Cloud Computing and Software Engineering

Assignment #1, 2025

Daniel Yellin

Assignment #1

Due Date: November 27, 2025. Time: 23:59

All the instructions in these slides must be followed **exactly**. If you have any questions, ask them in the Piazza **homework_#1** folder.

Assignment #1

- Build a REST application to manage the inventory of a pet store. The application will:
  - Keep track of all pets currently in the store,
  - Use a 3rd party to retrieve extra information about the pets,
  - Retrieve and store pictures of pets.
- For this assignment, the information stored in the application will not be persistent. Everytime the application runs it will start from scratch.
- The application must run in a Docker container.

Resources

1. /pet-types
2. /pet-types/{id}
3. /pet-types/{id}/pets
4. /pet-types/{id}/pets/{name}
5. /pictures/{file-name}

/pet-types is a collection of all the pet types (poodles, goldfish, siamese cats,...) in the pet store.

/pet-types/{id} is the pet-type with the given id.

/pet-types/{id}/pets is a collection of pets in the store of the given type.

/pet-types/{id}/pets/{name} is the pet with the given name.

/pictures is a collection of file names. Each file is a picture of a pet in the store. Pictures may not be available all pets. The client can only issue requests on /pictures/{name}, not on the collection.

Details on the exact JSON representation of these resources are giving in the following slides.

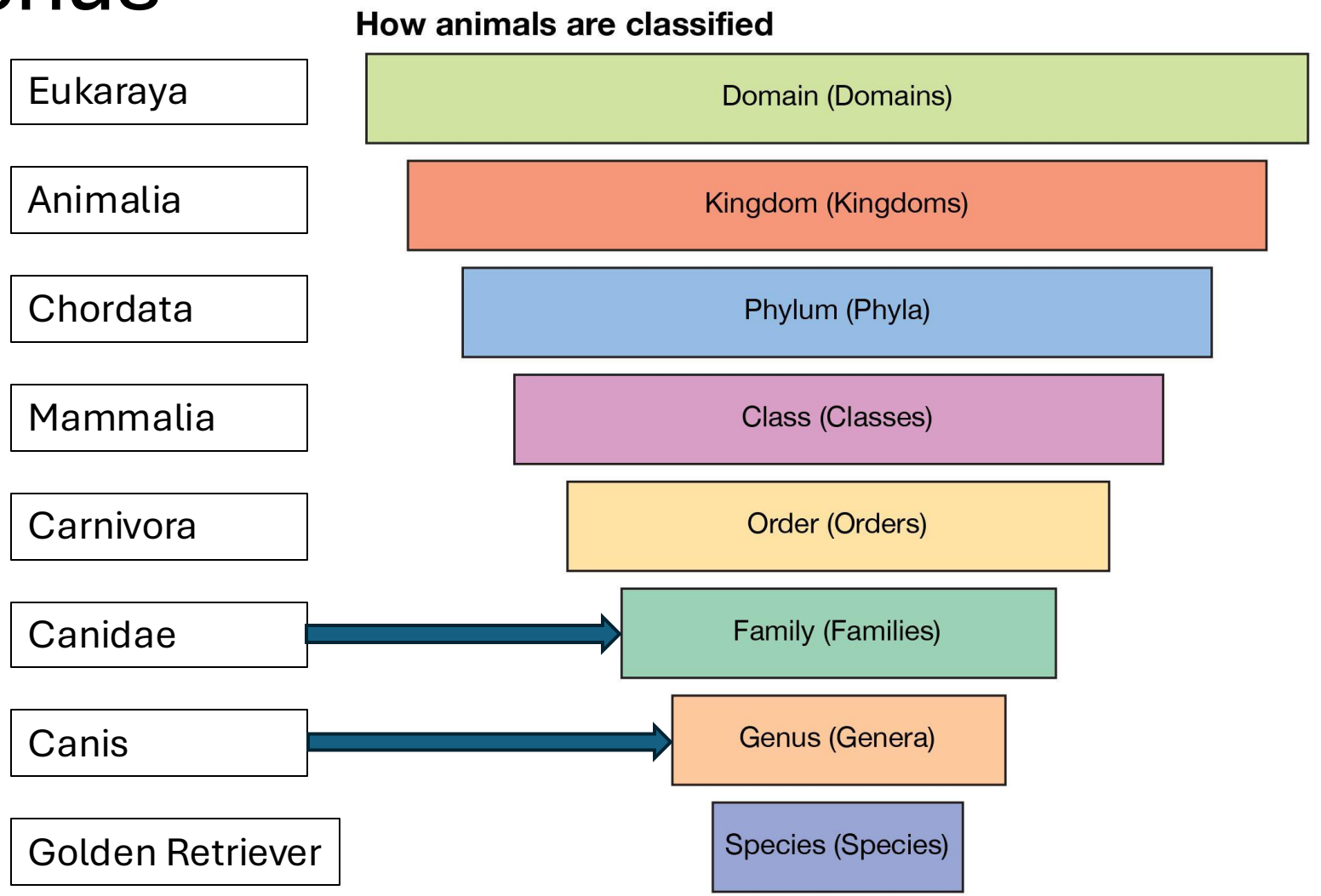
JSON for pet-type

```
{
  “id”: string,
  “type”: string,
  “family”: string,
  “genus”: string,
  “attributes”: array (of string),
  “lifespan”: int,
 “pets”: array (of string)
 // attributes may be empty.
 // lifespan may be null.
}
```

