Part A → folder 'frontend'

You are required to develop a frontend UI that demonstrates the utilization of the web service API implemented in Part A. The frontend interface should provide clients with the ability to **select or choose via form elements** the several API parameters, including visualFeatures, details, and language. Furthermore, it is imperative for clients to possess the capability to **identify the method by which an image might be specified**, whether it be through a **URL string** or by uploading a **binary image file** using a browse form element. It is essential to give the visualFeatures and details options to the client in the form of **checkbox selections** such that more than one option can be selected. The frontend component needs to have a proper stylesheet that effectively organizes and presents the user interface in a visually aesthetic UI design. The interface design should also be simple and intuitive for clients to use. The frontend must be developed using HTML, CSS and JavaScript. **You are free to use jQuery or React for implementing the frontend interface.** This frontend should be implemented in a separate project or folder from that of REST API in Part A (e.g., Part A: "backend", Part B: "frontend"), particularly if using React in Node.js.

C) API Docs: Using Swagger Specification

Source: Swagger UI: https://petstore.swagger.io/#/pet/uploadFile



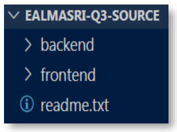
```
{
  "code": 0,
  "type": "string",
  "message": "string"
}
```

During the course of the completion of Assignment 4, a significant amount of effort may have been dedicated to the design and development of HTML-based documentation for the implemented API. Would it not be ideal if there were a way that facilitated the automatic and effective generation of API documentation for our Web Service API? There are a variety of automatic documentation generators specifically designed for web service APIs, with a particular focus on Node and Express. Swagger UI is a widely used tool for generating automated documentation. You are required to create a Swagger-based API documentation that encompasses the necessary routes, parameters, sample requests, and sample answers for the web service API implemented in Part A. Additionally, it is recommended that the API documentation incorporates a functionality that allows users to test the routes in real-time (e.g., via the Try it out feature). The API documentation should encompass a comprehensive overview of all the service capabilities that are supported within the implementation of the REST API. The following links are recommended for further reference.

- Swagger Documentation UI Demo
- YouTube Tutorial: NodeJS Swagger API Documentation Tutorial Using Swagger JSDoc
- YouTube Tutorial: Automatically Generate Swagger Docs With ExpressJS & NodeJS

What to Submit

Submit a **userid-q3.zip** file containing the following elements (userid is your UW userid):



a) **userid-q3-source.zip**: It is unnecessary to include the node-modules in order to conserve disk space on Canvas. Therefore, only the source code files and folders for the Node.js project should be included.
   - Backend REST API must be developed in Node.js and stored in a folder called '**backend**'
   - Frontend interface must be developed in folder of its own called '**frontend**'

b) **readme.txt**: A "readme.txt" file which (a) includes the necessary npm install command for installing node modules throughout the grading process (e.g, "npm i express nodemon"), and (b) any Azure API keys and service endpoint URIs used (e.g., including region) for testing.

c) **userid-q3.mp4** (or webm or equivalent): A video presentation recording, with a duration of 10-12 minutes, demonstrating (a) the design and implementation of the REST API, (b) frontend, and (c) API Docs using Swagger UI.

   a) **REST API:** This section is meant to demonstrate the service capabilities and their implementation from Part A, which leverages the Azure Computer Vision API, by utilizing the **Swagger UI** developed in Part C. Please identify any design modifications that have been made while incorporating functionality from the Azure API into the built REST API. For instance, have there been any alterations to parameter names or other aspects of the design? You can take advantage of the 'Try it out' functionality within Swagger UI to showcase the design and a fully operational REST API.
      - In order to effectively demonstrate the REST API, it is recommended to provide a concise overview of the code and clarify key code sections.

   b) **Frontend**: Utilize the frontend that was developed in Part B to demonstrate the various functionalities of the REST API created in Part A. The frontend demonstration should encompass the diverse range of capabilities that have been implemented. It should provide end users with the ability to utilize HTML forms in order to execute the many functionalities of the API that have been implemented in Part A.

   c) Your presentation should demonstrate the following aspects:
      - A functional implementation of the Azure Computer Vision API (namely the Analyze Image service) operations has been integrated into the REST API developed in Part A. This integration serves to emphasize the diverse range of functionalities provided by the Azure Computer Vision API.
      - A practical utilization of JSON for data sharing and handling,
      - A functional implementation of a frontend user interface in Part B, which facilitates the generation of HTTP requests through a client interface.

   d) It is recommended to establish a test plan before commencing the recording of the presentation, outlining the techniques that will be demonstrated, and thereafter integrating them into the recording. It is strongly recommended to create a PowerPoint slide for integration into one's presentation.
      - Recommended presentation format (slide per each):

○ High-level design of the URL (routes) of the web service API	1 min
○ Frontend design and UI features implemented	1 min
○ REST Web service API design	1 min
▪ discuss service operations	
▪ discuss integration of features from Azure CV API	
○ Demonstrate your REST web service API using frontend	5 mins
○ Demonstrate the Swagger API Docs implemented	1 min
○ Challenges, summary and possible future work	1 min

Rubric and Grading Scheme (35 possible marks)

10 marks	This section is intended for the assessment of the design and implementation of the REST Web Service API from Part A. The evaluation criteria for this item will include aspects such as REST API design, creative thinking, granularity, and complexity.
10 marks	This section is intended for the assessment of the frontend user interface and code implementation from Part B. The evaluation criteria for this item will include the inclusion of graphic elements, use of CSS, and aesthetically appealing user interface elements.
9 marks	This section is intended for the assessment of the Swagger API documentation from Part C generated or developed for the REST Web Service API. The evaluation criteria for this item will include the appropriate design and structure of swagger-based API documentation, try it out features for testing the service capabilities, as well as the visual aesthetic of the swagger doc user interface.
6 marks	This section is intended for the assessment of the overall quality of the recorded presentation, considering elements such as the organization and coherence of the content, the ability to convey a comprehensive understanding of the provided services, and the demonstration of the Azure Computer Vision API (specifically, the Analyze Image) operations through the REST API from Part A.
(-3 marks)	For any missing files.
(+4 marks)	[**Bonus**] A maximum of three marks can be awarded for a successful integration of Twillio SMS notification message generation feature, utilizing SMS, in response to the processing of an image by your API.
(+4 marks)	[**Bonus**] A maximum of three marks can be awarded for the successful integration of the Azure Computer Vision API (3.2) OCR service into the REST API outlined in Part A, as well as the successful demonstration of its utilization through the frontend interface developed in Part B.
(+2 marks)	[**Bonus**] A maximum of two marks can be awarded if React was utilized in the development of the frontend interface.