

PYTHON PROGRAMMING AND MACHINE LEARNING

RESTFUL WEB SERVICE

Yunghans Irawan (yirawan@nus.edu.sg)

Objectives

- Understand the need for distributed computing and service based architecture
- Understand the motivation for microservices architecture
- Able to develop REST service using Flask
- Able to design a good REST service

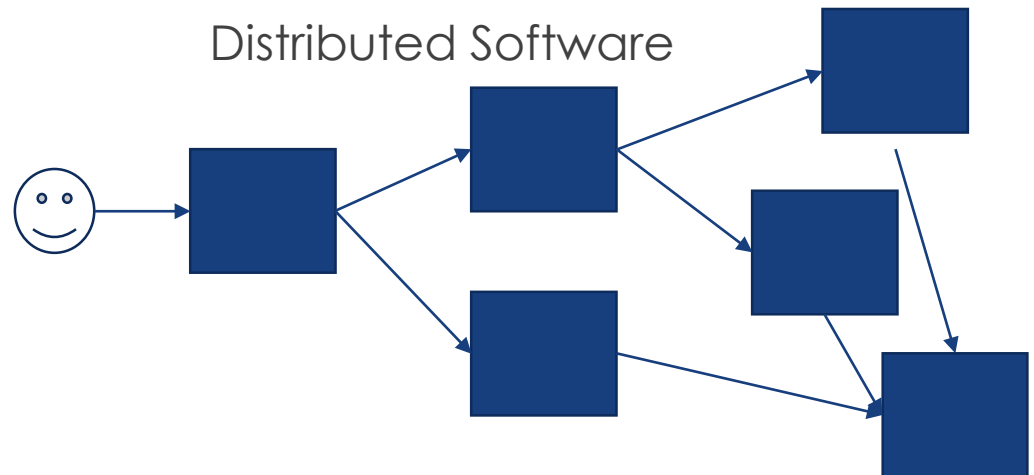
Problem of Distributed Computing

- One of the source of complexity in today's software is because our software is distributed
 - Components of the software are located in different machines

Standalone Software



Distributed Software



Example

- Telco's Mobile App
 - The mobile app would need to check with various backend systems for
 - Checking your contract status
 - Starting your contract renewal process
 - Start or terminate new value added services
 - The system that handles renewal process would also need to communicate with
 - System that manage your contract status
 - System that manage phone inventory to reserve the phone
 - System that manage delivery
 - And so on. You can imagine the complexity

The necessity for distributed design

- Why can't we just put everything in a single program?

History of distributed communication

- Programs must be within the same machine
- Programs can be deployed on different machines but the program must be using the same language and same operating system
- Programs must be built using the same language
- Programs deployed on multiple platform and languages can communicate

Challenges of distributed design

- Remote communication is slow
 - We also use the term "expensive" because it costs the program time to wait for the response
- Different language has different internal representation variables and data type differently
 - String in C and string in C# are handled differently
 - Integer in PC and integer in mainframe has different binary pattern

Challenges of distributed design

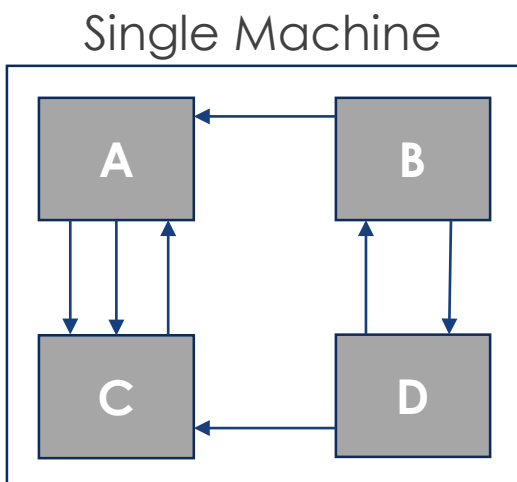
- Network is not reliable
 - Chance of packet loss and dropped connection
- Machines can crash and become unavailable
 - The system has to handle situation when it's only partially available

Some Solutions (today)

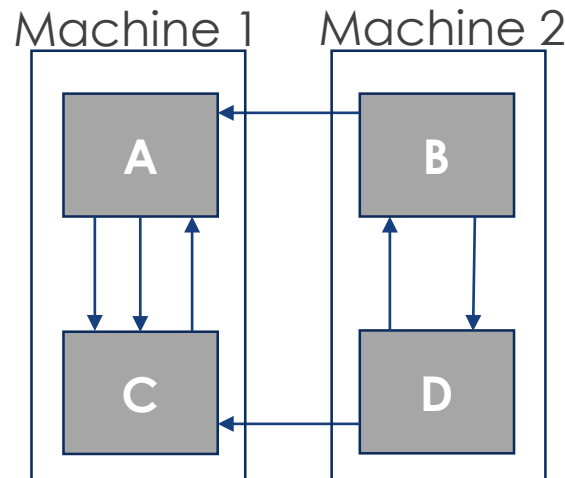
- Good modular decomposition
- Platform agnostic communication protocols
- Distributed consensus algorithms
- High availability solutions