Example pet-type

```
{
  “id”: “2”,
  “type”: “Poodle”,
  “family”: “Canidae”,
  “genus”: “Canis”,
  “attributes”: [“Intelligent”, “alert”,
                 “active”],
   “lifespan”: 16,
  “pets”: [“Tony”, “Lian”, “Jamie”]
}
```

Family and genus



JSON for a pet

```
{
  “name”: string,
  “birthdate”: string,
  “picture”: string // name of file
                    // storing the image
  // birthdate & picture may have
  // the value “NA”
}
```

Example pet

```
{
  “name”: “Jamie”,
  “birthdate”: “24-10-2023”,
  “picture”: “Jamie-poodle.jpg”
}
```

IDs and names

pet-type **ID**s are **unique** strings. The id could be simple string representation of integers (“1”, “2”,…) or could be some other unique string (“VT77G8UMQ”).

If an pet-type is deleted, its id should **not** be reused by a new object.

All **name**s of pets of a given pet-type are **unique** strings giving the name of the pet. You cannot have two pets of same pet-type with the same name.

JSON and picture representations

- All resources are represented by JSON documents except for an image. For JSON:
  - The Content-Type header is set to “/application/json”.
  - The order of fields in a JSON object is not significant.
  - The field names must match exactly field names given in these slides.
  - All string values are case insensitive (“Cat” == “cat”)
  - All dates are of form DD-MM-YYYY; e.g., “06-21-2011”
- When sending an image in response to a client request (GET /pictures/{file-name}):
  - The Content-Type header is set to “image/jpeg” or “image/png” (whichever is the appropriate MIME type for the image).

Requests on the /pet-type resource

- **POST request**: The POST request provides a JSON object payload that contains only the field named 'type'. The server returns the status code 201 and the JSON pet-type object containing all the fields. The 'pets' field will be an empty array. The values of the fields other than 'id' & 'pets' will be obtained from the Ninja-API.
- Possible error status codes returned: 400, 415, 500. If the Ninja API cannot find the pet with the given type, 400 is returned. For any other Ninja API error (not expected) return 500. If the POST request is for a pet-type that already exists, 400 is returned.
- The attributes field is obtained from the field named either **'temperament'** or **'group_behavior'** returned by the Ninja API. If none of these fields are returned by the Ninja API, then the 'attributes' field is empty. If both are listed then just use "temperament". If the 'lifespan' field is not returned by the Ninja API, then its value is the JSON keyword *null*. A later slide describes how to fill in the attributes & lifespan field.

