

Topic 2, part I: Encapsulation, Interfaces, and RESTful APIs

Dr. Daniel Yellin

THESE SLIDES ARE THE PROPERTY OF DANIEL YELLIN
THEY ARE ONLY FOR USE BY STUDENTS OF THE CLASS
THERE IS NO PERMISSION TO DISTRIBUTE OR POST
THESE SLIDES TO OTHERS

Modularity and information hiding

Encapsulation and loose coupling

Encapsulation



encapsulate verb

en·cap·su·late (in-'kap-sə-'lāt) en-

encapsulated; encapsulating

transitive verb

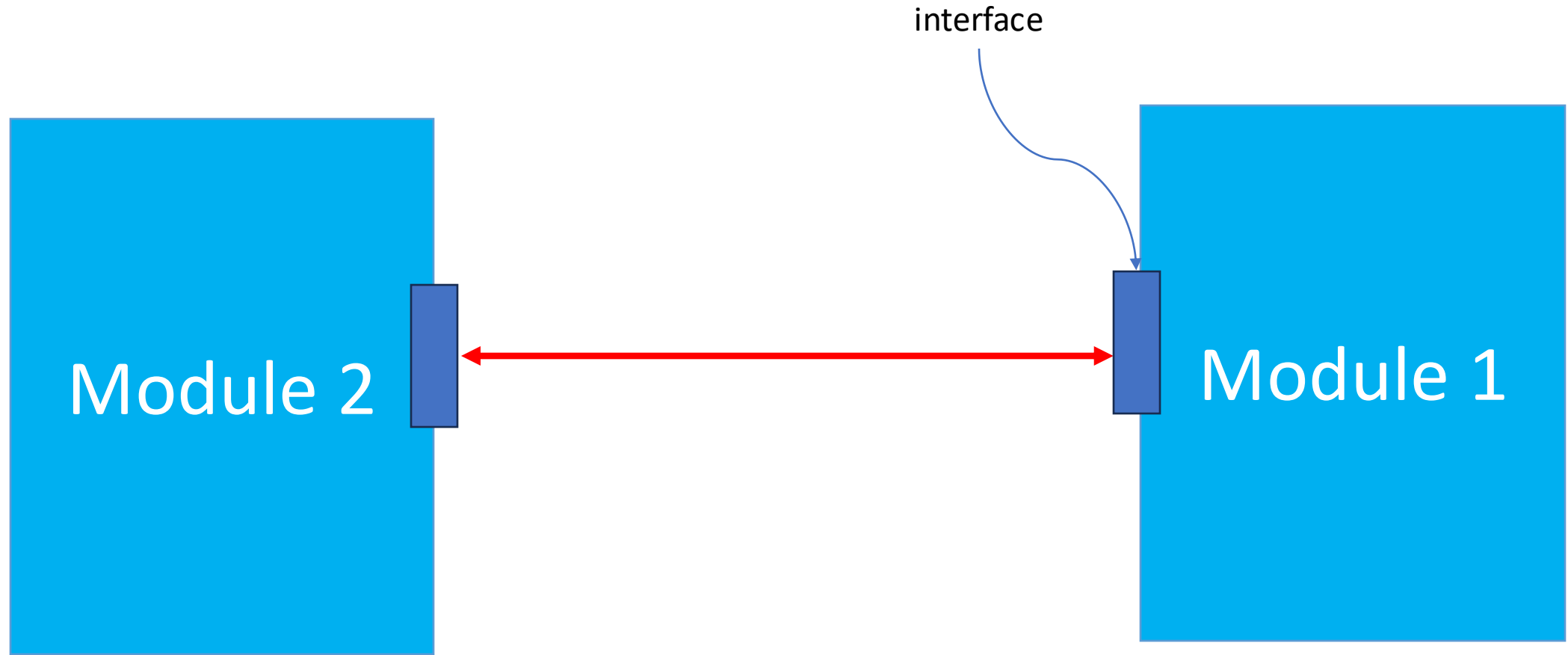
- 1 : to enclose in or as if in a capsule
| a pilot *encapsulated* in the cockpit
- 2 : **EPITOMIZE, SUMMARIZE**
| *encapsulate* an era in an aphorism

In the 1970s software engineering began to embrace *code encapsulation* and *information hiding* as key techniques for developing software:

- Partition code into separate *modules*.
- Each module has an *interface*.
- One module can talk to another module *only* through its interface.

David Parnas, a SE researcher & leading advocate for these techniques, showed 3 reasons why encapsulation and modularity are important

Modules, Interfaces, Communication

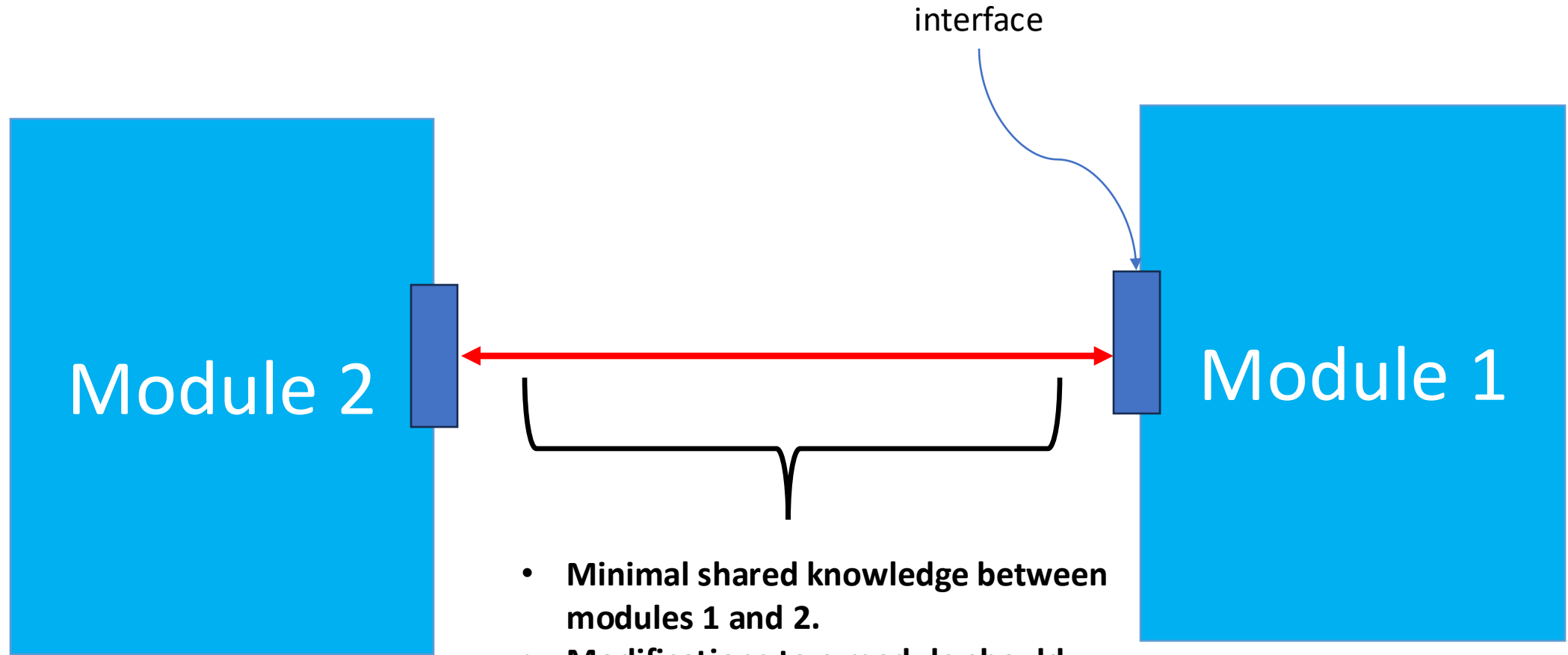


Loosely coupled architecture

A more modern expression of encapsulation goes a bit farther using the term *loose coupling*. Coupling refers to the degree of direct knowledge that one component has of another. Components in a system are *loosely coupled* if:

- The *shared knowledge* needed between components is the *minimum required*. For instance, a component in the system needs minimal knowledge of the implementation of other components, and has minimal reliance on communication protocols.
- A *change to the implementation* of one component does *not effect* (or effects *minimally*) other components.

Loose coupling



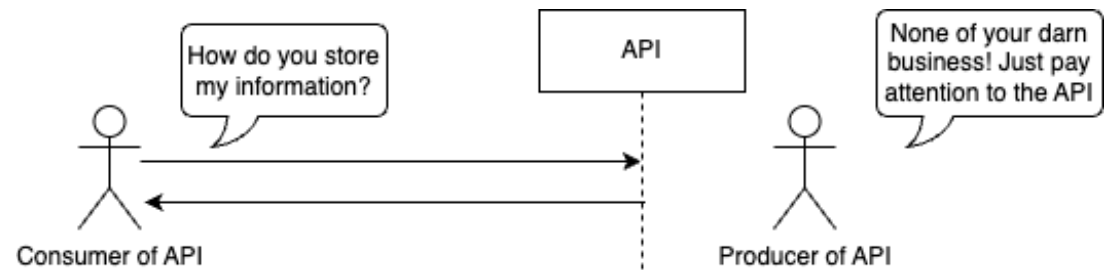
- Minimal shared knowledge between modules 1 and 2.
- Modifications to a module should usually not change the interface and not require changes to other modules.