Good Modular Decomposition

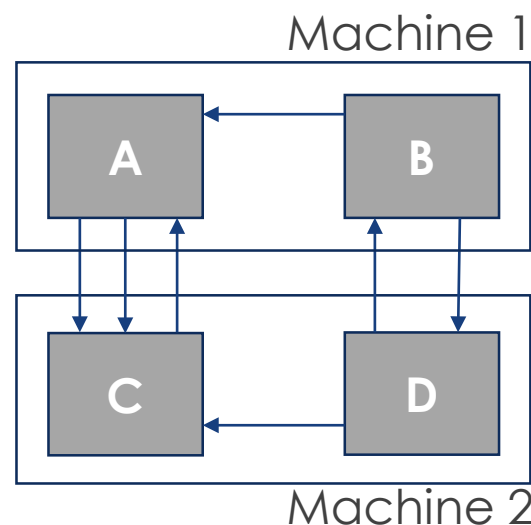
Original Setup



1st Setup

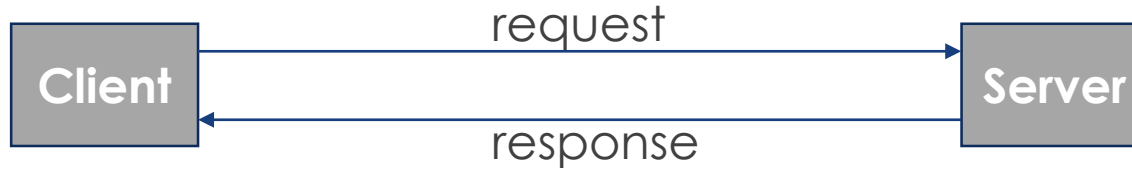


2nd Setup



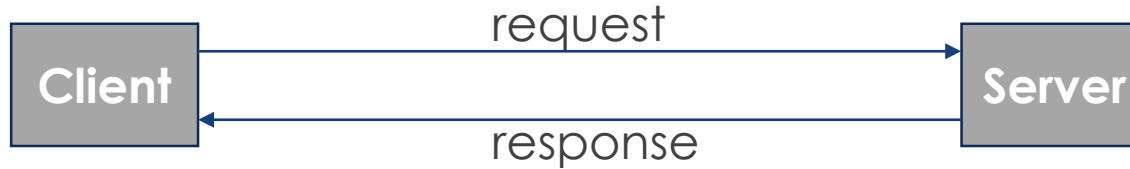
What's the difference between
1st setup vs. 2nd setup?

Platform Agnostic Communication Protocols



- Platform agnostic means that the client and the server can be built/deployed using different
 - Programming languages
 - Programming frameworks
 - Operating systems

Platform Agnostic Communication Protocols Example



GET /articles?include=author HTTP/1.1

HTTP/1.1 200 OK

Content-Type: application/vnd.api+json

```
{
  "data": [{
    "type": "articles",
    "id": "1",
    "attributes": {
      "title": "JSON:API paints my bikeshed!",
      "body": "The shortest article. Ever.",
      "created": "2015-05-22T14:56:29.000Z",
      "updated": "2015-05-22T14:56:28.000Z"
    },
    "relationships": {
      "author": {
        "data": {"id": "42", "type": "people"}
      }
    }
  }]
}
```

Platform Agnostic Communication Protocols Example

- How does the previous example achieve its agnostic-ness?
 - Protocol is based on text
 - All platforms can process text as long as we agree on the structure and the meaning of the text
- Can binary protocol be platform agnostic?
 - Yes, as long as there is an agreement
- The example that we use is called as REST (a.k.a. REST API, REST architectural style, RESTful web service)
 - This is what we are going to learn to build

Distributed consensus algorithms

- Algorithms used to resolve state consistency across distributed system
- Example:
 - We are playing an online game. In my device, I shot your character and your character died, however in your device, it's the opposite. Who win?
 - We deploy a distributed database with multiple replicas (copies) of data on different machines. Clients are allowed to contact any machines to read and update data. What will happen if multiple clients try to update the same record at the same time?
- We are not going to discuss the algorithms, but those interested can search for more information online.

High Availability

- How to ensure that your service/application is available 24/7?
 - The solution is to provide redundancy.
- Our systems is made of many components
 - Then provide redundancies for each of the components

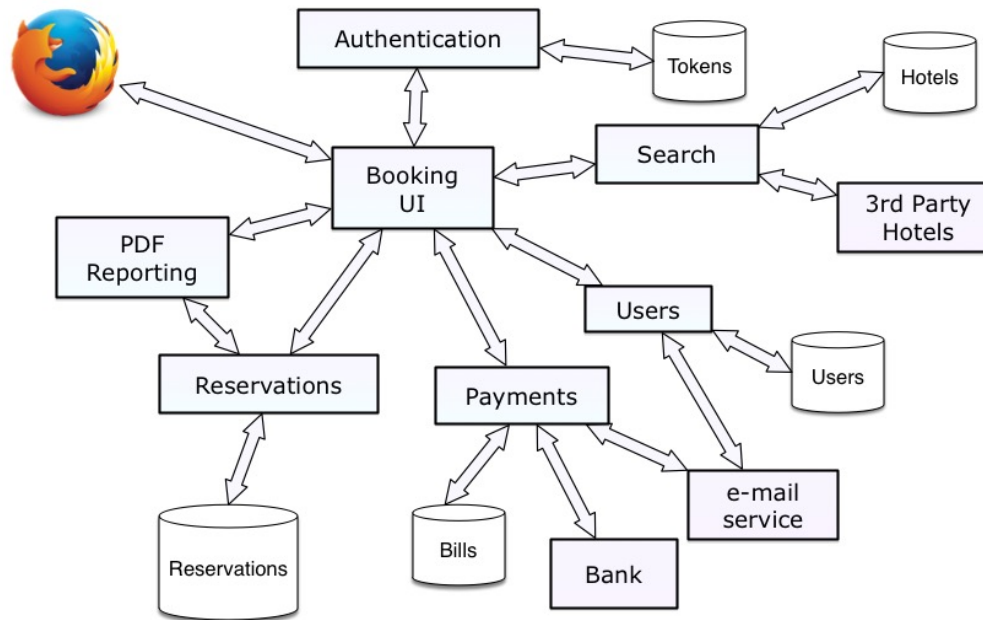
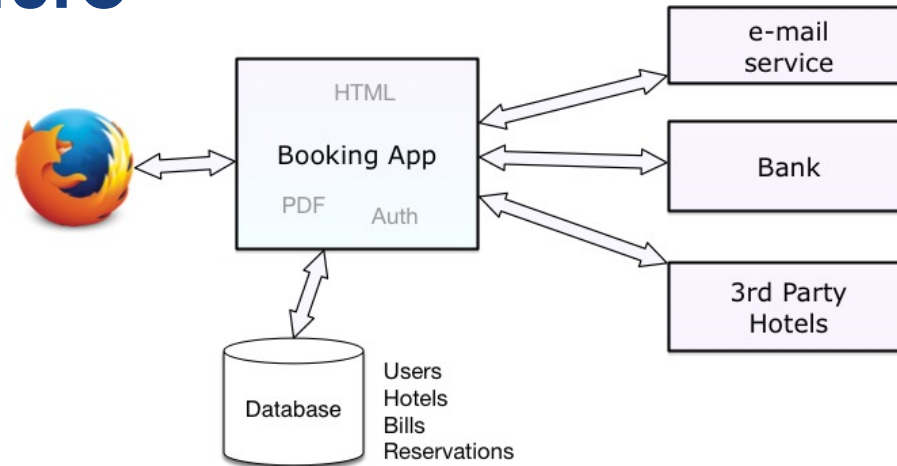
High Availability Example

- For a class to run, we need many components:
 - Room
 - Projector
 - PC
 - Lecturers
 - Cleaners (to clean the class before and after class)
 - Students
 - Let's exclude the students since if there's no students, there's no need to have the class.
- We want to make all of these redundant. Which components are easier/harder to make redundant. More importantly, why?