- **GET request**: The server returns a JSON array of pet-types with status code 200.
- No error status codes.

- For this assignment, you need to support query strings of the form "**<field>=<value>**", where field is one of {the 'id', 'type', 'family', 'genus', 'lifespan'}.

  You also need to support a query string of the form "**hasAttribute = <attribute>**".

  Example query strings:
  "family=Canidae"
  "hasAttribute=active"
  "hasAttribute=intelligent"

A later slide describes hasAttribute query strings in more detail.

Requests on /pet-types/{id}

- **GET**: if successful it returns the JSON representation of the pet-type of the given id with status code 200. Possible error status codes returned: 404.

- **DELETE**: deletes the pet-type object of the given id from the collection. If successful, it returns status code 204 with no content (the empty string).
- Possible error status codes returned: 400 and 404.
- If the ‘pets’ field of the pet-type is not empty, then it returns 400 (and does not delete the entry).

- **PUT**: is not allowed on /pet-types/{id}

Requests on /pet-types/{id}/pets

- **POST**: The POST request provides a JSON object payload that contains the ‘name’ field. Optionally it contains ‘birthdate’ and ‘picture-url’ fields.

  If successful it returns the status code 201 and the JSON pet object containing the ‘name’, ‘birthdate’ and ‘picture’ fields. The birthdate field will be “NA” if it was not supplied. The ‘picture’ field will be “NA” if the ‘picture-url’ field was not supplied. Otherwise, the server code will fetch the picture at the given url and store the image in a file. The ‘picture’ field will contain the name of that file.
- Possible error status codes returned: 400, 404 and 415.

- **GET**: If successful, it returns a JSON array of pets of this pet-type with status code 200.
- Possible error status codes returned: 404.
- For this assignment, you need to support query strings of the form “**birthdateGT=<date>**” or “**birthdateLT=<date>**”which returns only those pets with birthdates after or before the given date. <date> is of the form DD-MM-YYYY. It can also be of the form “**birthdateGT=<date>&birthdateLT=<date>**”.

  Example query strings:
  “birthdateGT=27-05-2021”
  “birthdateLT= 05-10-2023”
  “birthdateGT= 27-05-202&birthdateLT= 05-10-2023”

Requests on /pet-types/{id}/pets/{name}

- **GET request**: the server returns the JSON for the pet object with the given name with status code 200.
- Possible error status codes returned: 404.

- **DELETE request**: the server deletes the pet with the given name from the collection and returns status code 204 with no content (the empty string).
- Possible error status codes returned: 404.
- If the pet has a picture then you should also delete the file containing that picture when deleting the pet object.

- **PUT request**: the server updates the pet resource with the the new pet object provided in the payload payload. **The payload must include the name field of the pet object.** The values of the other fields are optional

  The server returns the status code 200 and the JSON pet object containing the 'name', 'birthdate' and 'picture' fields. The 'birthdate' field will be "NA" if it was not supplied. The 'picture' field will be "NA" if the 'picture-url' field was not supplied. Otherwise, the server will fetch the picture at the given url and store the image in a file*. The 'picture' field will contain the name of that file.
- Possible error status codes returned: 400, 404 and 415.

* If the picture url provided is the same as the previous one, then one does not need to fetch the image again.

Requests on /pictures/{file-name}

- **GET request**: is the only allowable request on the /pictures/{file-name} object. The payload is the JSON below. The server returns the image (of Content Type .png or .jpg) with status code 200.

  ```
  {
    “file-name”: string
    // name of file storing the image
  }
  ```

- Possible error status codes returned: 404.

