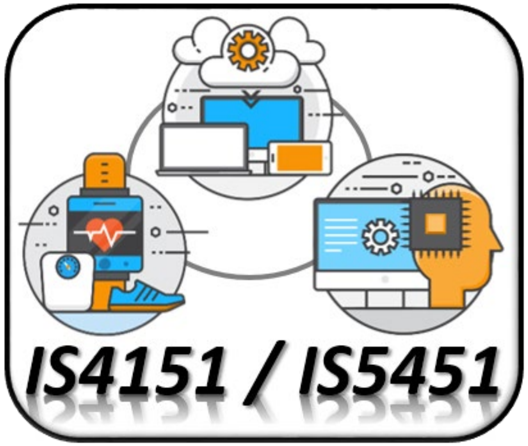


# Lecture 8

# IoT Backend Integration

IS4151/IS5451 – AIoT Solutions and Development
AY 2024/25 Semester 2

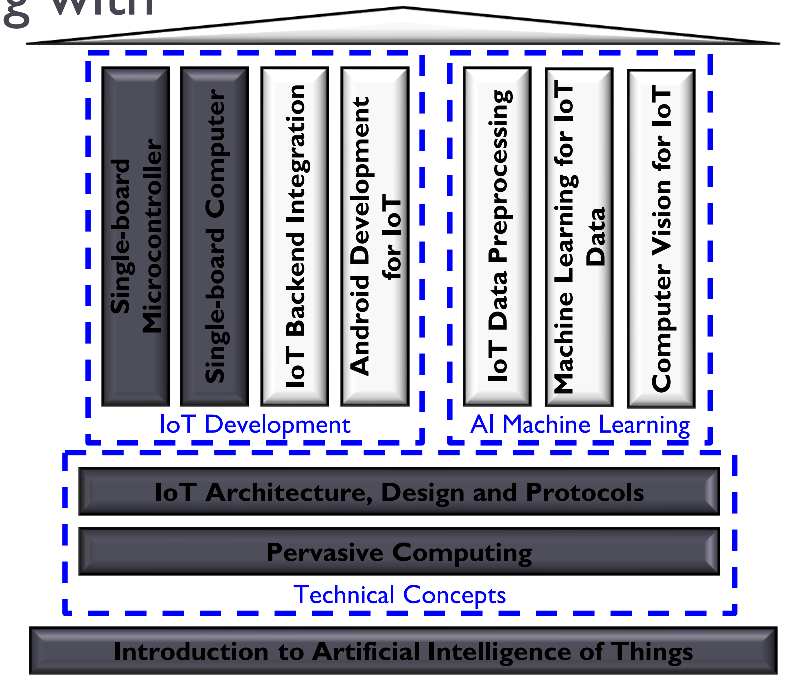
**Lecturer**: A/P TAN Wee Kek

**Email**: tanwk@comp.nus.edu.sg :: **Tel**: 6516 6731 :: **Office**: COM3-02-35

**Consultation**: Tuesday, 2 pm to 4 pm. Additional consultations by appointment are welcome.

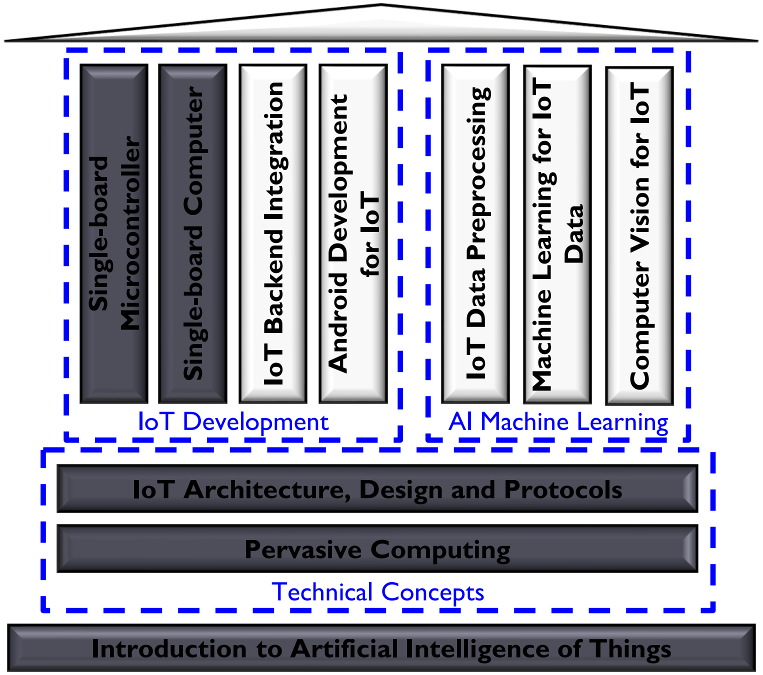
# Quick Recap…

- In the preceding two lectures, we learnt:
  - The technical characteristics of the Raspberry Pi and appreciate its capability to act as both node device and hub plus edge processing.
  - How to perform GPIO programming with the Raspberry PI using both digital and analogue signal.
  - How to control and interact with one or more micro:bit devices via radio and BLE wireless communication.


Single-board Microcontroller
Single-board Computer
IoT Backend Integration
Android Development for IoT
IoT Development
IoT Data Preprocessing
Machine Learning for IoT Data
Computer Vision for IoT
AI Machine Learning
IoT Architecture, Design and Protocols
Pervasive Computing
Technical Concepts
Introduction to Artificial Intelligence of Things

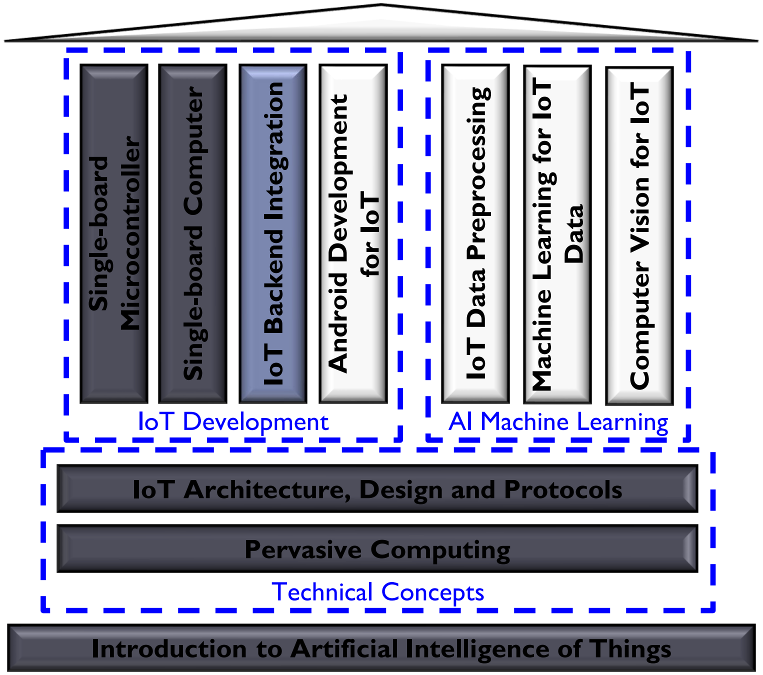
# Quick Recap… (cont.)

- We now have sufficient knowledge to implement the fog computing and cloud computing architecture.
- This lecture continues our learning journey to find out how to:
  - Connect a node device or hub to a fog processor or cloud server.
  - Integrate all three architectures of edge, fog and cloud.