Microservice Architecture

- Architectural style that split
a monolithic application (a single
application that is in charge of everything)
into
several different microservices component
that runs in different processes

Monolithic vs. Microservices Architecture



Monolithic Application

- Starting a project as a monolith is easy, and probably the best approach.
- A centralized database simplifies the design and organization of the data.
- Deploying one application is simple.
- Any change in the code can impact unrelated features. When something breaks, the whole application may break.
- Solutions to scale your application are limited: you can deploy several instances, but if one particular feature inside the app takes all the resources, it impacts everything.
- As the code base grows, it's hard to keep it clean and under control.

Microservice Definition

- *A microservice is a lightweight application, which provides a narrowed list of features with a well-defined contract. It's a component with a single responsibility, which can be developed and deployed independently.*

From: Python Microservices Development Book

Microservice Architecture

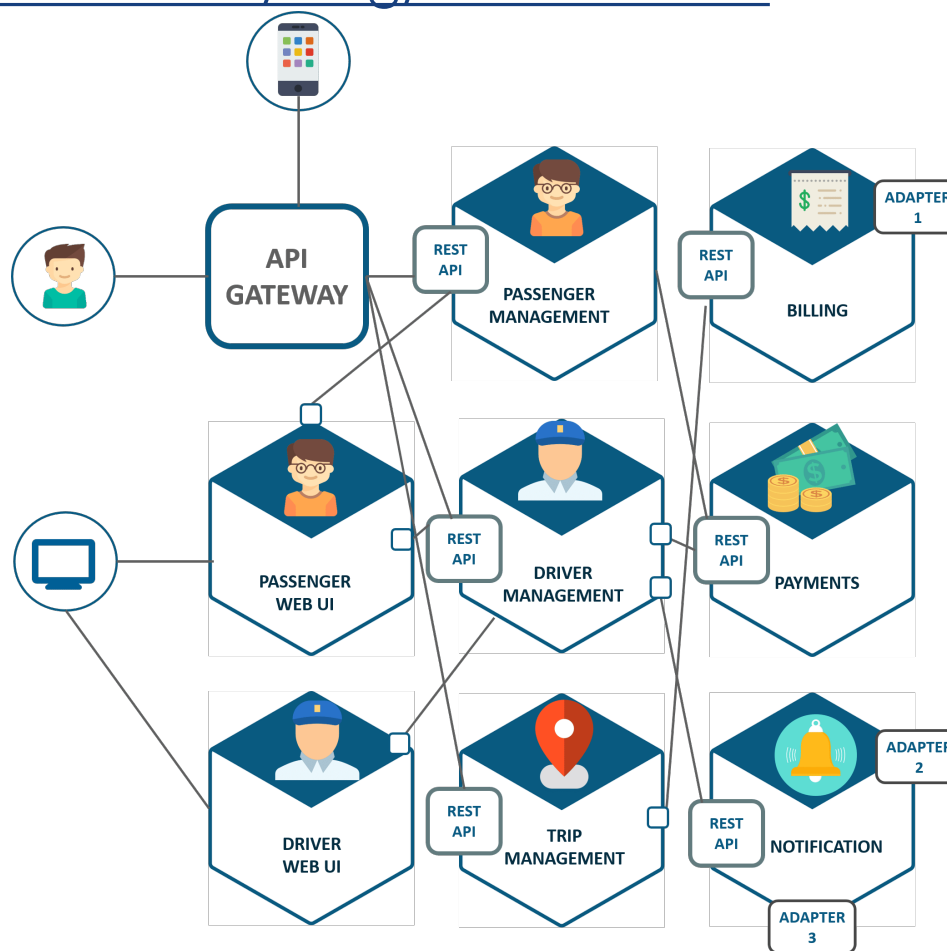
- Separation of concerns
 - Each microservices can be developed independently by different teams with its own database
 - Encourage loose coupling between components
- Smaller projects to deal with
 - Each component is a smaller project with less complexity, faster deployment, less complexity and can be developed with different technology
- More scaling and deployment options
 - Different components can be deployed in different machines with different characteristics (more RAM for memory intensive services, more CPU for CPU intensive services)
 - Multiple databases allow higher total throughput (database access/sec) and bigger maximum storage capacity

Microservice vs. Service

- Is microservice = small service and there should be something called a macroservice?
 - No. The focus is in the distributed mindset instead of a monolithic mindset.
- The way a service is exposed is still the same whether an application is designed in a monolithic way or a microservices way
 - Although a monolithic application may require less services to be exposed
- Currently REST is a popular standard to expose a service
 - We are going to learn how to implement a service in Python

Example of Microservice

- Microservice Architecture of Uber
 - According to <https://www.edureka.co/blog/microservice-architecture/>



- Flask is one of the popular web **microframework** for Python
- microframework = minimalistic web application framework
 - keep the core simple but extensible
 - won't make many decisions for you, such as what database/framework to use
- Different philosophy compared to Django, another very popular Python web framework

Writing a simple service

app.py

```
from flask import Flask, request

app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello world!"

if __name__ == '__main__':
    app.run(debug=True)
```

Run from command prompt/terminal

```
python app.py
```

Flask is the main class that we use

First parameter represent the name of the module if we use multi packages in Flask. In a single module, it's not so important

This code is only executed once, when we run "python app.py" and not when this file is imported
Very common python code in many frameworks

Testing a service using Postman

CREATE A NEW COLLECTION

×

Name

sa48

Description

Authorization

Pre-request Scripts

Tests

Variables

This description will show in your collection's documentation, along with the descriptions of its folders and requests.