Reasons for encapsulation & information hiding

1. *Independent development.*

Different teams can work on different parts of a system with minimum interaction needed between them.

This requires *information hiding*: A consumer of a module only needs to understand its interface to use it. He does not need to understand its implementation.



Reasons for encapsulation & informing hiding (cont)

2. *Maintainability (locality of change)*. A change can be made to one module without affecting other modules.
3. *Comprehensibility*. Increased comprehensibility due to the ability to understand a module independent from the others. Ability to understand the whole program by looking at interfaces and communication between modules.



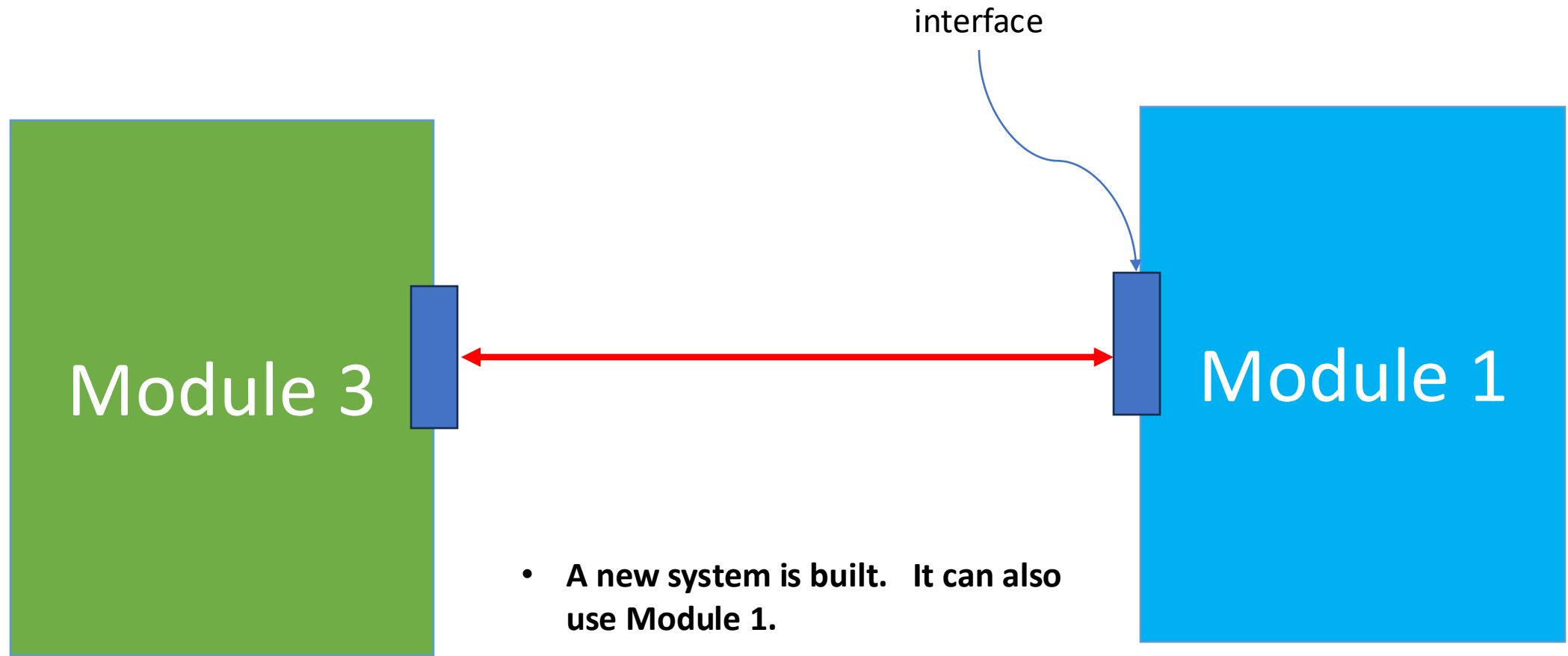
Reasons for encapsulation & informing hiding (cont)

As software became a large industry, a search for software productivity ensued. Another reason for encapsulation and modularity surfaced:

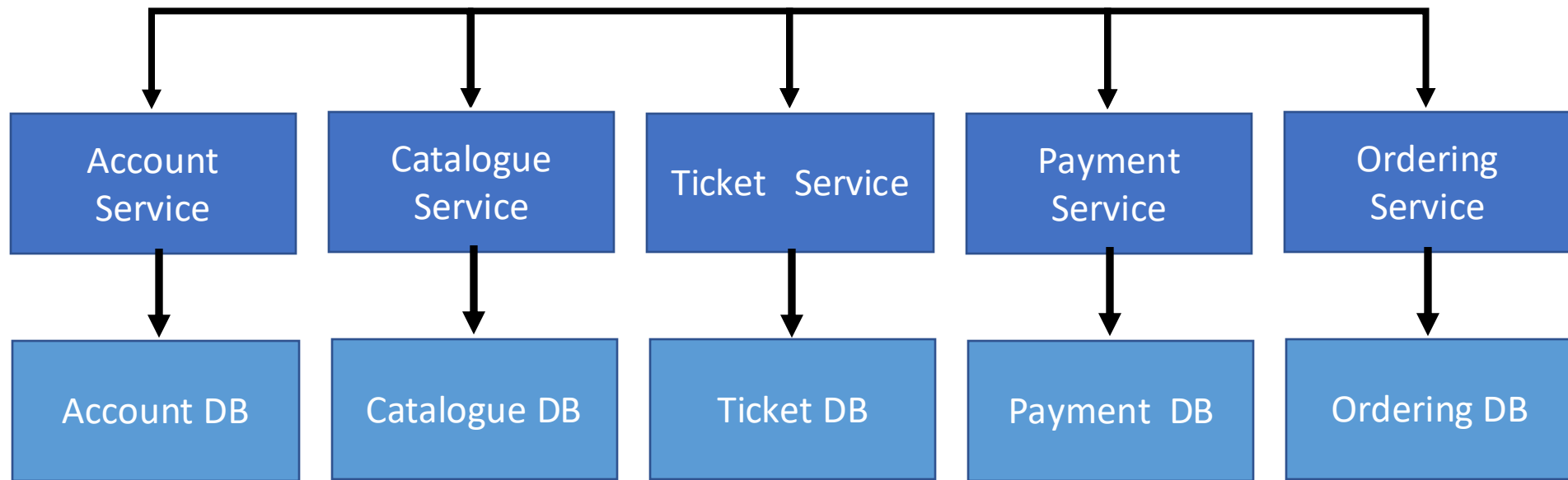
4. *Reusability*. By properly separating modules into self-contained independent units, they can be reused in different contexts and combined in different ways. Researchers formalized when substitution produces the desired affects. For object-oriented programming languages, this is known by the *Liskov substitution principle* and *behavioral subtyping*.



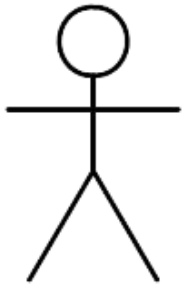
Reusability



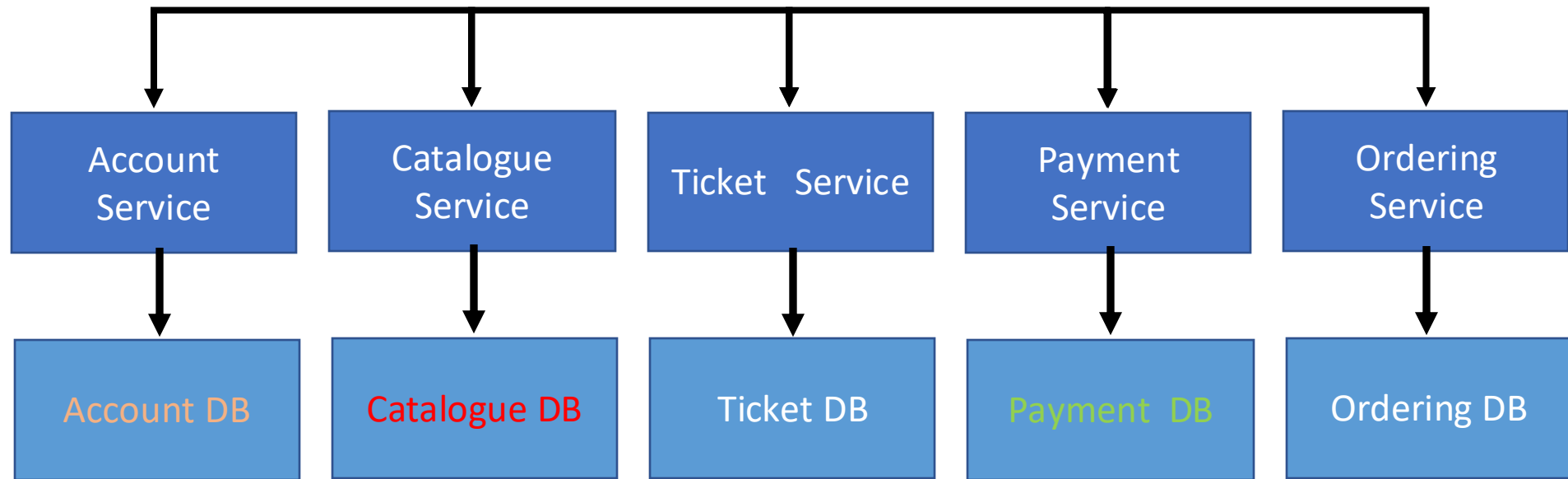
Consider the following system



How to implement the following query?