Single-board Microcontroller
Single-board Computer
IoT Backend Integration
Android Development for IoT
IoT Development
IoT Data Preprocessing
Machine Learning for IoT Data
Computer Vision for IoT
AI Machine Learning
IoT Architecture, Design and Protocols
Pervasive Computing
Technical Concepts
Introduction to Artificial Intelligence of Things

# Learning Objectives

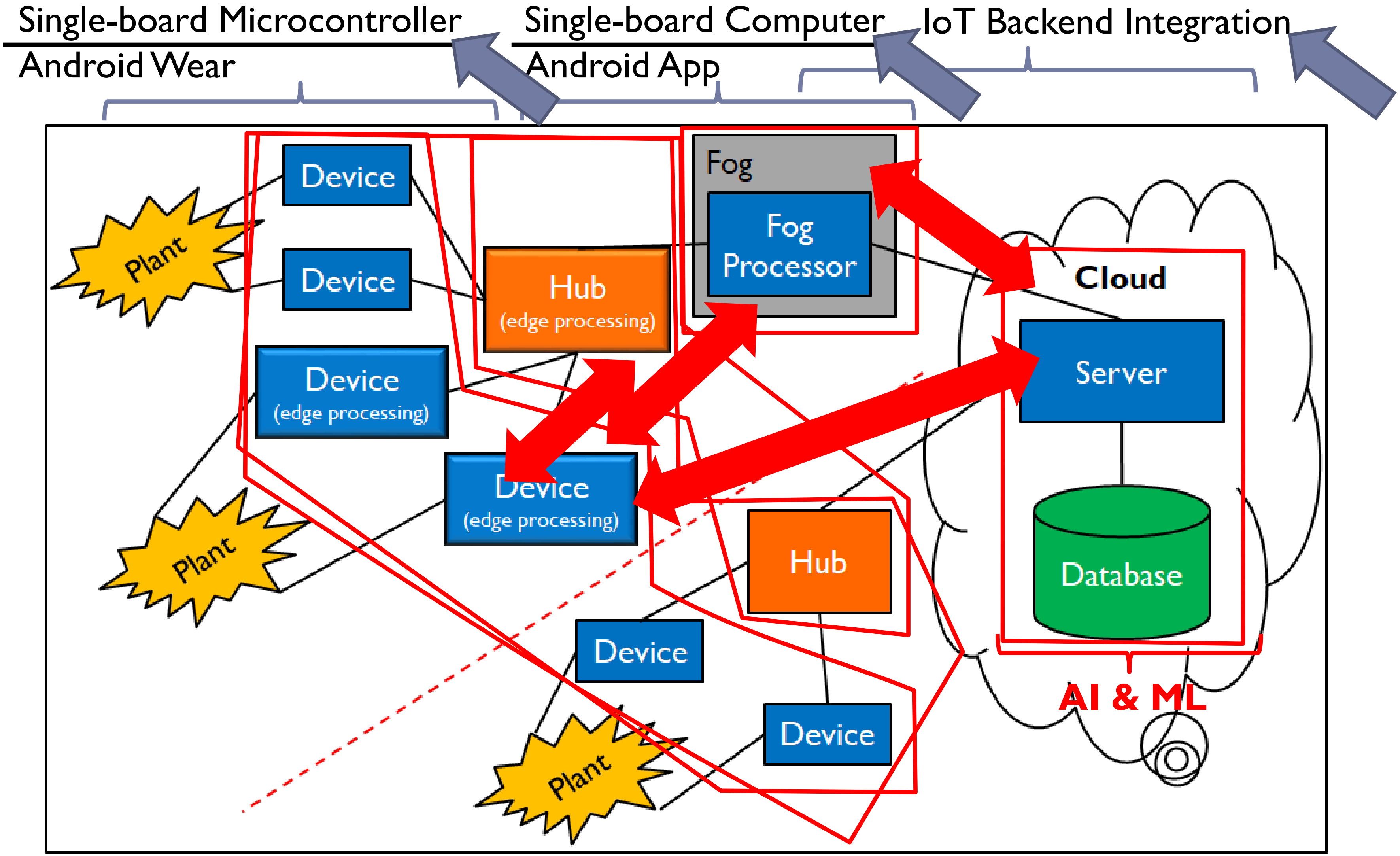
- At the end of this lecture, you should understand:
  - What is Service-Oriented Architecture.
  - What is RESTful web service.
  - How to create RESTful web service in Python with Flask and Connexion.
  - How to test RESTful web service in Postman.
  - How to consume RESTful web service in Python.
  - Persisting the data to a relational database.


Single-board Microcontroller
Single-board Computer
IoT Backend Integration
Android Development for IoT
IoT Data Preprocessing
Machine Learning for IoT Data
Computer Vision for IoT
IoT Development
AI Machine Learning
IoT Architecture, Design and Protocols
Pervasive Computing
Technical Concepts
Introduction to Artificial Intelligence of Things

# Readings

- Required readings:
  - None.
- Suggested readings:
  - None.