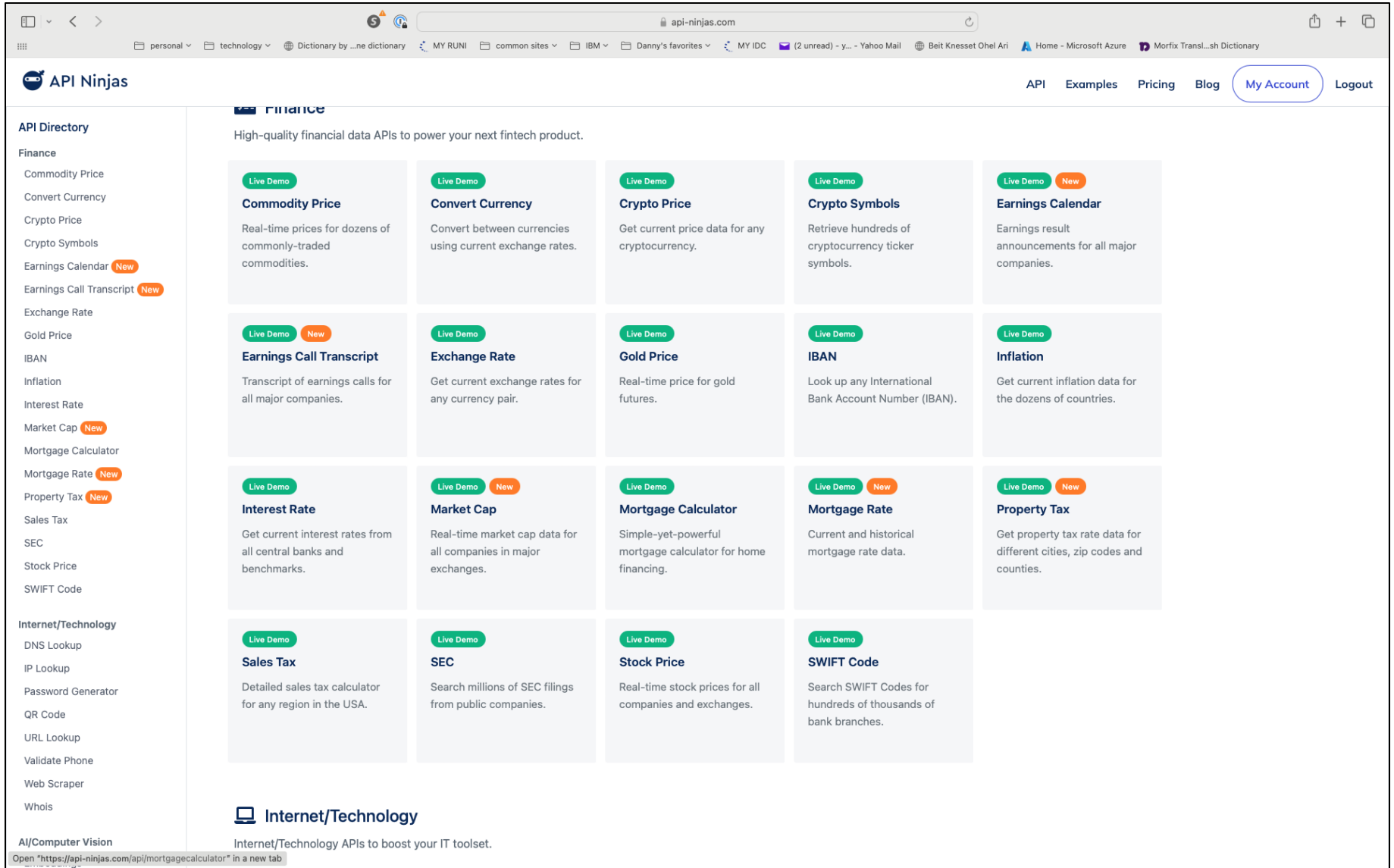
JSON format for error messages

If an exception occurs on a REST request, you return a response code with a JSON object. The following shows the format of the JSON object for each error response that might occur in this assignment.