Adding a description makes your docs better

Descriptions support Markdown

Cancel

Create

Testing a service using Postman

SAVE REQUEST

×

Requests in Postman are saved in collections (a group of requests).
[Learn more about creating collections](#)

Request name

/

Request description (Optional)

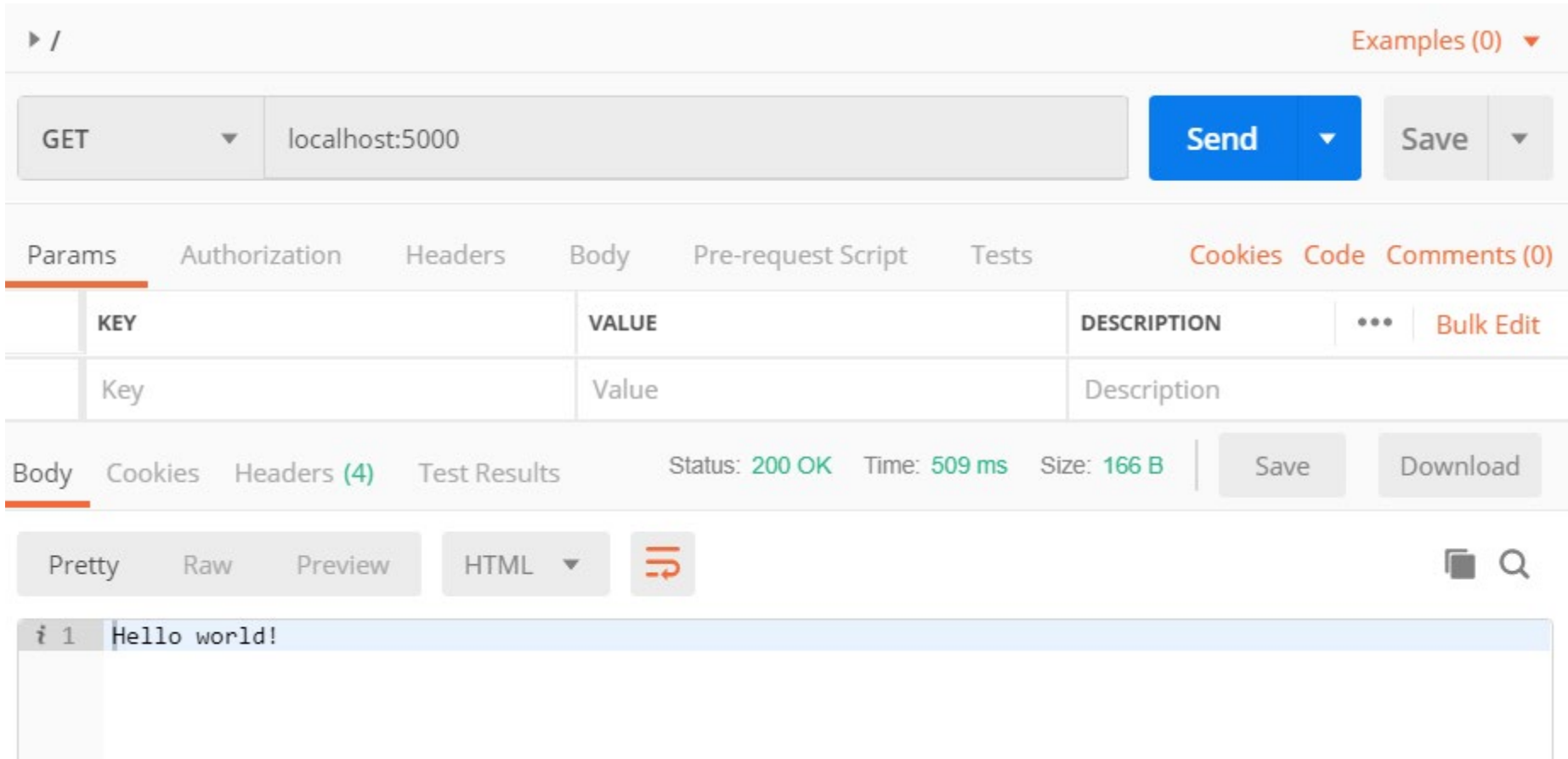
Adding a description makes your docs better

Descriptions support Markdown

Cancel

Save to sa48

Testing a service using Postman



Postman interface showing a GET request to localhost:5000. The response is 200 OK, with a time of 509 ms and a size of 166 B. The response body is "Hello world!".

Request: GET localhost:5000

Params:

KEY	VALUE	DESCRIPTION
Key	Value	Description

Response: Status: 200 OK, Time: 509 ms, Size: 166 B

Body: Hello world!

Writing a simple service

app.py

```
from flask import Flask, request

app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello world!"

if __name__ == '__main__':
    app.run(debug=True)
```

Run from command prompt/terminal

```
python app.py
```

We bind the root URL "/" to hello() function.

The return type must be a string,
a tuple of (response, status code),
a Response object or
a WSGI callable (need to understand WSGI, etc)

Can be used to return
HTML for web
application, or
JSON/XML/text for web
services