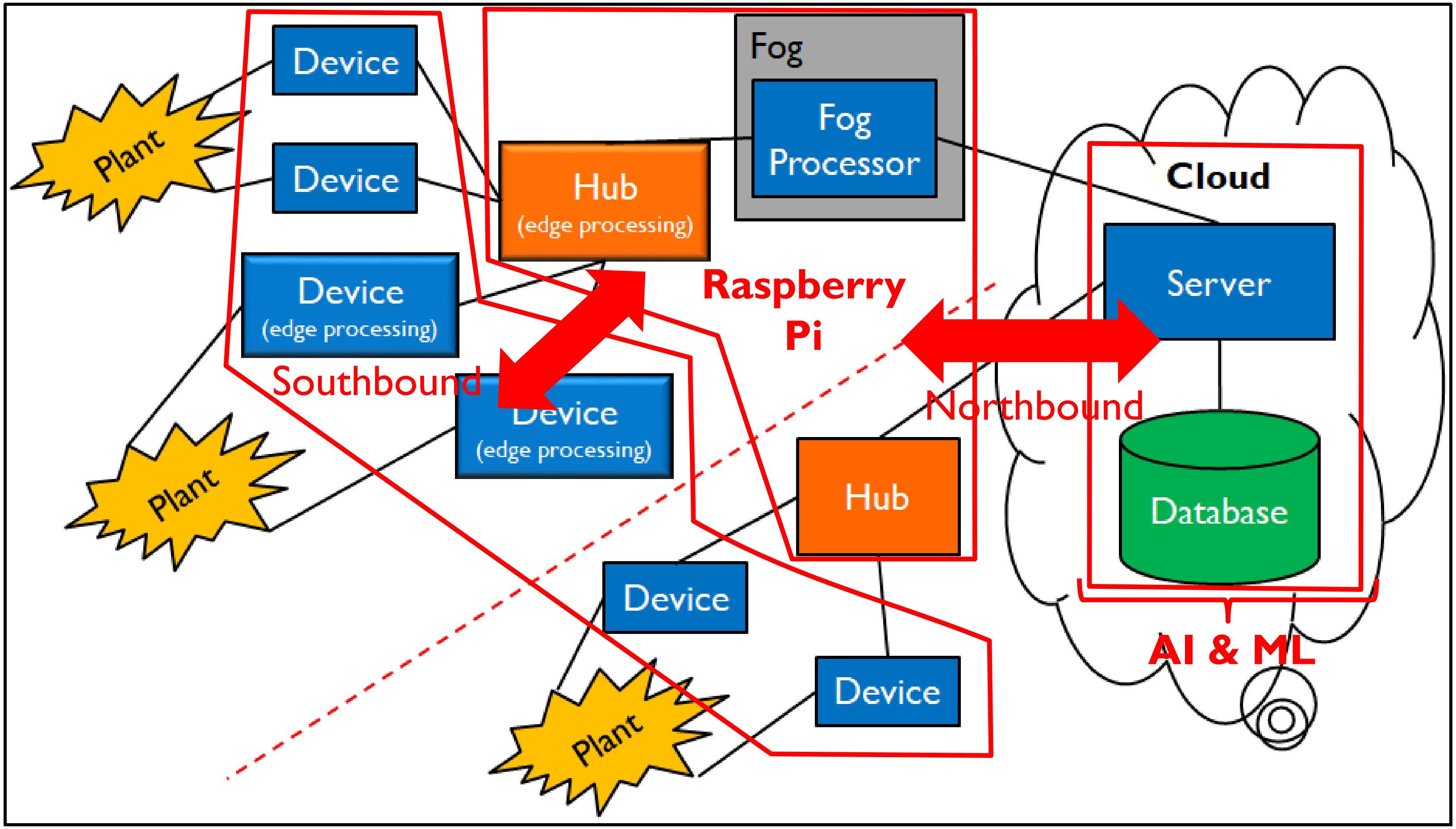
# Technical Roadmap for IS4151/IS5451



# Quick Recap on IoT System Development…

- In a simple IoT system setup, it is not critical to segregate between hub and fog processor.
- For simplicity, we use the Raspberry Pi single-board computer in this course as an integrated hub and fog processor:
  - On the southbound, the Raspberry Pi connects with the micro:bit devices (capable of performing edge processing) to collect sensor values and control their behavior:
    - ***This is where we stop last week.***
  - On the northbound, the Raspberry Pi connects with a cloud server to relay data.
  - At the same time, the Raspberry Pi provides data processing and handles localised information queries.

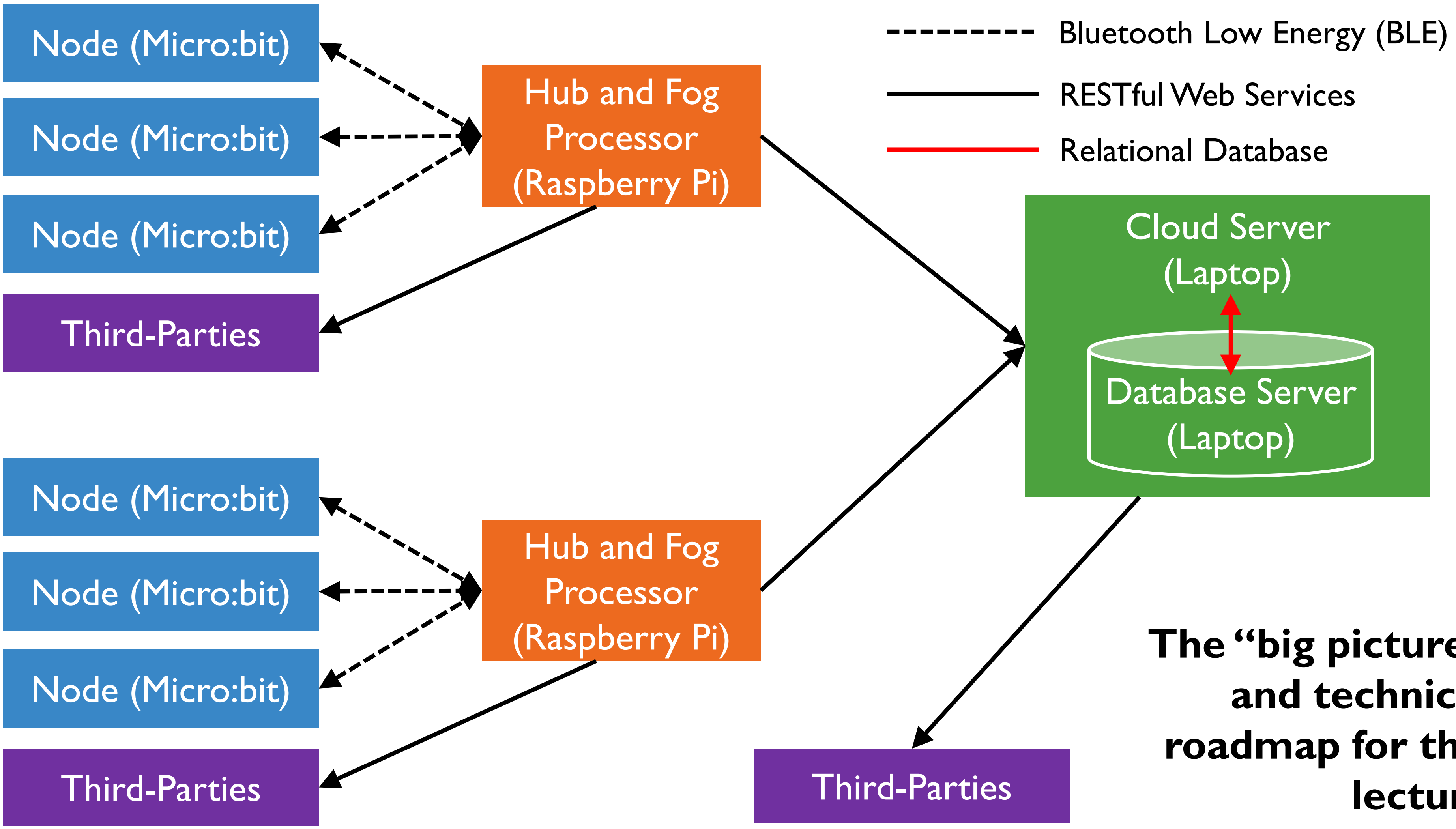
# Quick Recap on IoT System Development…



# In this Lecture…

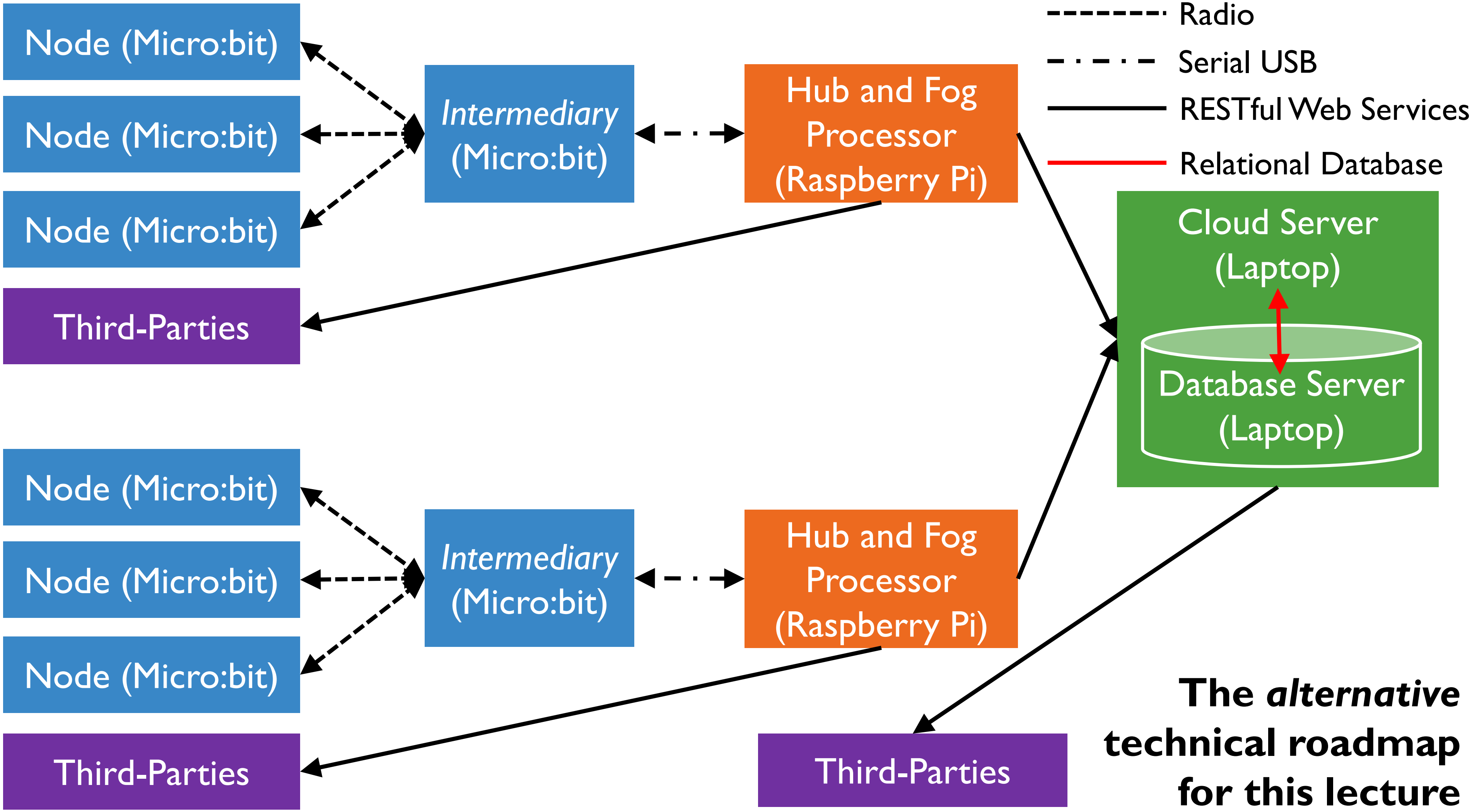
- We continue our journey to learn IoT backend integration using Service-Oriented Architecture SOA.
- On the cloud server:
  - Publishes web services for third parties to consume.
  - Fog processors or hubs can relay data to the cloud server.
  - Other third parties can make global information queries.
- On the fog processor:
  - Consumes web services published by the cloud server to relay data.
  - Also publishes web services for third parties to consume.
  - Hubs can relay data to the fog processor.
  - Other third parties can make localised information queries.