- {"error": "Malformed data"}, 400
- {"error": "Not found"}, 404
- {"error": "Expected application/json media type"}, 415
- Similar to the above for other media types (such as jpeg)
- {"server error": "API response code " + str(response.status_code)}, 500
  where response.status_code is the exception status code returned by the NINJA API call.
  As mentioned above, if the 'type' of pet is not recognized by the Ninja API, then 400 status code is returned (not 500).

API Ninjas offers a lot of APIs

https://api-ninjas.com/



Animals API

The Animals API provides interesting scientific facts on thousands of different animal species.

/v1/animals GET

`https://api.api-ninjas.com/v1/animals`

Returns up to **10** results matching the input name parameter.

Parameters

`name` required

Common name of animal to search. This parameter supports partial matches (e.g. `fox` will match `gray fox` and `red fox`).

Headers

`X-Api-Key` required

API Key associated with your account.

Sample Request Live Demo!

name goldfish

`https://api.api-ninjas.com/v1/animals?name=goldfish`

Headers

X-Api-Key cMf1OoKKukDu21C4xel9L2i3EP9cYMbUesYfmFYg

Send Request

Sample Response

JSON


```
[
  {
    "name": "Goldfish",
    "taxonomy": {
      "kingdom": "Animalia",
      "phylum": "Chordata",
      "class": "Actinopterygii",
      "order": "Cypriniformes",
      "family": "Cyprinidae",
      "genus": "Carassius",
      "scientific_name": "Carassius auratus"
    },
    "locations": [
      "Asia"
    ],
    "characteristics": {
      "prey": "Insects, small crustaceans",
      "group_behavior": "Gregarious",
      "estimated_population_size": "Many millions",
      "biggest_threat": "None",
      "most_distinctive_feature": "The orange-gold color of domestic fish",
      "water_type": "Fresh",
      "optimum_ph_level": "6.0 to 8.0",
      "habitat": "Cold freshwater rivers, canals, ditches with lots of vegetation",
      "predators": "Waterbirds, turtles, larger fish",
      "diet": "Omnivore",
      "type": "Fish",
      "common_name": "Fish",
      "number_of_species": "1",
      "color": "BlackWhiteOrangeOliveGolden",
      "skin_type": "Scales",
      "top_speed": "0.86 mph",
      "lifespan": "Up to 41 years",
      "weight": "Up to 5 pounds",
      "length": "1 to 2 inches to over 16 inches"
```


For this assignment, use the Animals API

https://api-ninjas.com/api/animals

Family and genus fields

group_behavior field

lifespan field

In this example, the API returns ‘temperament’ (not ‘group_behavior’ field). You would use this field for the JSON ‘attributes’ field of the pet-type object.

If the API returns both ‘temperament’ and ‘group_behavior’ fields, you should use only the ‘temperament’ field.

JSON


```
[
  {
    "name": "Beagle",
    "taxonomy": {
      "kingdom": "Animalia",
      "phylum": "Chordata",
      "class": "Mammalia",
      "order": "Carnivora",
      "family": "Canidae",
      "genus": "Canis",
      "scientific_name": "Canis lupus"
    },
    "locations": [
      "Europe"
    ],
    "characteristics": {
      "temperament": "Gentle and intelligent but stubborn",
      "training": "Should be trained in obedience from an early age d
      "diet": "Omnivore",
      "average_litter_size": "7",
      "common_name": "Beagle Dog",
      "slogan": "Have become popular family pets!",
      "group": "Hound",
      "color": "BrownRedBlackWhiteTan",
      "skin_type": "Hair"
    }
  },
```

Ninja Animals API

- Look carefully at the Ninja API and try some examples. You need to take the data returned from Ninja and put it into the JSON we defined. You may need to manipulate the returned data a bit to get it into the right format.
  - Example: Ninja API *might* return the field **“lifespan”: “up to 41 years”**. However the REST JSON for pet-type must be an integer and would have the form: **“lifespan”: 41**.
  - Example 2: Ninja API *might* return the field **“lifespan”: ”from 2 to 5 years**”. In this case the REST JSON for pet-type would have the form **“lifespan”: 2**. More generally, always use the **lowest** number in the returned text.