Writing a simple service

app.py

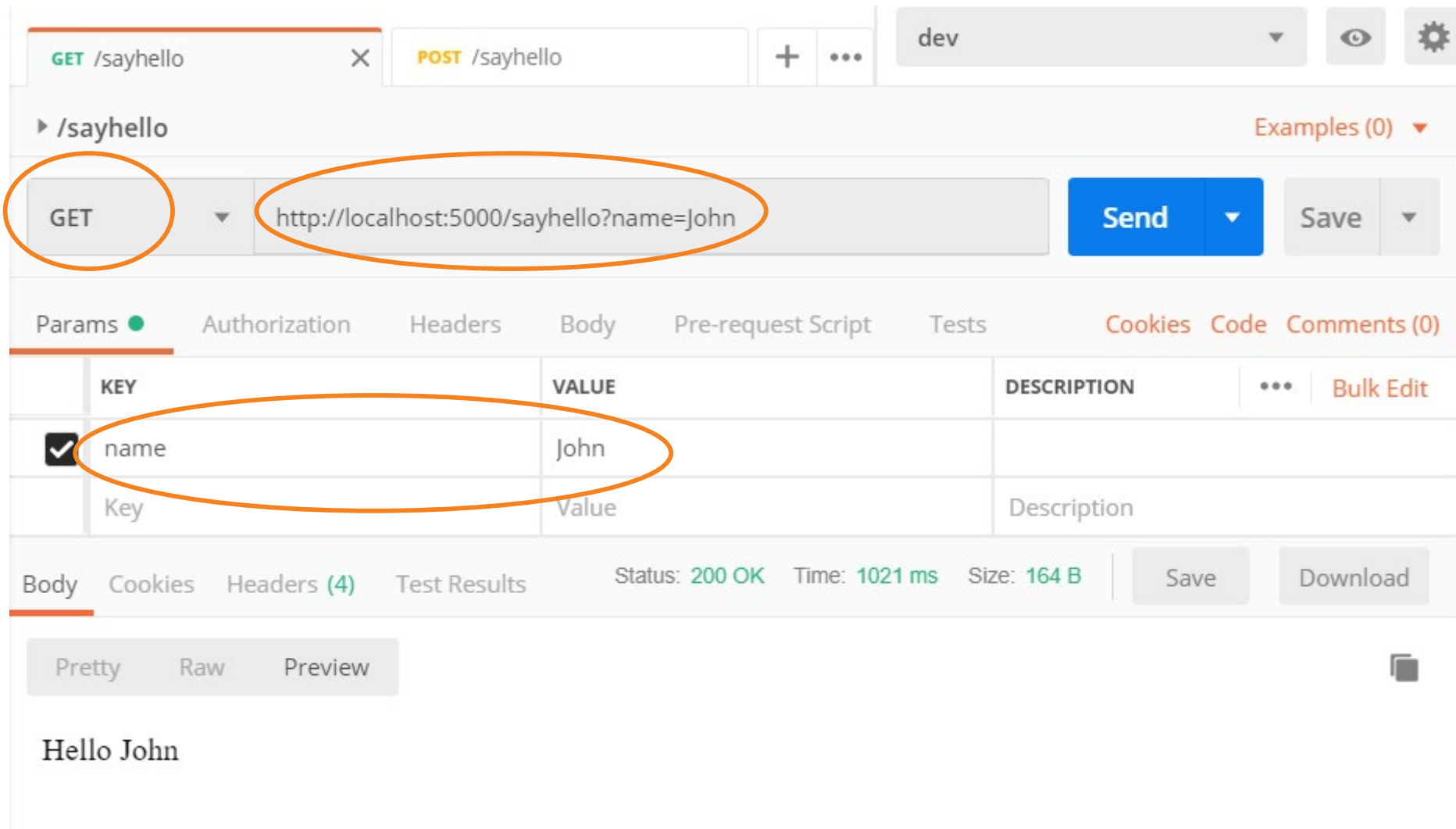
```
@app.route('/sayhello/', methods=['GET', 'POST'])
def say_hello():
    name = request.args.get('name') or request.form.get('name')
    return "Hello " + str(name or '')
```

We bind the URL `"/sayhello"` to `say_hello()` function.

We get the name either from the URL's query string or from the form submitted using POST method – therefore we use the 'or' trick here

We return a simple text

Sample GET Request (Postman)



The image shows the Postman interface for a GET request. The request URL is `http://localhost:5000/sayhello?name=John`. The request is successful with a status of `200 OK`, a time of `1021 ms`, and a size of `164 B`. The response body is `Hello John`.

Request Details:

- Method: GET
- URL: `http://localhost:5000/sayhello?name=John`

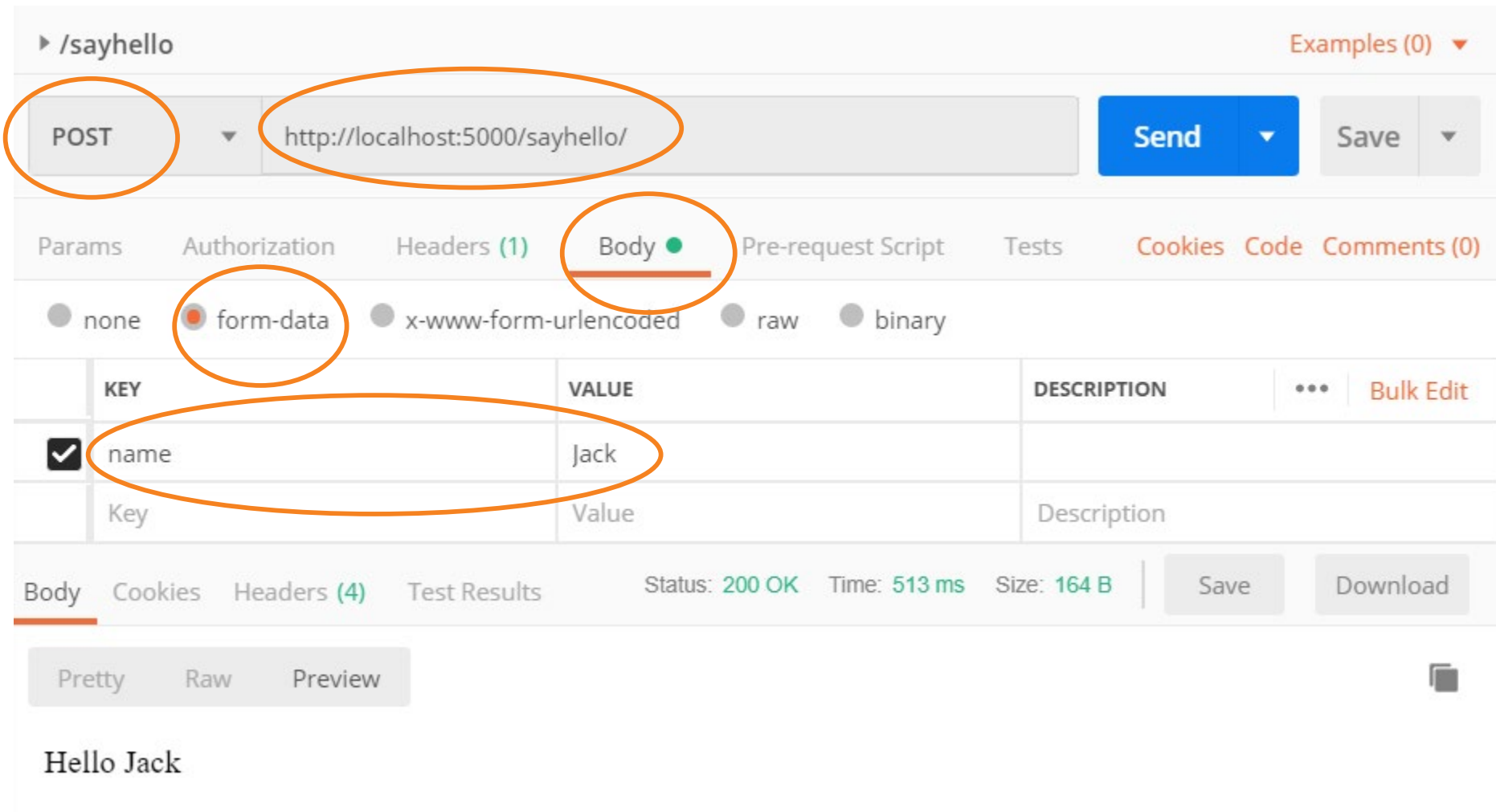
Params:

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> name	John	
Key	Value	Description

Response:

Body: Hello John

Sample POST Request (Postman)



► /sayhello Examples (0) ▼

POST Send ▼ Save ▼

Params Authorization Headers (1) **Body ●** Pre-request Script Tests Cookies Code Comments (0)

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

	KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input checked="" type="checkbox"/>	name	Jack			
	Key	Value	Description		

Body Cookies Headers (4) Test Results Status: 200 OK Time: 513 ms Size: 164 B Save Download

Pretty Raw Preview

Hello Jack

Writing a simple service

app.py

```
@app.route("/greet/",  
           defaults={'name': 'nobody'},  
           methods=['GET', 'POST'])  
@app.route("/greet/<name>",  
           methods=['GET', 'POST'])  
def greet(name):  
    return "Good Morning " + name
```

Instead of passing the parameter through query string/form, parameter can be passed through the URL

We can define multiple patterns and define the default value. In the above case, the value of name will be 'nobody' if it's not supplied

Writing a simple service

app.py

```
@app.route("/greet/",
           defaults={'name': 'nobody'},
           methods=['GET', 'POST'])
@app.route("/greet/<name>",
           methods=['GET', 'POST'])
def greet(name):
    return "Good Morning " + name
```

We can also specify which HTTP methods is supported by the function

► /greet Examples (0) ▼

GET ▼ localhost:5000/greet Send ▼ Save ▼

Params Authorization Headers Body Pre-request Script Tests Cookies Code Comments (0)

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (4) Test Results Status: 200 OK Time: 1042 ms Size: 173 B | Save Download

Pretty Raw Preview HTML ▼ ↺

Good Morning nobody




► /greet Examples (0) ▼

GET ▼ localhost:5000/greet/John Send ▼ Save ▼

Params Authorization Headers Body Pre-request Script Tests Cookies Code Comments (0)

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (4) Test Results Status: 200 OK Time: 507 ms Size: 171 B | Save Download

Pretty Raw Preview HTML ▼   

i 1 Good Morning John

Splitting over multiple files

```
from flask import request
```

handler1.py

```
def hello():  
    return "Hello world!"
```

```
def say_hello():  
    name = request.args.get('name') or  
request.form.get('name')  
    return "Hello " + str(name or '')
```

```
def greet(name):  
    return "Good Morning " + name
```

handler2.py

Splitting over multiple files

```
from flask import Flask,request
import handler1
import handler2
```

```
app = Flask(__name__)
```

```
app.add_url_rule('/',view_func=handler1.hello)
```

```
app.add_url_rule('/sayhello/',
    view_func=handler1.say_hello,
    methods=['GET','POST'])
```

```
app.add_url_rule('/greet/',
    view_func=handler2.greet,
    defaults={'name': 'nobody'},
    methods=['GET','POST'])
```

```
app.add_url_rule('/greet/<name>',
    view_func=handler2.greet,
    methods=['GET','POST'])
```

```
if __name__ == '__main__':
    app.run(debug=True)
```

Comparing single vs multiple

```
app.add_url_rule('/',view_func=handler1.hello)
```

```
app.add_url_rule('/sayhello/',  
view_func=handler1.say_hello,  
methods=['GET','POST'])
```