# In this Lecture… (cont.)



**The "big picture" and technical roadmap for this lecture**

# In this Lecture… (cont.)

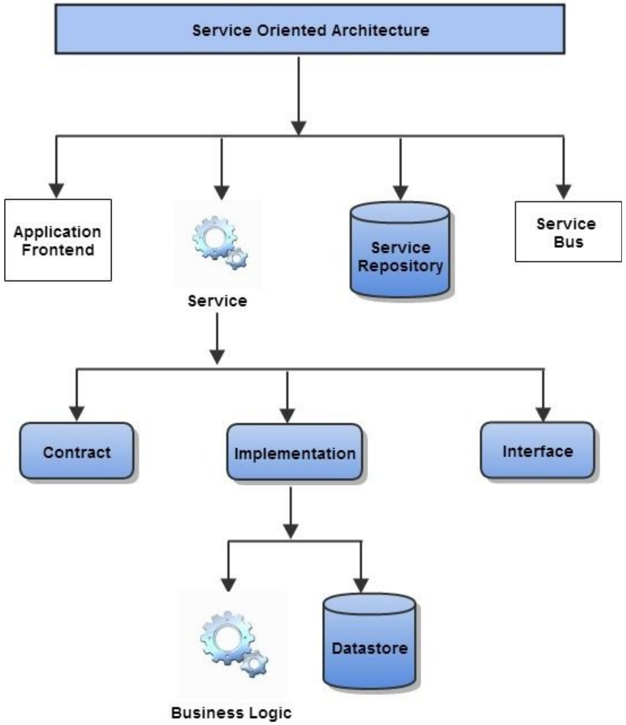


**The *alternative* technical roadmap for this lecture**

# Introduction to Service-Oriented Architecture

# Service-Oriented Architecture

- **SOA** is a software architecture emphasising:
  - Software components providing services to other components by exchanging messages via a standard network communications protocol (e.g., Simple Object Access Protocol or SOAP).
  - In theory, a service is a self-contained unit of functionality, such as retrieving the exchange rate for a currency pair.
  - In practice, a service is an interface definition that may list several discrete services/operations that are semantically related.



# Web Services

- A service following a standard protocol implemented with any technology can be consumed by another software element implemented with any other technologies.
- SOA is typically implemented using the **web services** approach:
  - A web service is a service that is offered by one software element to another via communicating over the World Wide Web (WWW).
  - Machine-to-machine communication is enabled by protocols such as HyperText Transfer Protocol (HTTP).
  - More specifically, messages (e.g., SOAP) are sent and received over HTTP.

# Web Services (cont.)

- Web services can be implemented in two ways:
  - “Big” web services or SOAP web services
  - RESTful web services.

# SOAP Web Services

- Use XML messages that follow the **Simple Object Access Protocol (SOAP)** standard.
- SOAP is an XML language defining a message architecture and message formats.
- Uses a machine-readable description of the operations offered by the service, written in the **Web Services Description Language (WSDL)**.
- WSDL itself is also an XML language but it is used for defining interfaces syntactically.

# RESTful Web Services

- **Representational State Transfer (RESTful)** web services is an alternative to SOAP web services:
  - More suitable for basic, ad hoc integration scenarios.
  - Better integrated with **HTTP** than SOAP-based services.
  - Do not require XML-based SOAP messages or WSDL service-API definitions.
  - Only requires HTTP.
- Desirable characteristics compared to SOAP:
  - Loose coupling.
  - Architecturally simplicity.
  - Ease of consumption on client side.
  - Use standard HTTP methods to manipulate resources.

# RESTful Web Service (cont.)

- RESTful principles:
  - Resource identification through URI.
  - Uniform interface for manipulating resources using the standard HTTP verbs:
    - Create – PUT
    - Read – GET
    - Update – POST
    - Delete – DELETE
  - Self-descriptive messages via decoupled resources that can be represented in any format – XML, JSON, HTML and plain text
  - RESTful request messages are stateless or self-contained.
  - Stateful interactions may be achieved by exchanging state – URI rewriting, cookies, hidden form fields.

# RESTful Web Service (cont.)

- In Python, RESTful web services can be created using:
  - Flask:
    - A web application framework for Python.
  - Connexion:
    - Connexion is a framework on top of Flask that automagically handles HTTP requests defined using OpenAPI (formerly known as Swagger).
    - For building open API using RESTful web services.

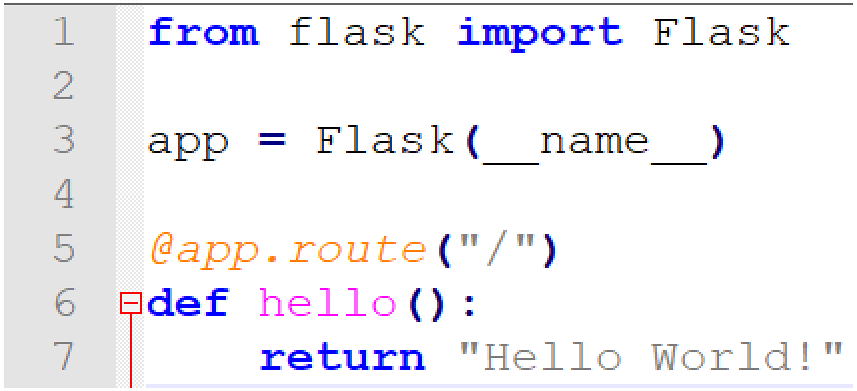
# RESTful Web Services and Relational Databases in Python

# Installing Flask

- Flask can be installed with pip using the following command:

```
python -m pip install flask
```