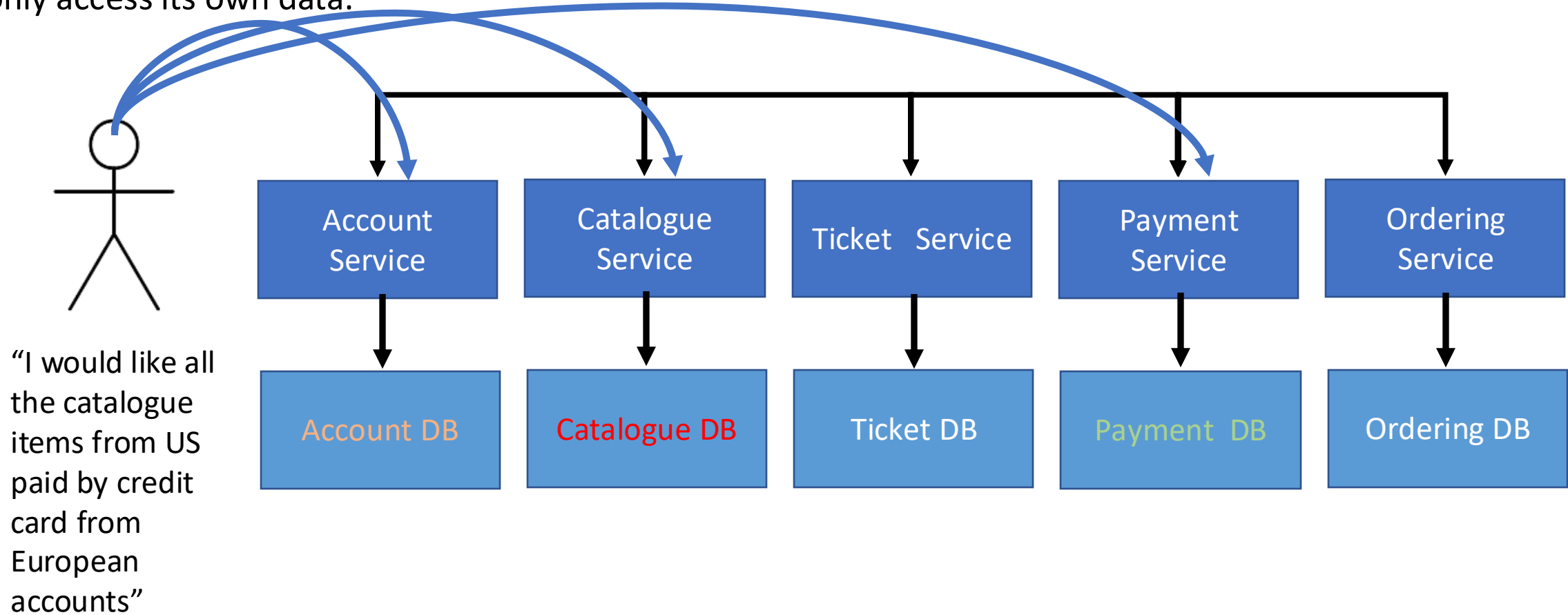


“I would like all
the **catalogue**
items from US
paid by **credit**
card from
European
accounts”



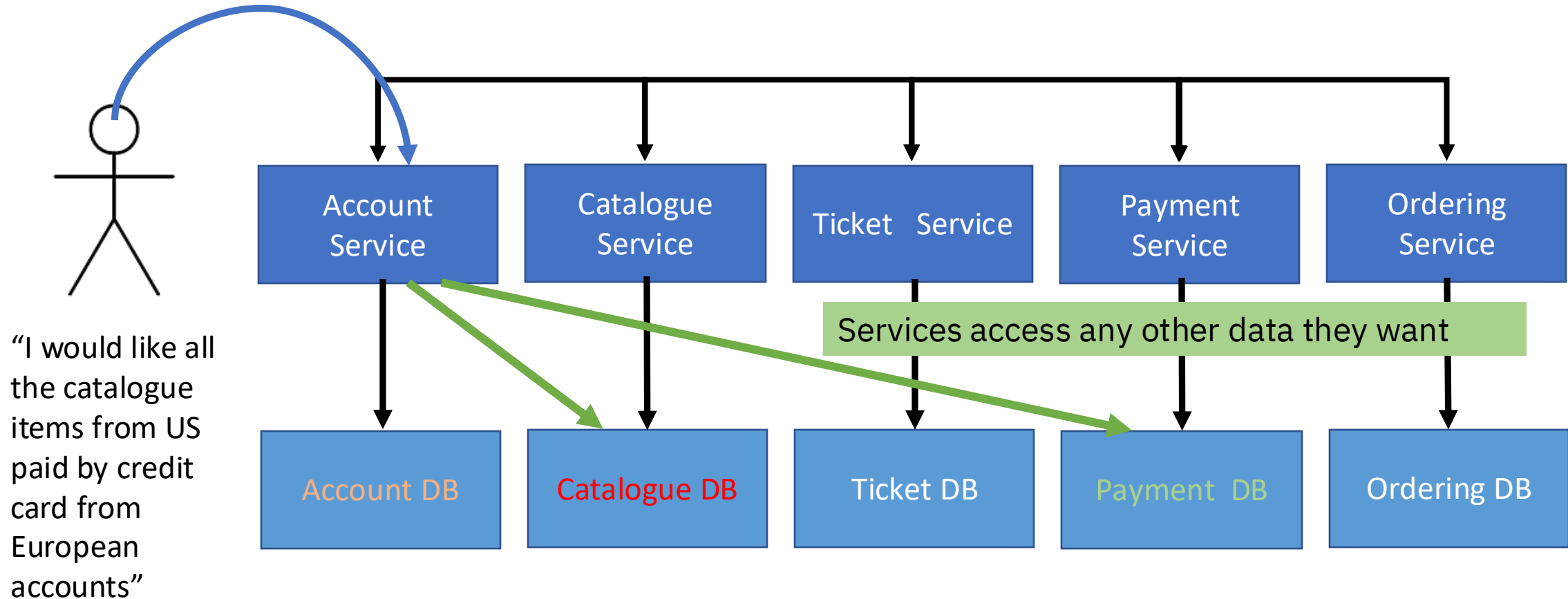
Approach #1

Business logic calls the different services & coordinates information flow between them. Each component can only access its own data.



Approach #2

Allow the account service to process the query by directly gathering the data required



Which approach is best

- Approach #2 can be more efficient. It does not require the overhead of making multiple calls to various services, retrieving extraneous data through standardized interfaces, and having to integrate the data from the different services.
- However approach #2 violates the idea of the *loosely coupled systems*:
 - No longer is minimal information shared between components. The account service must know the data schema and other information about the catalogue and payment databases.
 - A change to one database may affect any number of other components in the system.

Implications on lack of and information hiding

When systems lack encapsulation and information hiding the following can occur:

- Hard to understand and debug the system, as changes are non-local
- Very difficult to maintain and modify the system
 - If you change the data model in a specific component, who else does it affect?
 - If you want to add some logic that will be invoked everytime a specific data element is about to be changed, where do you put that logic?
 - If you need to determine (and limit) access rights to some data, how is this accomplished?

Enhancing loose coupling of components

- Communicate is via standard protocols (like HTTP) and data formats (such as XML or JSON).
- Minimize hardwiring of addresses of components.
 - Meaning that component X should not need to know a priori the address of Y to communicate with it. Instead, X can dynamically find out Y's address or it can use a message broker to send messages to Y.
- Build components to be resilient to changes in the message format.
 - For instance, the component can process messages even if new data elements are added, data elements are deleted, or the order of data elements are changed in the received messages.

Summary: 4 benefits of encapsulation and data hiding

1. Independent development
2. Maintainability
3. Comprehensability
4. Reusability

Another reason for modularity and encapsulation will be discussed when we talk about microservices

Limits of modular design and information hiding

Encapsulation does not solve all problems

Limits of modular design and information hiding

Encapsulation is not sufficient to capture all information needed by the user of the interface:

1. The interface does not capture *all* aspects of the implementation necessary to reason about the use of the module.
 - Non-Functional Requirements (NFRs) describe system quality attributes for a system, as distinct from the functional requirements, which detail a system's business features and capabilities. NFRs include availability, performance, security, costs, etc.

Limits of modular design and information hiding

Encapsulation is not sufficient to capture all information needed by the user of the interface:

1. The interface does not capture *all* aspects of the implementation necessary to reason about the use of the module.
 - Non-Functional Requirements (NFRs) describe system quality attributes for a system, as distinct from the functional requirements, which detail a system's business features and capabilities. NFRs include availability, performance, security, costs, etc.
2. Internal implementation details that the caller *must* know about, such as memory allocation and thread safety.

Limits of modular design & information hiding (cont)

3. *Software evolution.* It is hard to predict how software will change, and therefore the right information to hide and how to abstract. This can be witnessed by the number of major changes to software packages that change the API.

Internet (RESTful) APIs

They are everywhere and have changed the way we write programs!

APIs

Until now we have spoken about *interfaces* in the context of decomposing a program into modules.