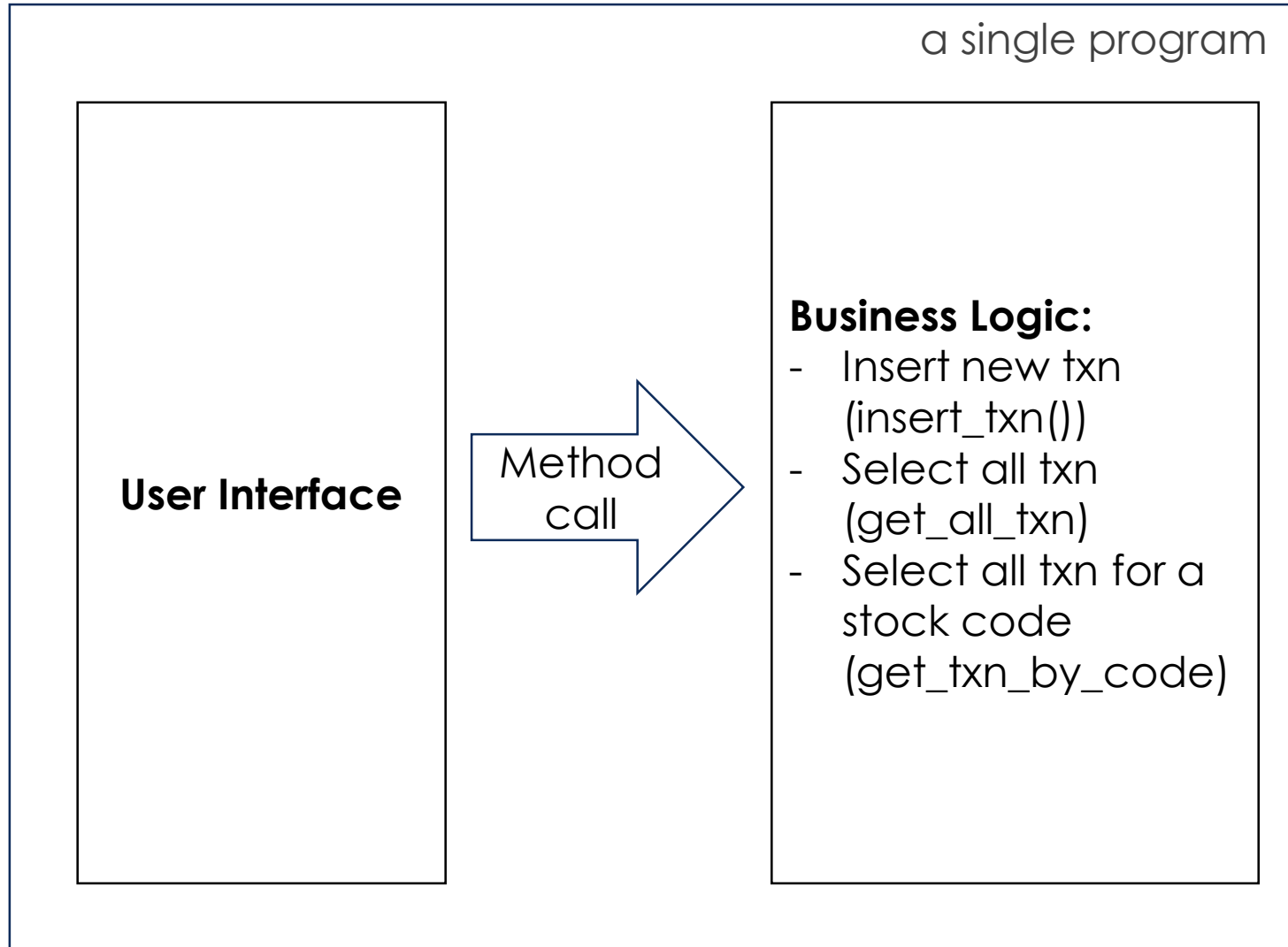
```
app.add_url_rule('/greet/',  
view_func=handler2.greet,  
defaults={'name': 'nobody'},  
methods=['GET','POST'])
```

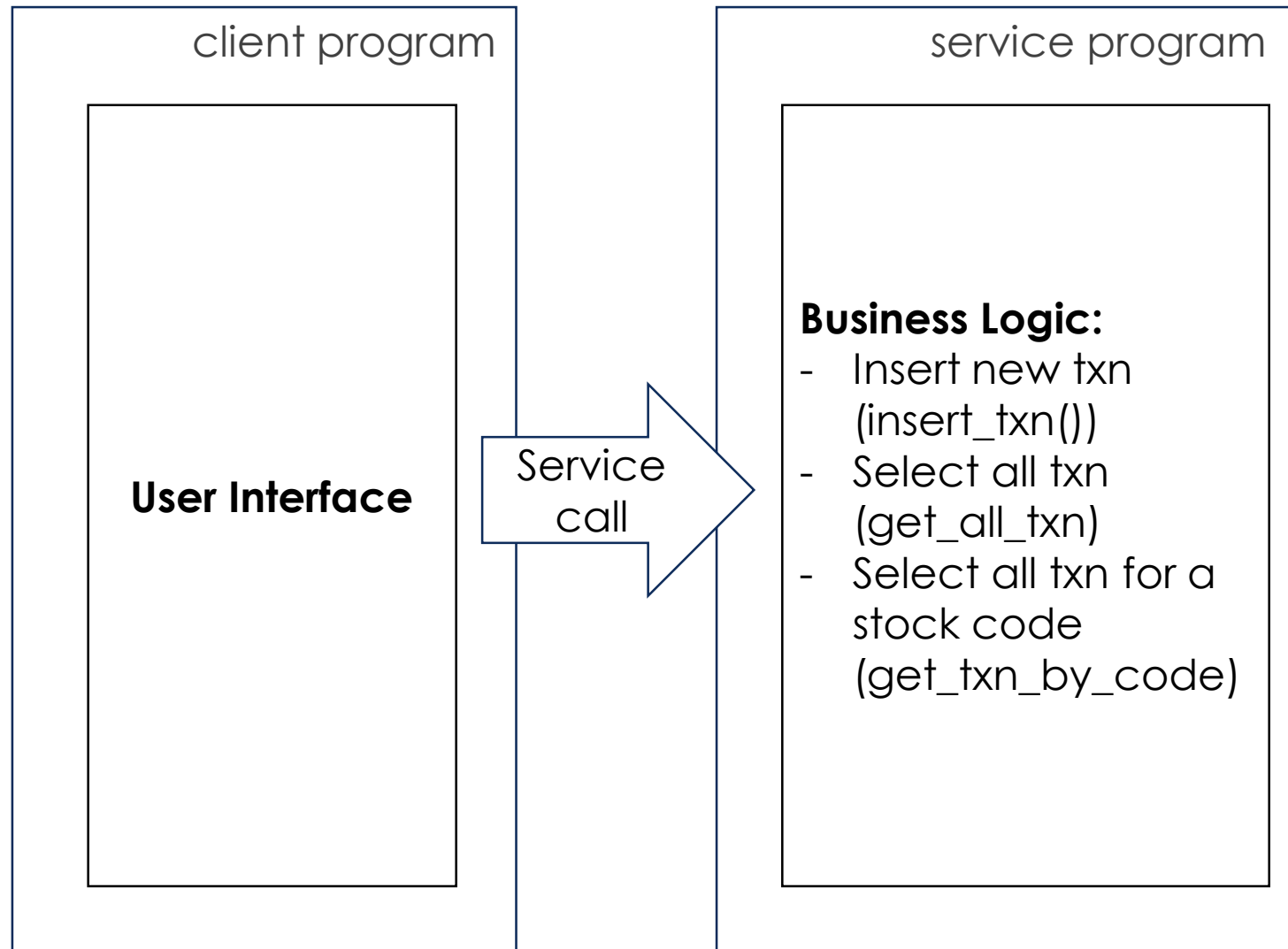
```
app.add_url_rule('/greet/<name>/',  
view_func=handler2.greet,  
methods=['GET','POST'])
```

```
@app.route("/")  
def hello():  
    return "Hello world!"
```

```
@app.route("/greet/",  
defaults={'name': 'nobody'},  
methods=['GET','POST'])  
@app.route("/greet/<name>",  
methods=['GET','POST'])  
def greet(name):  
    return "Good Morning " + name
```

```
@app.route('/sayhello/', methods=['GET','POST'])  
def say_hello():  
    name = request.args.get('name') or  
    request.form.get('name')  
    return "Hello " + str(name or '')
```