- When requesting information on an animal in the Ninja Animals API, you provide the animal name. The Ninja API returns an array of animals that are relevant. **You must pick the entry with the exact name in the REST request**. An example is given on the next slide.

In this example the request is for “Golden retriever”.

- The API returned 2 entries, one for “English Cream Golden Retriever” and one for “Golden Retriever”.
- You need to pick the entry for “Golden Retriever”, as that matches exactly the request (ignoring capitalization).
- Note that neither “temperament” nor “group_behavior” is supplied for the Golden Retriever entry.

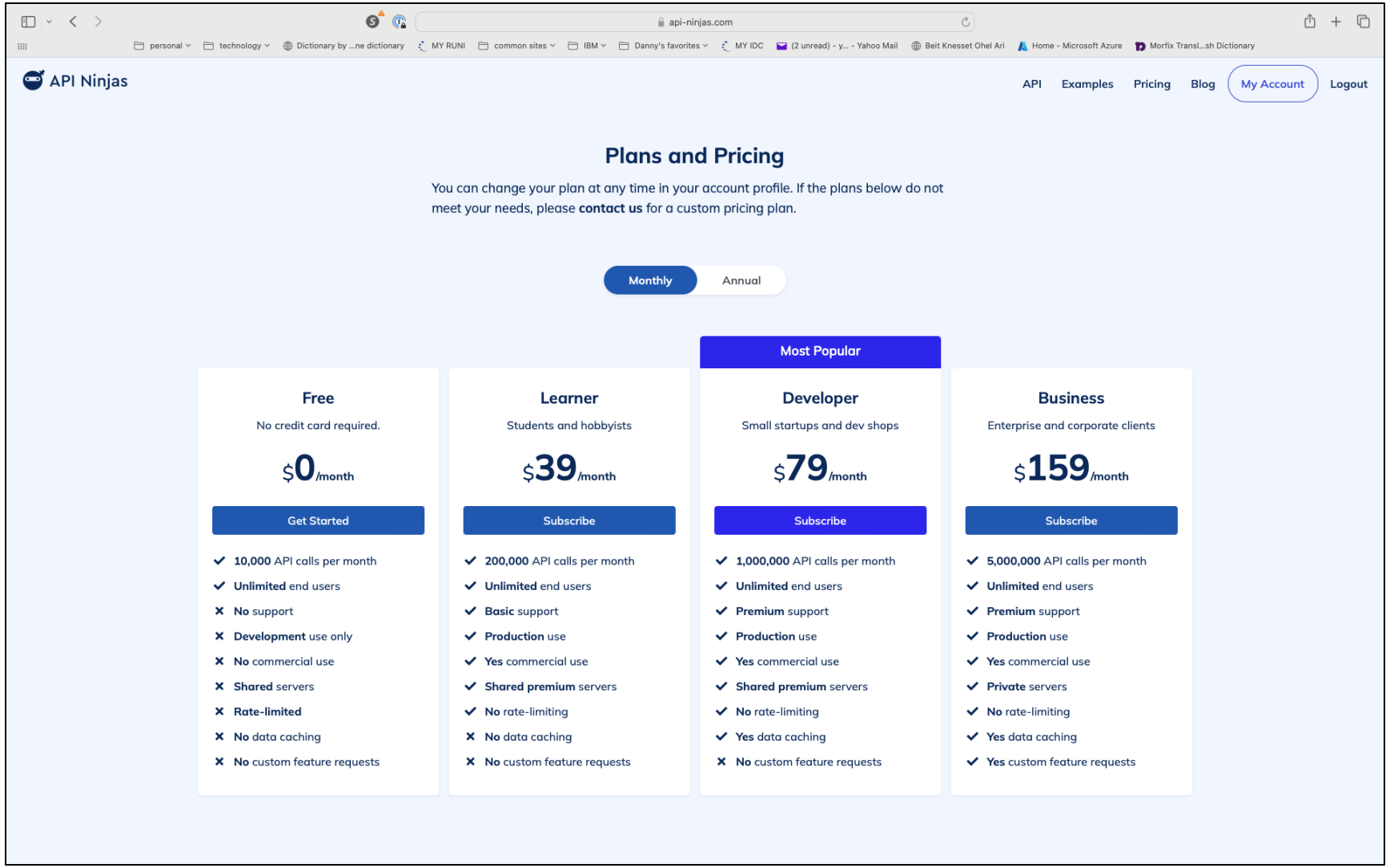
Lifespan would be 10.