- Each programming language has a mechanism for defining an *Application Programming Interface* (API) using that programming language's constructs.

Roy Fielding's dissertation *Architectural Styles and the Design of Network-based Software Architectures* at UC Irvine in 2000 introduced *REpresentational State Transfer* (REST).

- Fielding described the idea of a "**network-based Application Programming Interface**" that he contrasted with traditional "library-based" APIs. [\(Wikipedia\)](#)
- Let's call them **Internet APIs** for short.

Internet APIs are everywhere!

[AbstractAPI](#)

[RapidAPI](#)

The screenshot shows the RapidAPI website interface. On the left is a sidebar with category links: Food, Transportation, Music, Business, Visual Recognition, Tools, Text Analysis, Weather, Gaming, SMS, Events, Health and Fitness, Payments, and a 'View All Categories' link. The main content area is divided into two sections: 'Recommended APIs' and 'Popular APIs'. Each section contains a grid of API cards. Each card displays an API icon, name, a brief description, a 'Verified' status with a green checkmark, and performance metrics like latency and success rate.

Recommended APIs

APIs curated by RapidAPI and recommended based on functionality offered, performance, and support!

- API-FOOTBALL**: +950 football leagues & cups. Livescore (15s), live & pre-match odds, events, line-ups, coaches, players. Verified, 10 ms latency, 337 ms response time, 100% success rate.
- VACCOVID - coronavirus, vaccine and treatment tracker**: VACCOVID.LIVE is a comprehensive up-to-date Vaccine tracker, COVID-19. 9.8 ms latency, 579 ms response time, 96% success rate.
- SendGrid**: Welcome to SendGrid's Web API v3! This API is RESTful, fully featured, and easy to integrate with. Verified, 9.9 ms latency, 288 ms response time, 100% success rate.
- Referential**: The fastest API to access countries, states, cities, continents, dial and zip codes in up to 20. Verified, 9.8 ms latency, 41 ms response time, 100% success rate.

Popular APIs

APIs that are popular and frequently used on RapidAPI!

- Recipe - Food - Nutrition**: The spoonacular Recipe - Food - Nutrition API gives you to access to thousands. Verified, 10 ms latency, 337 ms response time, 100% success rate.
- uNoGS**: uNoGS (unofficial Netflix online Global Search) allows anyone to search the global. 9.8 ms latency, 579 ms response time, 96% success rate.
- Bionic Reading**: Bionic Reading® is a new method facilitating the reading process by guiding. Verified, 9.9 ms latency, 288 ms response time, 100% success rate.
- Trip Purpose Prediction**: Understand the reason for trip with Amadeus AI APIs. The Trip Purpose Prediction. Verified, 9.8 ms latency, 41 ms response time, 100% success rate.

The screenshot shows the AbstractAPI website interface. The header reads 'Explore Abstract's library of API's'. Below the header is a grid of API cards, each with an icon, name, and a brief description.

Explore Abstract's library of API's

- IP Geolocation API**: Geolocate and get details on any IP worldwide.
- Email Verification API**: Validate any email quickly and reliably.
- Phone Validation API**: Validate any phone number worldwide.
- VAT Validation & Rates API**: Comply with VAT laws and get the latest verified.
- Website Screenshot API**: Get a screenshot of any HTML file or URL.
- User Avatar API**: Create flexible user avatars.
- Public Holidays API**: Get holidays for any country at any time.
- Image Processing API**: Compress and optimize any image.
- Web Scraping API**: Extract data from any website.
- Exchange Rates & Currencies API**: Get and convert exchange rates.
- Time, Date, and Timezone API**: Get and convert time globally.
- Company Enrichment API**: Get data on any domain or company.

Two example API Hubs

How should one specify an Internet API? Does it depend on the language the provider implements it in?

Recipe API in Java

```
import java.util.UUID;
import Ingredient.Recipes;
public enum Rating {High, Medium, Low};
interface Recipes {
    public Ingredient[] findRecipeByMain(Ingredient
        mainIngredient, int maxCalories, Rating score)
    public boolean submitRecipe(UUID myID, String
        name, Ingredient[] ingredients);
}
```

Geolocation API in Rust

```
mod Geolocation {
    use geoutils::Location;
    use uuid::Uuid;
    struct Place {
        city: String,
        state: States
    }
    pub fn geoFromIP(ip: Ipaddress,
        myKey: ID: Uuid) -> Place
    pub fn geoFromCoords(loc: Location) -> Place
}
```

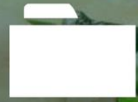
Problems with using language-based APIs for Internet APIs

- What language should the API provider pick? He wants to support lots of developers.
 - Different languages have different syntax, different semantics, different types, different libraries
- Even in the same language, different APIs can be hard to understand
 - Each API specifies its own datatypes, methods (functions), and interaction style, leading to a proliferation of types and styles
- How does the client's computer find and invoke the API?
 - The application server sits in some company datacenter (or closet) in some unknown location
 - Different organizations may provide APIs with the same name and signature.

Lets look at some real APIs and how they do it

They use a standard style defined by Fielding

All Your Food. One Place.



Save and organize
recipes from any site



Free meal planner
and food tracker



Collect your favorite
products

Start Now!

(yup, it's free)



fast

Chorizo Parmesan Brussels Spro...



Instant Pot Buffalo Garlic But...



Dijon Garlic Brussels Sprouts



Chocolate Orange Cake



“The only food API you'll ever need”

GET

<https://api.spoonacular.com/recipes/complexSearch?query=pasta&maxFat=25&number=2>

Request

Response is a JSON
(JavaScript Object Notation)
document

Response

Example Request and Response

GET

<https://api.spoonacular.com/recipes/complexSearch?query=pasta&maxFat=25&number=2>

```
{
  "offset": 0,
  "number": 2,
  "results": [
    {
      "id": 716429,
      "title": "Pasta with Garlic, Scallions, Cauliflower & Breadcrumbs",
      "image": "https://spoonacular.com/recipeImages/716429-312x231.jpg",
      "imageType": "jpg",
    },
    {
      "id": 715538,
      "title": "What to make for dinner tonight?? Bruschetta Style Pork & ",
      "image": "https://spoonacular.com/recipeImages/715538-312x231.jpg",
      "imageType": "jpg",
    }
  ],
  "totalResults": 86
}
```

[Source](#)



Analysis of the API



GET

[https://api.spoonacular.com
/recipes/complexSearch
?query=pasta
&maxFat=25
&number=2](https://api.spoonacular.com/recipes/complexSearch?query=pasta&maxFat=25&number=2)

- **GET** is a “generic verb” and says you want to fetch information

Analysis of the API



GET

[https://api.spoonacular.com](https://api.spoonacular.com/recipes/complexSearch?query=pasta&maxFat=25&number=2)
[/recipes/complexSearch](https://api.spoonacular.com/recipes/complexSearch?query=pasta&maxFat=25&number=2)
[?query=pasta](https://api.spoonacular.com/recipes/complexSearch?query=pasta&maxFat=25&number=2)
[&maxFat=25](https://api.spoonacular.com/recipes/complexSearch?query=pasta&maxFat=25&number=2)
[&number=2](https://api.spoonacular.com/recipes/complexSearch?query=pasta&maxFat=25&number=2)

- **GET** is a generic verb and says you want to fetch information
- **URL** says where to find this API

Analysis of the API



GET

<https://api.spoonacular.com>

[/recipes/complexSearch](#)

?query=pasta

&maxFat=25

&number=2

- **GET** is a generic verb and says you want to fetch information
- **URL** says where to find this API
- The additional **PATH** tells which thing you are talking about (in this case, Recipes)

Analysis of the API



GET

<https://api.spoonacular.com/recipes/complexSearch>

?query=pasta
&maxFat=25
&number=2

- **GET** is a generic verb and says you want to fetch information
- **URL** says where to find this API
- The additional **PATH** tells which *thing* you are talking about (in this case, Recipes)
- The **query string** lists parameters to the operation

IP Geolocation API

“Get the location of any IP with a world-class API serving city, region, country and lat/long data”

Request

abstract

APIs ▾ Documentation Pricing Login [SIGN UP](#)

IP Geolocation API


Get the location of any IP with a world-class API serving city, region, country and lat/long data.