Service Interface Design (option 1)

Method	POST
URI	/insert_txn

Method	GET
URI	/get_all_txn

Method	GET
URI	/get_txn_by_code

Service Interface Design (option 1)

Method	POST
URI	/insert_txn
Sample URL	/insert_txn
Sample JSON request	<pre>{ "date": "2006-03-28", "price": 45.0, "qty": 1000.0, "symbol": "IBM", "trans": "BUY" }</pre>
Sample Response Code	200
Sample JSON response	<pre>{ "status": "ok" }</pre>

Service Interface Design (option 2)

Method	POST
URI	/txn/

Method	GET
URI	/txn

Method	GET
URI	/txn/<code>

Service Interface Design (option 2)

Method	GET
URI	/txn/<code>
Sample URL	/txn/IBM
Sample JSON request	N/A
Sample Response Code	200

Service Interface Design (option 2)

Sample
JSON
response

```
[
  {
    "date": "2006-03-28",
    "price": 45.0,
    "qty": 1000.0,
    "symbol": "IBM",
    "trans": "BUY"
  },
  {
    "date": "2006-04-06",
    "price": 53.0,
    "qty": 500.0,
    "symbol": "IBM",
    "trans": "SELL"
  }
]
```

Option 1 vs. Option 2

Which one is better?

Method	POST
URI	/insert_txn

Method	GET
URI	/get_all_txn

Method	GET
URI	/get_txn_by_code

Remote Procedure Call
(RPC) style

Method	POST
URI	/txn/

Method	GET
URI	/txn

Method	GET
URI	/txn/<code>

REpresentational State Transfer
(REST) style

API Design Best practices – 1

Nouns for URLs

- Easy to read and work with
- Hard to misuse
- Complete and concise

Well-designed API

- Improves developer experience
- Faster documentation
- Higher adoption for the APIs

- Use Nouns to describe URLs
- Describe resource functionality with HTTP methods
- Give meaningful feedback in HTTP Responses to help developers succeed
- Handle complexities in HTTP Requests elegantly

API Design Best practices – 2

Resource-Oriented

Resource Oriented Design

- The Goal is to develop APIs that are simple, consistent and easy to use
- Recently, most APIs are built as HTTP REST APIs
 - The core principle is to work with ‘named resources’ (simple or collection)
 - The resources and methods are known as nouns and verbs of APIs.
 - Methods usually map to HTTP methods POST, GET, PUT, PATCH and DELETE

API Design Best practices – 3

HTTP methods

- Describe resource functionality with HTTP methods

Method	Description
GET	Used to retrieve a representation of a resource.
POST	Used to create new resources and sub-resources
PUT	Used to update existing resources
PATCH	Used to update existing resources
DELETE	Used to delete existing resources

API Design Best practices – 4

Idempotent requests

Idempotent Requests

Method	Description
GET	Should not impose any side effects. Repeated requests on the same resource should result in the same state.
POST	Used to create new resources and sub-resources, should not have any unrelated side-effects.
PUT	Should not impose any side effects. Repeated requests on the same resource should result in the same state.
PATCH	Should not impose any side effects. Repeated requests on the same resource should result in the same state.
DELETE	Should not impose any side effects. Repeated requests on the same resource should result in the same state. First request might return status code 204 (no content) Subsequent requests might return status code 404 (not found)

<https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-implementation>

API Design Best practices – 5

Meaningful feedback in responses

- Give meaningful feedback in Responses to help developers succeed
 - The client application behaved erroneously (client error - 4xx response code)
 - The API behaved erroneously (server error - 5xx response code)
 - The client and API worked (success - 2xx response code)
- Give examples of the responses

Well-designed API

- Improves developer experience
- Faster documentation
- Higher adoption for the APIs

API Design Best practices – 5

Example

Users

Method	HTTP request	Description
URIs relative to https://www.googleapis.com/gmail/v1/users , unless otherwise noted		
<code>getProfile</code>	GET <code>/userId/profile</code>	Gets the current user's Gmail profile.
<code>stop</code>	POST <code>/userId/stop</code>	Stop receiving push notifications for the given user mailbox.
<code>watch</code>	POST <code>/userId/watch</code>	Set up or update a push notification watch on the given user mailbox.
https://developers.google.com/gmail/api/v1/reference/		

Exercise – Designing Service Interface

- Design REST service for salesman and customer database
 - Similar to Exercise 2 – Database Access
- Functionalities:
 - Create, update, delete
 - Get salesman by id
 - Get customer by id
 - Get customer by salesman id
 - Get salesman by customer id
 - Get customer by grade

column	type
salesman_id	text primary key
name	text
city	text
commision	real

Column Name	Type
customer_id	text primary key
cust_name	text
city	text
grade	integer
salesman_id	text foreign key

salesman_id	name	city	commission
5001	James Hoog	New York	0.15
5002	Nail Knite	Paris	0.13
5005	Pit Alex	London	0.11
5006	Mc Lyon	Paris	0.14
5003	Lauson Hen		0.12
5007	Paul Adam	Rome	0.13

customer_id	cust_name	city	grade	salesman_id
3002	Nick Rimando	New York	100	5001
3005	Graham Zusi	California	200	5002
3001	Brad Guzan	London	100	5005
3004	Fabian Johns	Paris	300	5006
3007	Brad Davis	New York	200	5001
3009	Geoff Camero	Berlin	100	5003
3008	Julian Green	London	300	5002
3003	Jozy Altidor	Moscow	200	5007

Exercise – Flask Implementation

- Implement your design