Lifespan would be 12.

The ‘attributes’ field and query strings

- The “temperament” and “group_behavior” values in the Ninja API are strings. But the “attributes” field of a pet-type object is an array of strings.
- To convert the “temperament” (or “group_behavior”) string to an array of strings, you simply make each word in the “temperament” sentence into its own string. The array is the list of words in the sentence.
- Example. “temperament” is “Very active and intelligent”.
- “attributes” would be set to [“Very”, “active”, “and”, “intelligent”].
- Ignore any punctuation (commas, periods, …)
- If the query string is “hasAttribute=active” then an entry satisfies the query string if any of the strings in the “attributes” field is “active”.

10,000 calls free per month



https://api-ninjas.com/pricing

- 10,000 API calls per month are free – that is far more than enough for this project.
- You need to go to the site and register for this service and you will get your API-Ninjas API key.

- You API-Ninjas API key must be included with your code or accessible from your code, as we will test your app using your key.
  - Make sure there are enough API calls remaining in your account for us to test with.

How to use your API-Key

1. You can assign a variable in your code to your key. This is not secure coding practice as discussed in the lecture.
2. Alternatively, use the ENV command in Dockerfile to set the value of your NINJA API key and then have the program read that environment variable at run-time.
3. We will discuss a better method, using an ".env" file when we use Docker Compose for the next assignment.

You will not be graded on where you put your NINJA API key. However, you must make sure that when the **TA runs your code, it will run without him having to do anything else.**

Submitting your assignment

If submitting work with a partner, please **attach a document listing all the members working on the assignment**.

This must be done for each assignment, as partners can change from one assignment to another.

The name of the zip file (see next slide) should include the names of the partners on the assignment. E.g., partner1_partner2.zip

Points will be deducted if these and other instructions are nor followed.

Docker

The code you write will need to run in a Docker container. This means that you need to submit:

- A Dockerfile which automatically builds a docker image to run your program
- Any files that the Dockerfile needs to build the image (which includes any code or auxiliary files you use)

- You should submit all of you code in a zip file. If your Dockerfile assumes a specific file structure (e.g., it assumes that code on the host is in a subdirectory) then unzipping the zip file should preserve that directory structure.
- **You should test that your assignment works before submitting.** I recommend that you give your zip file to a friend to unzip and issue the Docker build and run commands. I.e., make sure that it will work for the TA (not only on your computer) before submitting.

Grading of the assignment (1)

We will test your code by:

- Running your Dockerfile to build the image
- Creating a container from the image
- Issuing REST requests to the container on the specified port and checking that the correct responses are returned.

Grading of the assignment (2)

The TA will issue REST requests on your docker container:

- Does the docker build command work? Does it create the right image?
- Does the container built from the image run your code without a problem? Is it running at the right IP address?
- Does it process POST, GET, DELETE and PUT requests properly?
- Does it route the requests for different resources to the correct code?
- Does it return the right status codes for each request?
- Does it return the right JSON for each request?
- Does it handle incorrect requests properly, returning the right status codes and JSON error messages?

Docker port

- You should have your API server application listen for HTTP (API) requests on port 5001
  - In other words, the Docker container port is 5001. When we test the program, we will forward requests (using --publish) from the host to the container port 5001.
- You should submit your source code as part of the zip file, even if you do not need it to run your application (e.g., you are using compiled code for instance).
- If your docker file does not run, it will be returned to the student to resubmit and point