[Get started](#)


```
https://ipgeolocation.abstractapi.com/v1/  
? api_key = YOUR_UNIQUE_API_KEY  
& ip_address = 92.184.105.98
```


```
{  
  "ip_address": 92.184.105.98,  
  "city": "Caen",  
  "city_geoname_id": 3029241,  
  "region": "Normandie",  
  "region_iso_code": "FR-NOR",  
  "region_geoname_id": 11071621  
  "postal_code": "14949"  
  "country": "France"  
  "country_code": "FR"  
  "country_geoname_id": 3017382  
}
```

Response

 Worldwide coverage with 4 billion IPs

 Location data with leading precision

 Rich data including timezone and unicodes

 Real-time daily update

IP Geolocation API

GET

https://ipgeolocation.abstractapi.com

/v1/

? api_key = YOUR_UNIQUE_API_KEY
& ip_address = 92.184.105.98

- Same structure:

- GET
- URL
- PATH
- query string

IP Geolocation API

GET

<https://ipgeolocation.abstractapi.com>

/v1/

? api_key = YOUR UNIQUE API_KEY

& ip_address = 92.184.105.98

- Same structure:
 - **GET**
 - **URL**
 - **PATH**
 - **query string**
- Response is a JSON document
- The query string contains an **API-KEY**

News API

GET

`https://newsapi.org`
`/v2/topheadlines`
`?country=us`
`&apiKey=API_KEY`

Response is a JSON document

Top headlines in the US

Definition

`GET https://newsapi.org/v2/top-headlines?country=us&apiKey=API_KEY`

Example response

```
{
  status: "ok",
  totalResults: 36,
  articles: [
    {
      source: {
        id: null,
        name: "CBS Sports"
      },
      author: "Jordan Dajani",
      title: "Jets vs. Jaguars score: Trevor Lawrence defeats benched Zach Wilson for Jacksonville's third straight win - CBS Sports",
      description: "Jacksonville dominated New York on 'Thursday Night Football,' and is closing in on the AFC South",
    }
  ]
}
```

[Source](#)

The API Economy

- Over 90% Of Developers Use Internet APIs and REST is used extensively
 - **83% of public APIs** use REST architecture as of Q4 2024 ([RapidAPI Developer Survey 2024](#))
 - **92% of Fortune 1000 companies** have REST APIs in production ([MuleSoft Connectivity Benchmark Report](#))
- Developers Spend 30% of Their Time Coding APIs
- API Management Market Valued at \$5.1 Billion by 2023
- There Are Over 2 Million API Repositories On GitHub
- Open Banking to Have 130 Million Users by 2024
- 91% of Organizations Had An API Security Incident in 2020

[Source1](#)

[Source2](#)

REST and alternatives

REST is a clear leader for Internet (Web) Services. However, other technologies are often used depending on the use case:

- **REST** for public APIs and simple CRUD operations
- **GraphQL** for complex data requirements and mobile apps
 - GraphQL usage has grown strongly in the last few years
- **gRPC** for internal high-performance microservices communication
- **Kafka/messaging** for event streaming, async workflows, and data pipelines

We will touch on some of these other technologies later in the course

Summary so far

- Internet APIs are really important and intrinsic to the **API Economy**
- The majority of Internet APIs today are built using **RESTful** APIs
- They are language independent, and provide a **uniform** way of expressing an API

RESTful API details

Conventions for building RESTful APIs

RESTful request structure

A RESTful request has 4 main parts:

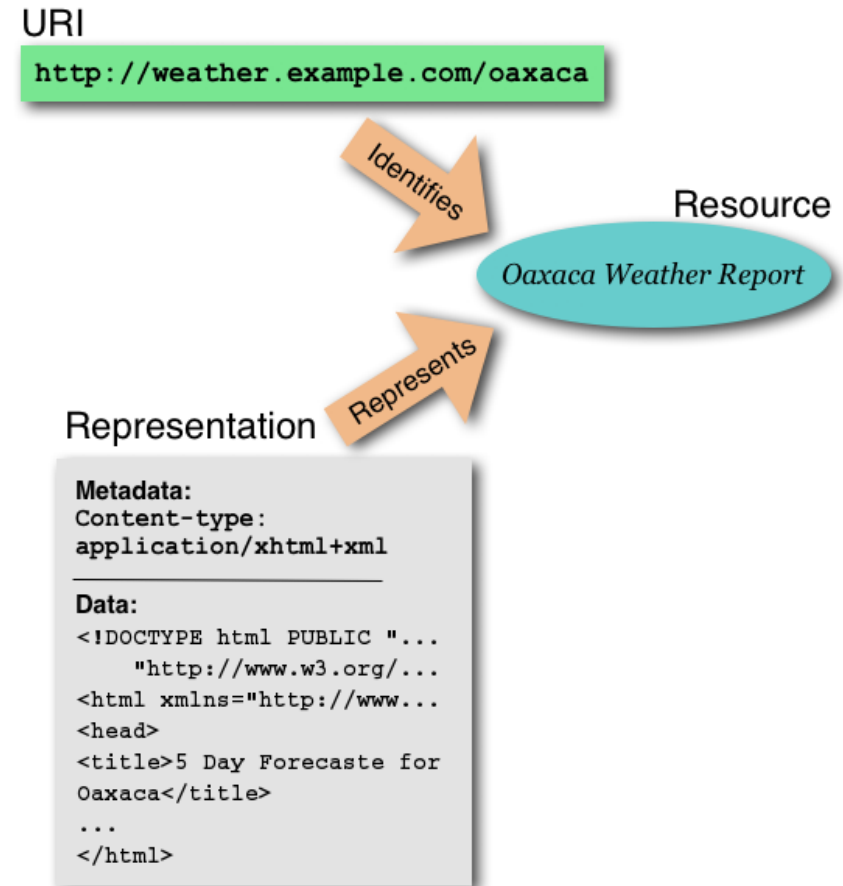
1. A **URI** (Universal Resource Identifier) that describes *which resource* (entity) you are referring to. This is also called the *endpoint*.
2. A “**verb**” describing *what action* you want to perform on that resource.
3. A **body** or **payload** providing *additional information*, if any, you need to accomplish the action.
4. A **Header** that contains additional information about the request, such as *content type* or *authentication credentials*.

Resources and representations

1. An **Identifier** specifies which *resource* (entity) is acted upon in a REST request.
2. While a **resource** refers to a *conceptual* object, a **representation** of the resource is a sequence of bytes that gets transmitted.
3. A **Self descriptive message** conveys *meta-data* about the representation of the message payload; it describes the *content (media, Mime) type* of the resource in the message payload.
 - Example content types: image/gif, text/html, image/jpeg, application/json, video/mpeg, application/pdf.

Resources and representations (cont)

- A **uniform resource identifier (URI)** is a standard mechanism to identify a resource (an entity).
- A **resource** may be a jpeg image, a product in a product catalogue, or even dynamically generated information, such as “current weather in Tel Aviv”.
- The **representation** of the resource encodes information about resource state.
- **Meta data** accompanying the representation describes its **media (content) type**. There can be multiple representations for the same resource, and which one is chosen can be negotiated dynamically.



Uniform Resource Identifier (URI)

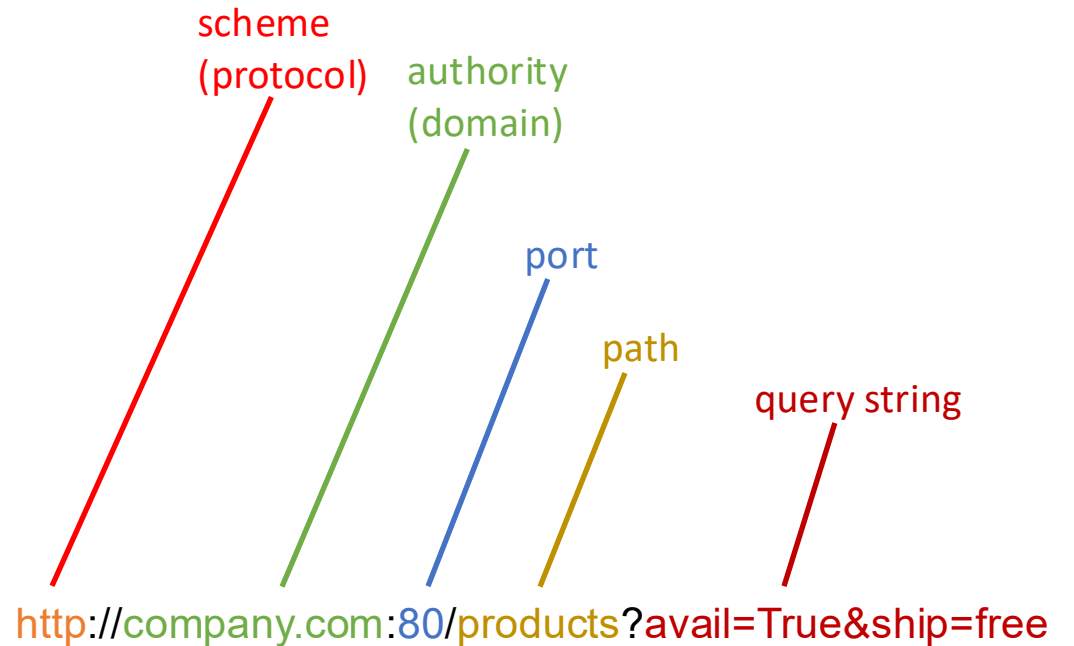
Scheme (protocol) HTTP (HTTPS) is the defacto protocol used by REST. The scheme dictates how identifiers are defined.

Authority (Domain/host): the host where this resource is found.

Port: optional port on the host used to locate this resource.

Path: additional identifying information needed by the host to identify this **resource**.

Query string: optional parameters for refining which resource to be returned. E.g., It may filter or search a collection for specific items meeting some criterion.