- To test your Flask installation, create the “Hello World!” Python script below:



```
from flask import Flask

app = Flask(__name__)

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

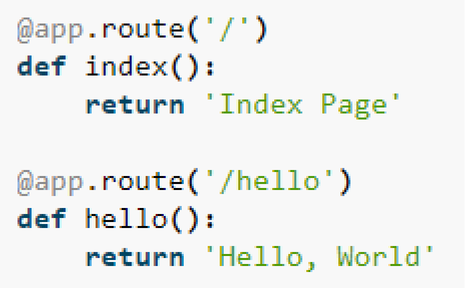
src01.py

- To run the script with Flask:

```
python -m flask --app src01 run
```

# Routing

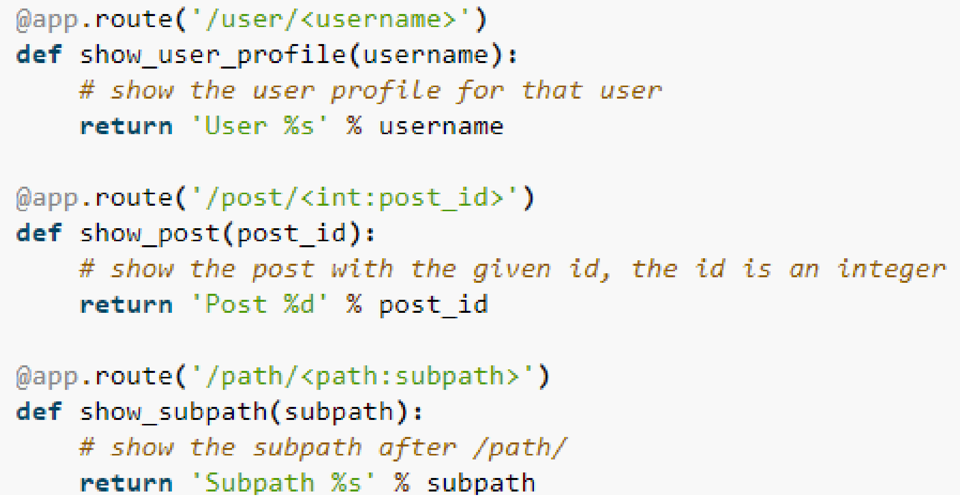
- Modern web applications use meaningful URLs to help users navigate around the websites:
  - Users are more likely to like a page and come back if the page uses a meaningful URL.
  - URL helps user to remember a page and use it to visit a page directly.
- The route() decorator is used to bind a function to a URL:



```
@app.route('/')
def index():
    return 'Index Page'

@app.route('/hello')
def hello():
    return 'Hello, World'
```

# Routing (cont.)

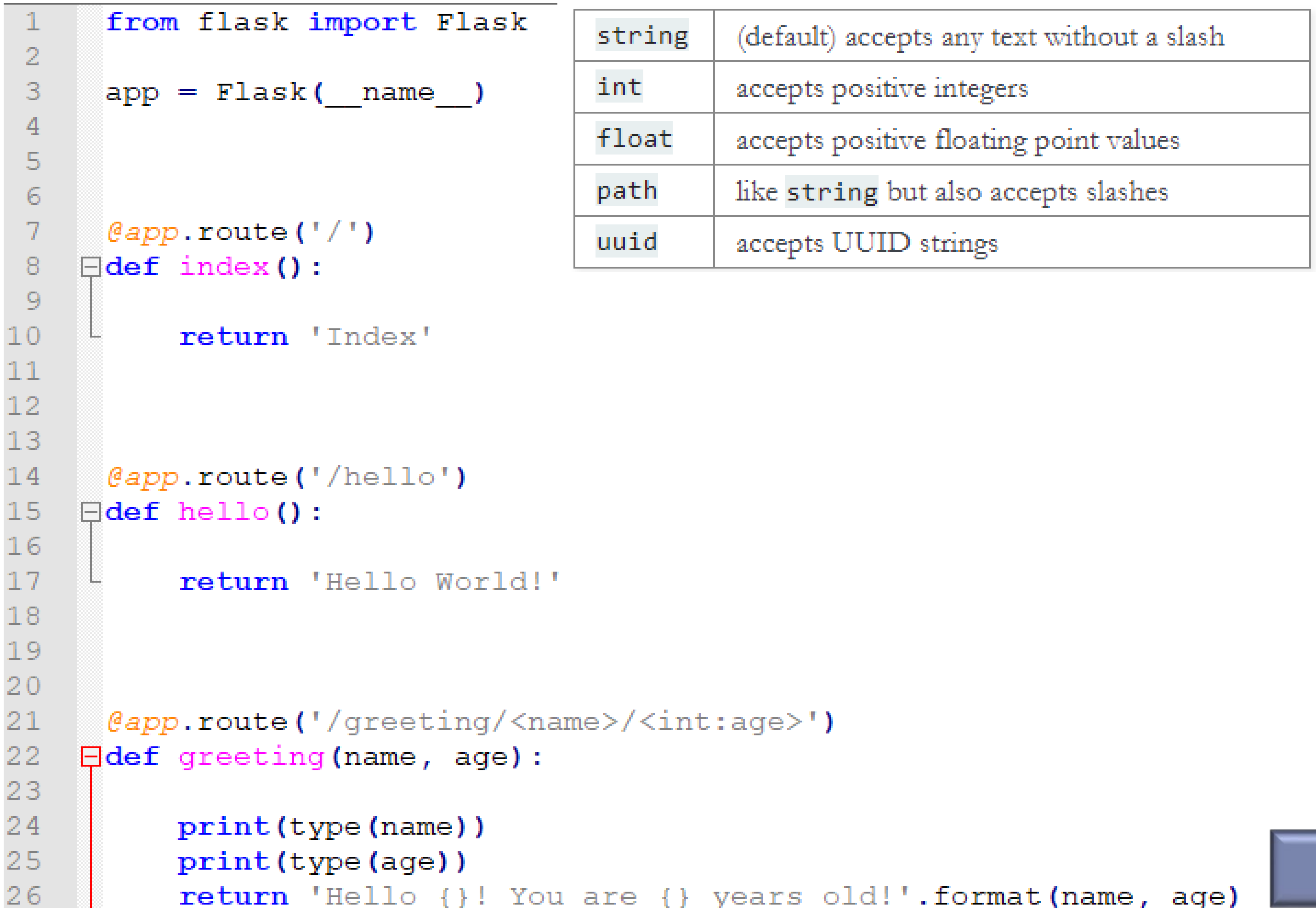
- Variables rules:
  - Add variable sections (a.k.a. path parameters) to a URL by marking sections with the `<variable_name>` notation.
  - Python function then receives the `<variable_name>` as a keyword argument of the same name.
  - Can use an optional converter to specify the type of the argument like `<converter:variable_name>`.