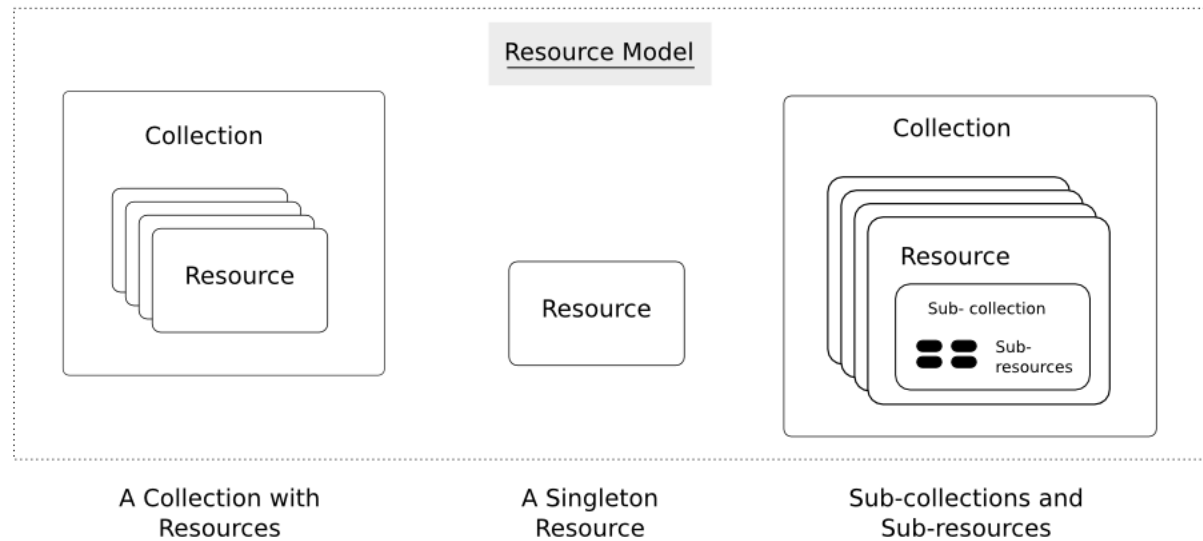
REST convention for naming resources: Nouns

Nouns – collections and singletons are commonly used to specify *resources*:

- A collection resources:
 - <http://company.com/customers>
- A single resource in a collection:
 - <http://company.com/customers/12>
- Subcollections:
 - <http://company.com/customers/{customerId}/purchases>
 - <http://company.com/customers/{customerId}/purchases/{transID}>
- Filtering collections:
 - <http://company.com/customers?region=USA&buysAlot=True>

Picking intuitive and appropriate names is important for designing a useable API

Singleton and Collection Resources



REST convention for methods: HTTP Verbs

- Client must specify to the server the action that it wants to perform on the resource
- It uses HTTP methods, known as **HTTP verbs**. The HTTP standards assigns exact semantics to each verb.

Verb	Description	
POST	Sends a resource representation using POST to create a new resource on the server	C
GET	Reads a resource using a GET request to an endpoint.	R
PUT	Replaces an existing resource in its entirety	U
PATCH	Partially alters data to an existing resource without replacing it.	
DELETE	Delete a resource.	D

- There are additional HTTP verbs, but we will not discuss those here

Message payload and header

- POST and PUT requests include a payload, giving details on the resource being created or updated.
- Example:

The diagram shows an HTTP POST request with several components annotated by red arrows:

- Verb:** Points to the word `POST`.
- URI (endpoint):** Points to the URL `'http://www.myhost.com/todolist'`.
- Header:** Points to the line `-- header 'Content-Type: application/json'`.
- Media type (of payload):** Points to the value `application/json` in the header.
- Payload:** Points to the JSON data block starting with `-- data {`.

```
POST 'http://www.myhost.com/todolist'
-- header 'Content-Type: application/json'
-- data {
  "item": "prepare supper",
  "when": "July 7",
  "priority": "high"}
```

- GET and DELETE requests typically do not include a payload - the resource is identified in the URI and no additional information is needed to carry out the request. Example: `GET http://0.0.0.0:80/todolist/{ID}`

JSON

JSON is the most popular format for textual data: ubiquity, simplicity, readability, scalability, flexibility.

In JSON, values must be one of the following data types:

- a string
- a number
- an object (JSON object)
- an array
- a boolean
- *null*

JSON example:

```
{
  "title": "War and Peace",
  "author": "Leo Tolstoy",
  "published_year": "1869",
  "num_pages": 1225,
  "reviews": [
    { "publication": "NYTimes",
      "review": "Great read! ..." },
    { "publication": "The Atlantic",
      "review": "Required reading for ..." },
    ...
  ]
}
```

RESTful response structure

A RESTful response has 3 main parts:

1. A *status code*, telling you if the request was successful, or why it failed.
2. A *header* containing metadata about the response (optional)
3. A *body* or *payload* that contains the request data
 - Not all REST requests return a payload:
 - A GET request returns a representation of the requested resource.
 - A POST request will usually return a URI or ID for the newly created resource.*

In this class you can assume that all payloads are JSON.

* For our first assignment, a POST request will return a JSON object being POSTed. Alternatively it just returns a link to the created object. We will talk about this more at the end of the lecture.

Common Response Status Codes

Code	Meaning	Description
200	OK	The requested was successfully executed
201	Created	A new resource was created
202	Accepted	The request was received but no modification has been made yet
204	No Content	The request was successful but the response has no content
400	Bad Request	The request was malformed
401	Unauthorized	The client is not authorized to perform the requested action
404	Not Found	The requested resource was not found
415	Unsupported Media Type	The request data format is not supported by the server
422	Unprocessable Entity	The request data was properly formatted but contained invalid or missing data
500	Internal Server Error	The server threw an error when processing the request

JSON format for error messages

The following are the JSON objects returned with error status codes:

- {"error": "Malformed data"}, 400
- {"error": "Not found"}, 404
- {"error": "Expected application/json media type"}, 415
- {"server error": str(e)}, 500
where `e` is the exception raised by the server
- {"server error": "API response code " + str(response.status_code)}, 500
where `response.status_code` is the exception status code returned by a call to an external API.

Query strings

A query string is used to refine the query so only resources that satisfy the query string are returned.

GET 'http://www.myhost.com/todolist?name=homework'

GET 'http://www.xyz.com/products?product=shirt&color=blue'

Summary

RESTful requests specify:

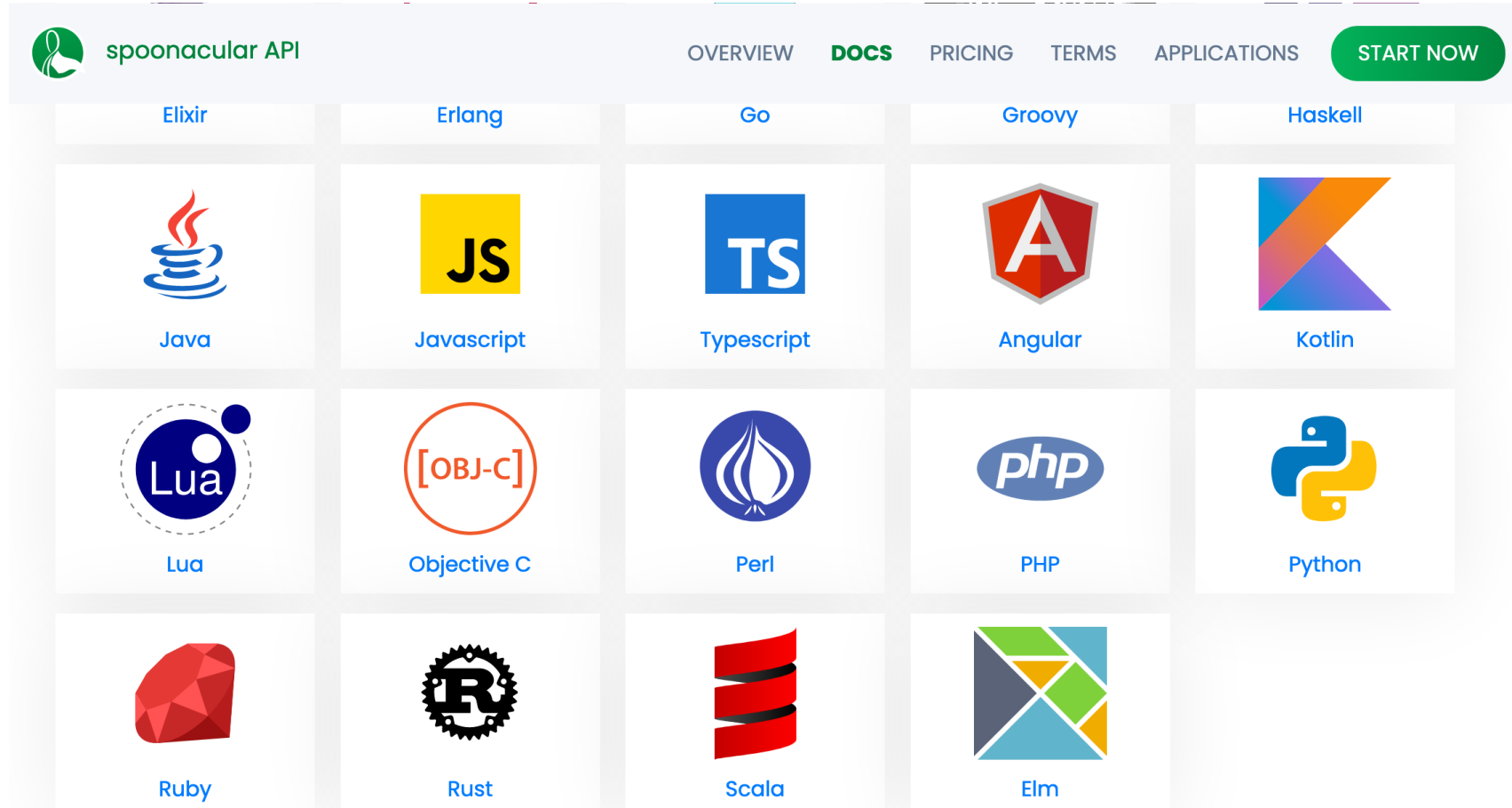
1. A **URI** (Universal Resource Identifier): describing the resource, the object you are acting upon.
2. An HTTP “**verb**” describing *what action* you want to perform on that that resource.
3. A **payload** providing additional information, if any, you need to accomplish the action.
4. A **header** specifying meta information, such as the content type of resource or authentication credentials.

RESTful APIs by example

Using Python to invoke and provide REST APIs

Language Bindings:

Libraries facilitate building & invoking RESTful APIs for specific languages



[Source](#)

Programming language bindings to REST

A programming language **binding** must:

- For REST invocations:
 - Provide a way of specifying the HTTP operation and parameters. It returns the response payload and status in a way natural to that programming language.
 - Allow the programmer to specify and read header information.
- For serving REST methods:
 - Provide a mechanism for specifying which HTTP requests get routed to which methods/functions in the code
 - e.g., `GET http://myself.com/todolist/5` gets routed to function `find-item-in-todo(5)`.
 - Provide a mechanism for server code implementing a request to read the parameters and header of the request and to send back a payload with metadata and header information.
 - Provide other functions making it easy for the programmer to use.

Python

- Python programs often use the **requests** package for client-side invocation of REST APIs*
- 3 popular Python packages for building Server-side REST APIs are **Flask**, Django, and FastAPI
- See: <https://realpython.com/api-integration-in-python/>
- In the rest of this section, we build a RESTful API for a “Toys server” written in Python using Flask. This server keeps an inventory of toys.

* Don't confuse the Python **requests** package with the **request** object used by Flask.

Python RESTful Toy Server

- REST API to manage information on toys in inventory (in a toy store).
- Supports traditional CRUD operations on toys (create, read, update, delete)
- Application written in Python
- Use Postman or Curl to test that it works correctly
- Code in GIT repository

Resources and operations

Resources:

/toys

/toys/{id}

Operations on resources:

/toys: GET, POST

/toys/{id}: GET, DELETE, PUT

Representing a toy in JSON:

```
{  
  'id': 'string',  
  'name': 'string',  
  'age': int,  
  'price': 'float',  
  'features': 'array'  
}
```

Each toy has a unique ID associated with it. The server will always assign IDs to a resource on a POST request and return that value to the caller.

'name' is the name of the toy, 'age' is the minimum age the toy is fit for, 'price' is how much it costs, and 'features' is an array of strings, each one describing a feature of the toy.

Implementing Toys Server using Flask framework

- Import the Flask class and other required modules
- Create a Flask application (app associated with this python code). Initialize an empty dictionary Toys.
- In the main program, start the Flask app, and tell it what host and port to run on.
- There are two main approaches in Flask to map REST requests to Python functions. We will illustrate one of the approaches.

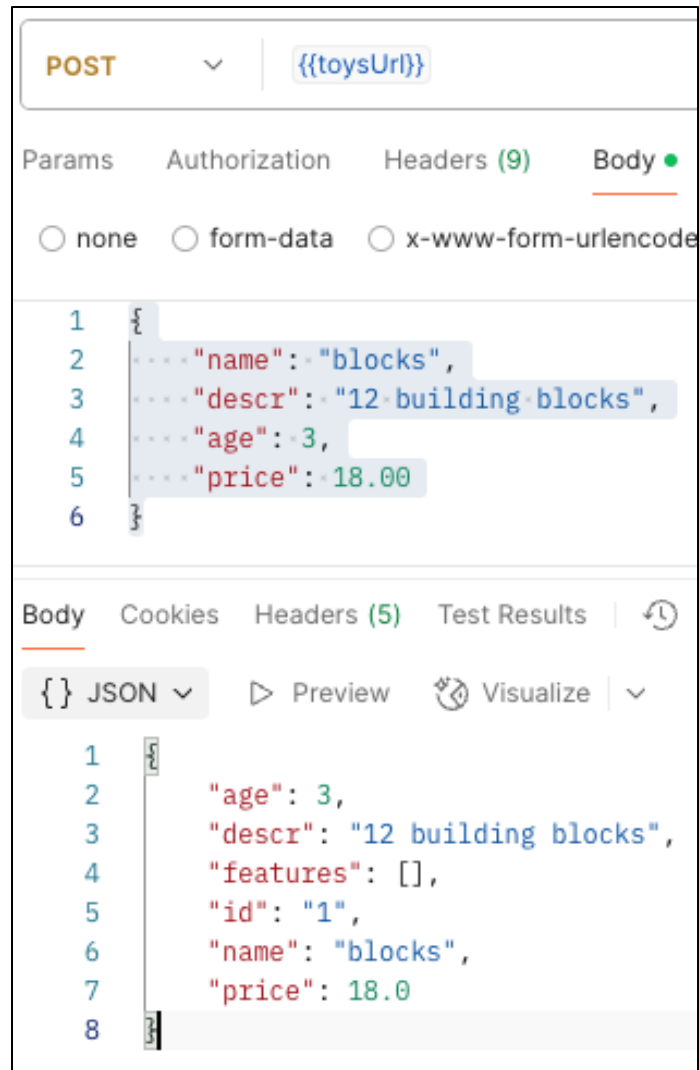
```
from flask import Flask, jsonify, request, make_response
import json
from genID import genID    # generates a unique string ID

app = Flask(__name__)    # initialize Flask
Toys = {}
```

```
if __name__ == '__main__':
    print("running toys server")
    # run Flask app.    default port is 5000
    app.run(host='0.0.0.0', port=8001, debug=True)
```

↑
This line tells Flask to run the app on my host (0.0.0.0) on port 8001

Issuing a POST request for a toy



This uses the POSTMAN tool to issue a POST request to the given URI. `{{$toysUrl}}` is a POSTMAN variable set to <http://localhost:8001/toys>. This request sends the payload using JSON.

www.postman.com to download postman

This shows the response to POST request, which returns a 201 status code and a JSON structure containing the id "1" (a string) of the created resource.

Using Curl or Python to issue a POST request

To perform the same POST from the command line, one can use curl:

```
curl --location 'http://0.0.0.0:8001/toys' \
--header 'Content-Type: application/json' \
--data '{
  "name": "blocks",
  "descr": "12 building blocks",
  "age": 3,
  "price": 18.00
}'
```

```
import requests
import json

data = {
    "name": "blocks",
    "descr": "12 building blocks",
    "age": 3,
    "price": 18.00
}

urlToys = 'http://localhost:8001/toys'
json_data = json.dumps(data)
response = requests.post(urlToys,
                          headers={"Content-Type": "application/json"},
                          data=json_data)
print(response.json())
```

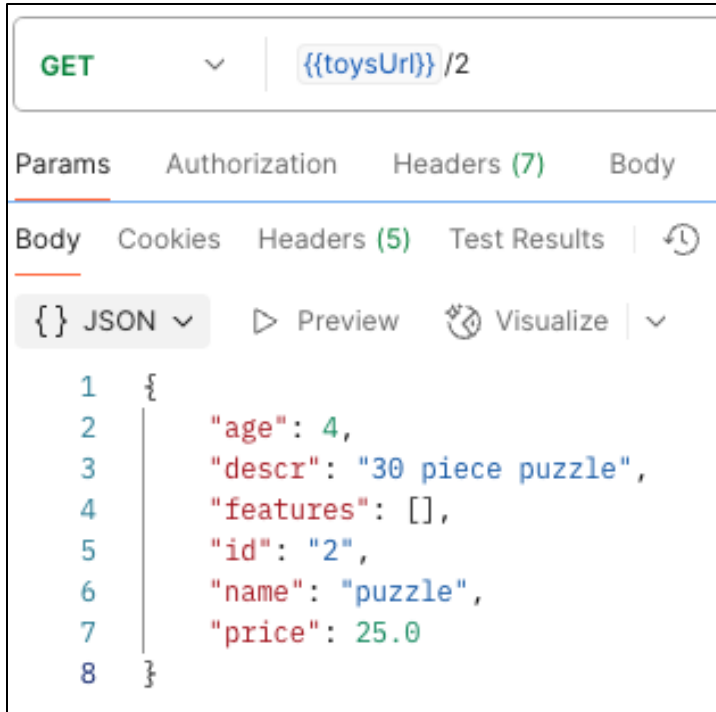
Server code to process a POST request

```
@app.route(rule: '/toys', methods=['POST'])
def addToy():
    print("POST toys")
    try:
        content_type = request.headers.get('Content-Type')
        if content_type != 'application/json':
            return jsonify({"error: Expected application/json"}), 415
        data = request.get_json()
        # Check if required fields are present
        required_fields = ['name', 'age', 'price']
        if not all(field in data for field in required_fields):
            return jsonify({"error": "Malformed data"}), 400
        newID = genID() # returns string ID
        if 'features' not in data:
            features = []
        else:
            features = data['features']
```

```
        if 'descr' not in data:
            descr = "Not Available"
        else:
            descr = data['descr']
        toy = {
            'id': newID,
            'name': data['name'],
            'descr': descr,
            'age': data['age'],
            'price': data['price'],
            'features': features
        }
        Toys[newID] = toy
        return toy, 201
    # return jsonify(toy), 201
except Exception as e:
    print("Exception: ", str(e))
    return jsonify({"server error": str(e)}), 500
```

Python Toy-server code. Note the different status codes returned on success or errors.
The genID() func returns a *unique string* ID.