```python
@app.route('/user/<username>')
def show_user_profile(username):
    # show the user profile for that user
    return 'User %s' % username

@app.route('/post/<int:post_id>')
def show_post(post_id):
    # show the post with the given id, the id is an integer
    return 'Post %d' % post_id

@app.route('/path/<path:subpath>')
def show_subpath(subpath):
    # show the subpath after /path/
    return 'Subpath %s' % subpath
```

# Routing (cont.)

- Flask's converters support various primitive types:



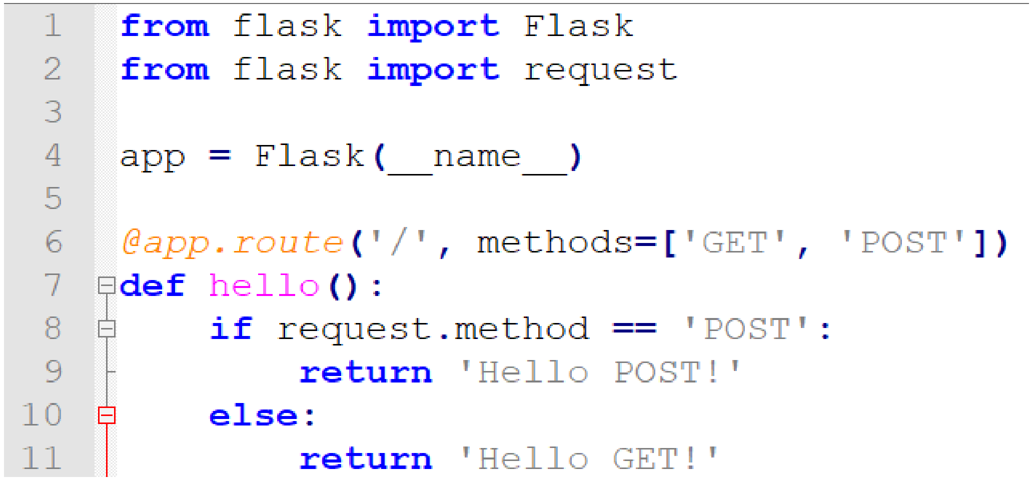
| | |
|---|---|
| `string` | (default) accepts any text without a slash |
| `int` | accepts positive integers |
| `float` | accepts positive floating point values |
| `path` | like `string` but also accepts slashes |
| `uuid` | accepts UUID strings |

```python
from flask import Flask

app = Flask(__name__)



@app.route('/')
def index():

    return 'Index'



@app.route('/hello')
def hello():

    return 'Hello World!'



@app.route('/greeting/<name>/<int:age>')
def greeting(name, age):

    print(type(name))
    print(type(age))
    return 'Hello {}! You are {} years old!'.format(name, age)
```

src02.py

# HTTP Methods

- Web applications use different HTTP methods when accessing URLs.
- By default, a route only responds to GET requests.
- You can use the `methods` argument of the `route()` decorator to handle different HTTP methods.



```python
from flask import Flask
from flask import request

app = Flask(__name__)

@app.route('/', methods=['GET', 'POST'])
def hello():
    if request.method == 'POST':
        return 'Hello POST!'
    else:
        return 'Hello GET!'
```

src03.py

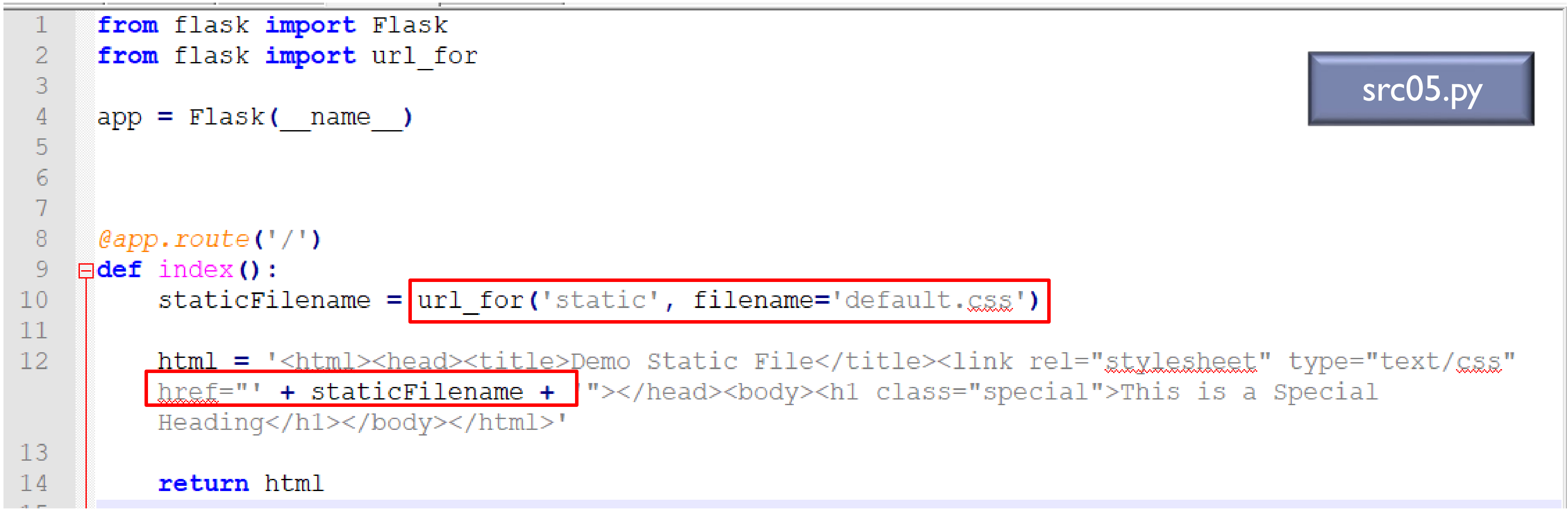
# HTTP Methods (cont.)

- What is the advantage of using HTTP POST method?
  - Data length limit:
    - GET has a limitation on the length of the values, generally 255 characters.
    - POST has no limitation on the length of the values.
  - POST method can be used to send HTML form data to the server:
    - HTML form data posted to the server are identified by their respective name attribute defined in the corresponding HTML input control.
    - Data received by POST method is not cached by server and needs to be processed, e.g., saved or persisted into a database.
  - See sample source file `src04.py`

# Static Files

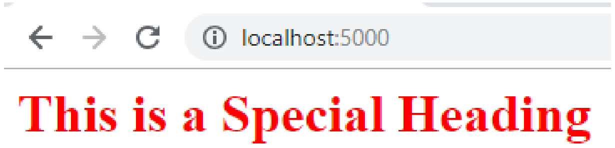
- Most web applications are designed to serve dynamic content.
- However, static files such as external JavaScript files and CSS stylesheets are still required.
- Flask is configured to serve static files from the `static` folder in the application folder:
  - Static files in the static folder will be made available at the URL `/static` of the web application.
  - The `url_for()` method can be used to generate URLs for static files.
  - E.g., `url_for('static', filename='style.css')` will reference the file stored at `static/style.css`.

# Static Files (cont.)

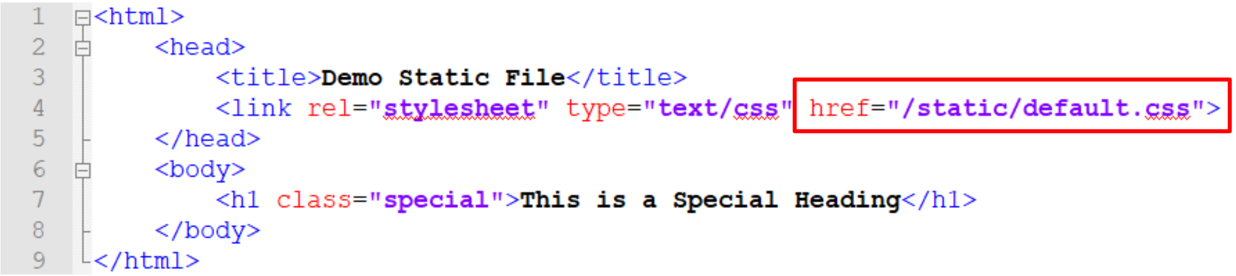


src05.py

```python
from flask import Flask
from flask import url_for

app = Flask(__name__)