GET /toys/{id} and GET /toys



```
GET {{toysUrl}}/2
```

Params Authorization Headers (7) Body

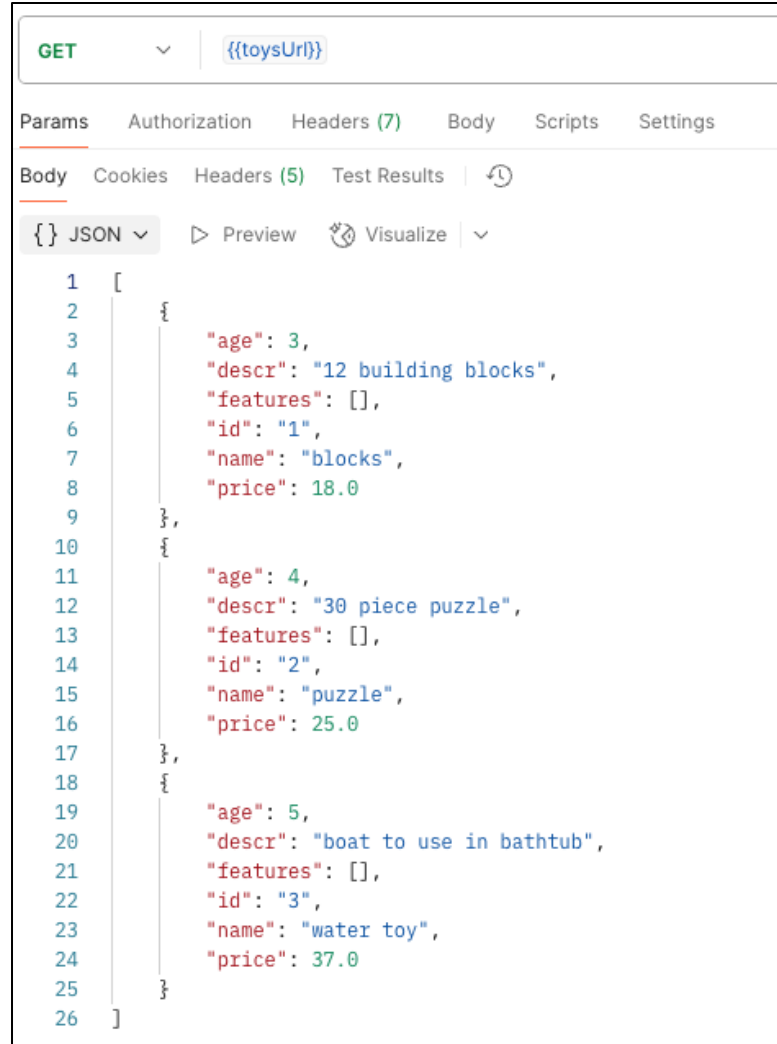
Body Cookies Headers (5) Test Results

{} JSON Preview Visualize

```
1 {
2   "age": 4,
3   "descr": "30 piece puzzle",
4   "features": [],
5   "id": "2",
6   "name": "puzzle",
7   "price": 25.0
8 }
```

Above: request for the toy with ID = 2.

Right: request for all toys. The return value is an array of JSON objects.



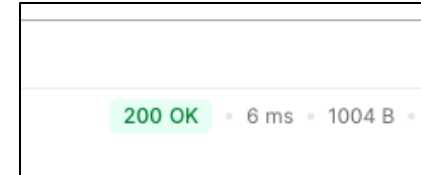
```
GET {{toysUrl}}
```

Params Authorization Headers (7) Body Scripts Settings

Body Cookies Headers (5) Test Results

{} JSON Preview Visualize

```
1 [
2   {
3     "age": 3,
4     "descr": "12 building blocks",
5     "features": [],
6     "id": "1",
7     "name": "blocks",
8     "price": 18.0
9   },
10  {
11    "age": 4,
12    "descr": "30 piece puzzle",
13    "features": [],
14    "id": "2",
15    "name": "puzzle",
16    "price": 25.0
17  },
18  {
19    "age": 5,
20    "descr": "boat to use in bathtub",
21    "features": [],
22    "id": "3",
23    "name": "water toy",
24    "price": 37.0
25  }
26 ]
```



```
200 OK • 6 ms • 1004 B
```

In both cases the return code is 200.

Server code to process GET requests

```
@app.route('/toys', methods=['GET'])
def getToys():
    try:
        return list(Toys.values()), 200
    except Exception as e:
        print("Exception: ", str(e))
        return jsonify({"server error": str(e)}), 500

@app.route('/toys/<string:toyid>', methods=['GET'])
def getToy(toyid):
    try:
        toy = Toys[toyid]
    except KeyError:
        print("GET request error: No such ID")
        return jsonify({"error": "Not found"}), 404
    except Exception as e:
        print("Exception: ", str(e))
        return jsonify({"server error": str(e)}), 500
    return jsonify(toy), 200
```

In order to return a JSON array of objects, we first take the values of the dictionary and then convert it to a list. What happens if you instead return `list(Toys)` or return `Toys.values()`?

What does jsonify do?

- **Serialization to JSON:**

It takes the provided Python data (e.g., a dictionary) and serializes it into a JSON string. This is similar to what `json.dumps()` does from Python's built-in `json` module.

- **Creates a Flask Response Object:**

It then wraps this JSON string within a `Flask Response` object. This `Response` object is what Flask ultimately sends back to the client.

- **Sets Content-Type Header:**

`jsonify` automatically sets the Content-Type HTTP header of the response to `application/json`.

Newer versions of flask **automatically call jsonify** when returning a value. It is not strictly necessary to call `jsonify` except:

- If you want to change the header
- To have more control, and to be explicit on the conversion.

Delete /toy/{id} request and server code

DELETE ▼ | http://0.0.0.0:8001/toys/2

Params | Authorization | Headers (7) | Body | Scripts | Settings

Query Params

	Key	Value	Description
--	-----	-------	-------------

Body | Cookies | Headers (4) | Test Results

204 NO CONTENT

```
@app.route('/toys/<string:toyid>', methods=['DELETE'])
def delToy(toyid):
    try:
        del Toys[toyid]
        return '', 204
    except KeyError:
        print("GET request error: No such ID")
        return jsonify({"error": "Not found"}), 404
    except Exception as e:
        print("Exception: ", str(e))
        return jsonify({"server error": str(e)}), 500
```

PUT and GET requests

PUT ▼ http://0.0.0.0:8001/toys/1

Params Authorization Headers (9) **Body** ● Scripts

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw

```
1 {
2   "id": "1",
3   "name": "blocks",
4   "descr": "12 building blocks",
5   "age": 3,
6   "price": 18.00,
7   "features": ["Each block 5 cm x 5 cm", "Meets NYS child safety standards"]
8 }
```

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON ▼

```
1 {
2   "id": "1"
3 }
```

200 OK = {}

GET ▼ http://0.0.0.0:8001/toys/1

Params Authorization Headers (7) Body Scripts

Query Params

Key

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON ▼

```
1 {
2   "age": 3,
3   "descr": "12 building blocks",
4   "features": [
5     "Each block 5 cm x 5 cm",
6     "Meets NYS child safety standards"
7   ],
8   "id": "1",
9   "name": "blocks",
10  "price": 18.0
11 }
```

200 OK = {}

PUT server code

```
@app.route('/toys/<string:toyid>', methods=['PUT'])
def update(toyid):
    try:
        content_type = request.headers.get('Content-Type')
        if content_type != 'application/json':
            return jsonify({"error": "Expected application/json media type"}), 400
        data = request.get_json()
        # Check if required fields are present
        required_fields = ['name', 'age', 'price']
        if not all(field in data for field in required_fields):
            return jsonify({"error": "Malformed data"}), 400
        if 'features' not in data:
            features = []
        else:
            features = data['features']
        if 'descr' not in data:
            descr = "No description Available"
        else:
            descr = data['descr']
        if not Toys[toyid]:
            print("PUT error:No such ID")
            # no toy of this id exists
            return jsonify({"error": "Not found"}), 404
```

```
toy = {
    'id': toyid,
    'name': data['name'],
    'descr': descr,
    "age": data["age"],
    'price': data['price'],
    'features': features
}
Toys[toyid] = toy
return toy
except Exception as e:
    print("Exception: ", str(e))
    return jsonify({"server error": str(e)}), 500
```

Alternative return value

- As we shall see, HATEOS advocates not returning a data object, but just a link to the object.
- In our case, instead of returning the json toy object, just return its url:
<http://localhost:8001/toys/2>
- According to the [HTTP specification](#), this is a complete or relative URI that is returned in the header of the response using the *Location* field.

```
toy = {  
    'id': toyid,  
    'name': data['name'],  
    'descr': descr,  
    "age": data["age"],  
    'price': data['price'],  
    'features': features  
}  
Toys[toyid]= toy  
data = {"id": id}  
response = make_response(data,201)  
response.headers['Location'] = f'/toys/{id}'  
return response
```