@app.route('/')
def index():
    staticFilename = url_for('static', filename='default.css')

    html = '<html><head><title>Demo Static File</title><link rel="stylesheet" type="text/css" href="' + staticFilename + '"></head><body><h1 class="special">This is a Special Heading</h1></body></html>'

    return html
```



This is a Special Heading



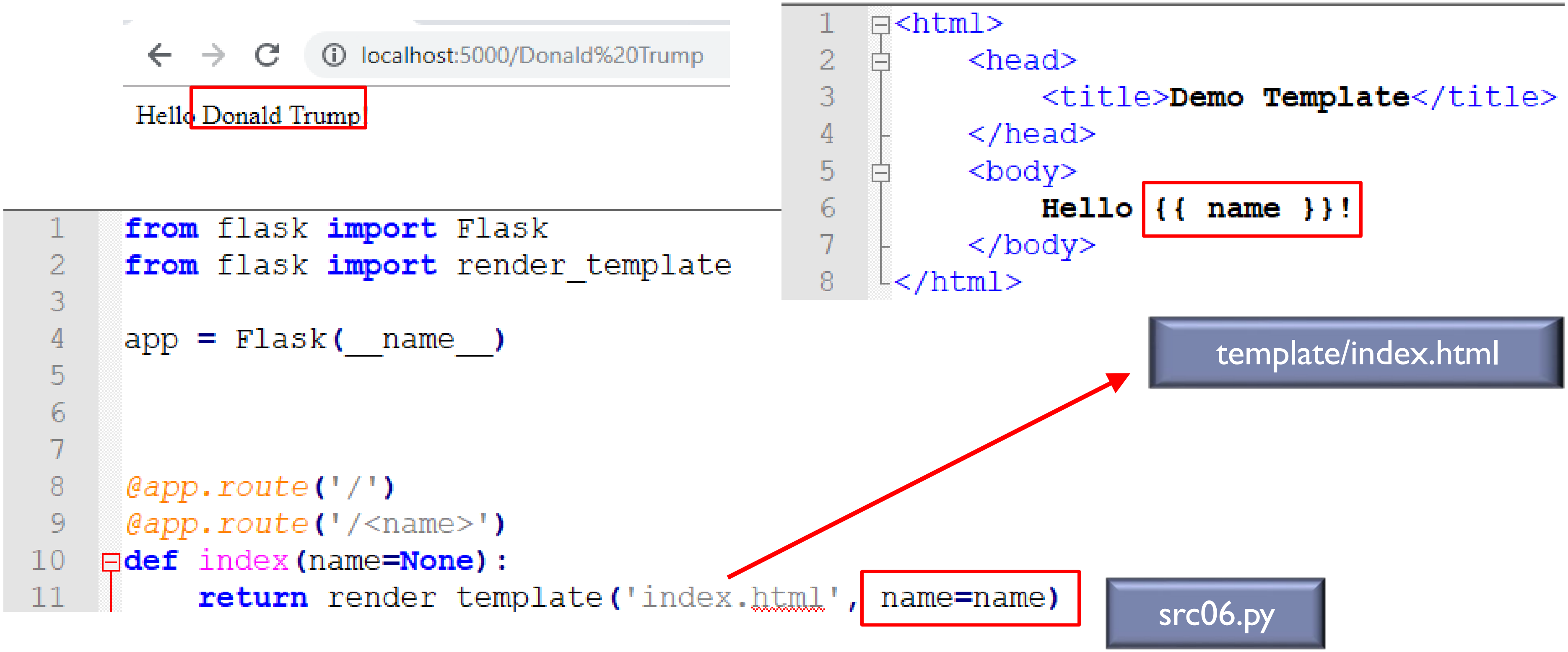
```html
<html>
    <head>
        <title>Demo Static File</title>
        <link rel="stylesheet" type="text/css" href="/static/default.css">
    </head>
    <body>
        <h1 class="special">This is a Special Heading</h1>
    </body>
</html>
```

# Rendering Templates

- Generating HTML from within a Python class is tedious and cumbersome.
- Flask is configured with the Jinja2 template engine automatically:
  - When used for web application, a Jinja template can contain static HTML and dynamic content.
  - Dynamic content can be rendered using the `request`, `session` and `g` (i.e., application) objects as well as parameters.
- The `render_template()` method is used to render a template:
  - Provide the name of the template.

# Rendering Templates (cont.)

- Provide the variables to be passed to the template engine as keyword arguments.
- All template files must be placed in the `templates` folder.



localhost:5000/Donald%20Trump

Hello Donald Trump

```html
<html>
    <head>
        <title>Demo Template</title>
    </head>
    <body>
        Hello {{ name }}!
    </body>
</html>
```

template/index.html

```python
from flask import Flask
from flask import render_template

app = Flask(__name__)



@app.route('/')
@app.route('/<name>')
def index(name=None):
    return render template('index.html', name=name)
```

src06.py

# Session State Management in Flask

- **Session** refers to the time interval when a client logs into a server and logs out of it:
  - The data, which is needed to be held across this session, is stored in a temporary directory on the server.
  - These are also known as session data or conversational state.
  - A session with each client is assigned a Session ID.
  - The Session data is stored on top of cookies and the server signs them cryptographically.
  - To perform this encryption, a Flask application needs to define a `SECRET_KEY`, which can be any random string.

# Session State Management in Flask (cont.)

- In Flask, session data are stored in the `session` object:
  - This is a Python dictionary object containing key-value pairs of session variables and associated values.
  - `session['name'] = value` is used to set a new session variable.
  - `session.pop('name', None)` is used to remove a session variable:
    - `None` becomes the default value that is to be returned when the key is not in the dictionary.
- See sample source file `src07.py` for a complete example.

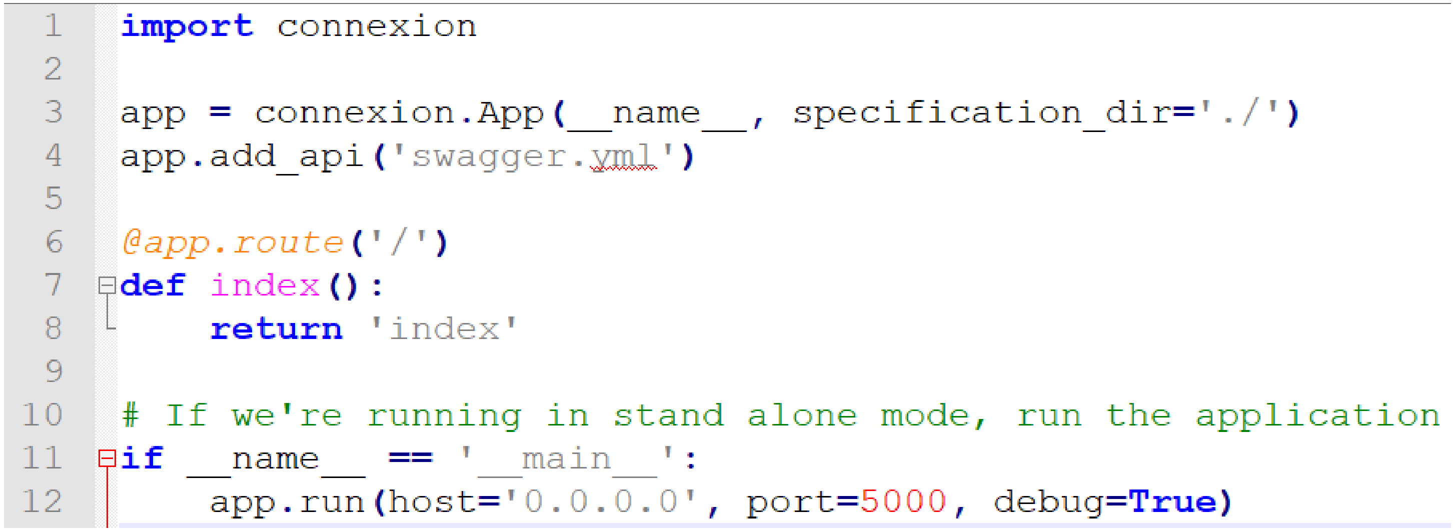
# Building RESTful Web Services

- Swagger:
  - An open-source software framework of tools.
  - Helps developers design, build, document and consume RESTful web services.
  - Currently known as OpenAPI.
- Connexion:
  - A framework for building and managing RESTful web service in Python on top of Flask using Swagger.
  - Supports automatic endpoint validation and OAuth2.
  - Can be installed with pip:

```
python -m pip install connexion
```

# Building RESTful Web Services (cont.)

- To add a RESTful web service endpoint to a Flask application:
  - Need to import the `connexion` module.
  - Then create a Swagger configuration file.



```python
import connexion

app = connexion.App(__name__, specification_dir='./')
app.add_api('swagger.yml')

@app.route('/')
def index():
    return 'index'

# If we're running in stand alone mode, run the application
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=True)
```

src08.py

# Building RESTful Web Services (cont.)

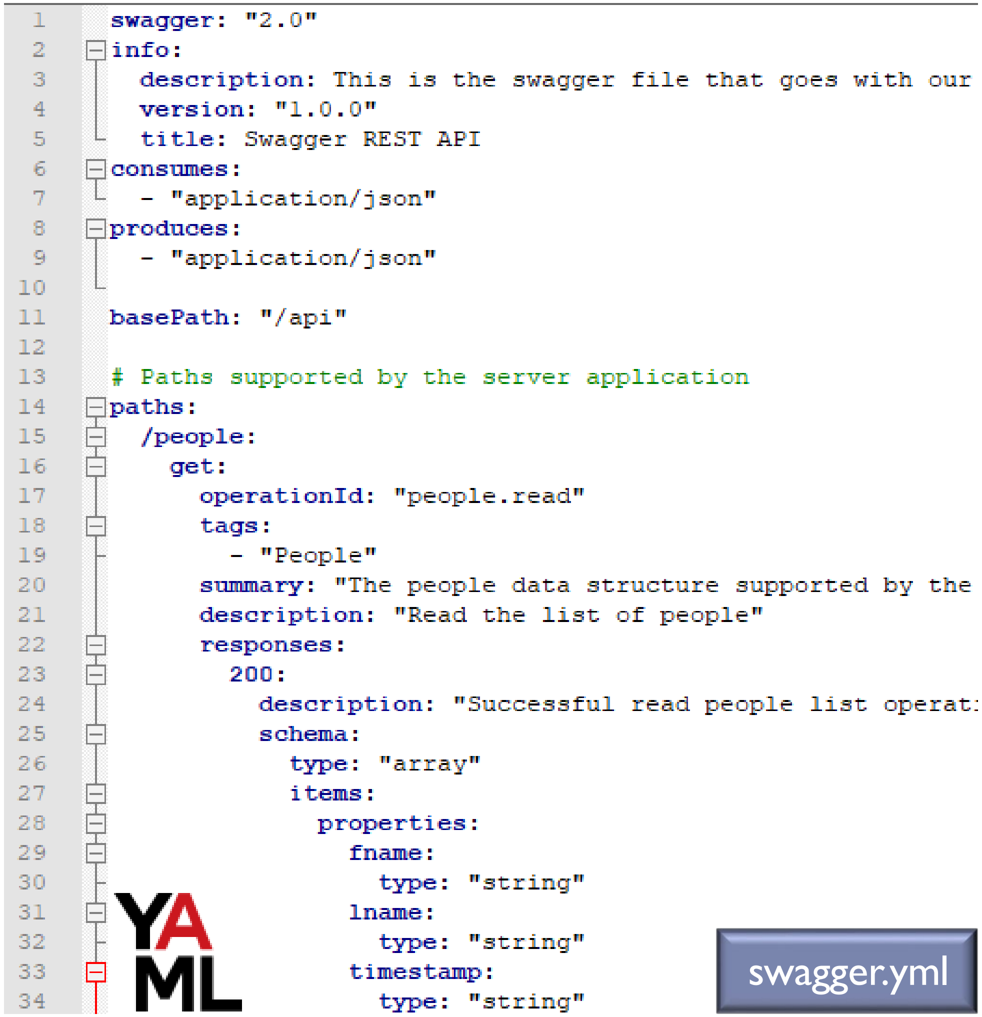
- Let's examine the details in `src08.py`:
  - `import` statement adds the `connexion` module to the program.
  - Create the application instance using Connexion rather than Flask:
    - Internally, the Flask app is still created but it now has additional functionality added to it.
    - A Connexion app can still serve HTML content.
    - The `app` instance creation command includes the parameter `specification_dir`.
    - This parameter tells Connexion which directory to look in for its configuration file.
    - In this case, the configuration file is placed in the current directory.

# Building RESTful Web Services (cont.)

- Configure the app instance to:
  - Read the file `swagger.yml` from the specification directory.
  - Provide the corresponding Connexion functionalities.

# Swagger Configuration File

- Configuration file:
  - YAML or JSON file containing information to configure the server to provide:
    - URL endpoint definition.
    - Input parameter validation.
    - Output response data validation.
    - Swagger UI.
  - **YAML** is a human-readable data serialization format – YAML Ain't Markup Language



```yaml
swagger: "2.0"
info:
  description: This is the swagger file that goes with our
  version: "1.0.0"
  title: Swagger REST API
consumes:
  - "application/json"
produces:
  - "application/json"

basePath: "/api"

# Paths supported by the server application
paths:
  /people:
    get:
      operationId: "people.read"
      tags:
        - "People"
      summary: "The people data structure supported by the
      description: "Read the list of people"
      responses:
        200:
          description: "Successful read people list operat
          schema:
            type: "array"
            items:
              properties:
                fname:
                  type: "string"
                lname:
                  type: "string"
                timestamp:
                  type: "string"
```

YAML

swagger.yml

# Swagger Configuration File (cont.)

- `swagger.yml` file defines the `GET /api/people` endpoint that our RESTful web service will provide.
- The file is organized in a hierarchical manner:
  - The indentation levels represent a level of ownership, or scope.
  - `paths` section defines the prefix string of the URLs for the API endpoints:
    - The `/people` value indented under `paths` defines the prefix string for all the `/api/people` URL endpoints.
    - The `get:` indented under `/people` defines the section of definitions associated with an HTTP GET request to the `/api/people` URL endpoint.
  - The same format is used to organize the remainder of the configuration for other API endpoints.

# Swagger Configuration File (cont.)

- Global configuration information:
  - `swagger` – Specify version of the Swagger API being used.
  - `info`:
    - Begins a new "scope" of information about the API being built.
    - `description`:
      - □ A user defined description of what the API provides.
      - □ This will be used in the Connexion generated UI system.
    - `version` – A user defined version value for the API.
    - `title` – A user defined title included in the Connexion generated UI system.
- `consumes` – Tells Connexion what <u>MIME type</u> is expected by the <u>input</u> of the API.

# Swagger Configuration File (cont.)

- `produces` – Tells Connexion what MIME type is expected by the caller of the API's output.
- `basePath` – `"/api"`
  - Defines the root of the API.
  - Path without `basePath` as the prefix will be served by Flask.
- `paths` section begins the configuration of the actual API REST endpoints:
  - `/people` – Defines the path for the URL endpoint.
  - `get` (can include other HTTP methods):
    - Defines the HTTP method this URL endpoint will respond to.
    - Together with the previous definitions, this creates the `GET /api/people` URL endpoint.

# Swagger Configuration File (cont.)

- The next section defines the configuration of the single `GET /api/people` URL endpoint:
  - `operationId`:
    - "`people.read`" defines the Python import path/function that will respond to a HTTP `GET /api/people` request.
    - `operationId` can go as deep as required to connect a Python function to the HTTP request.
    - E.g., `<package_name>.<package_name>.<package_name>.<function_name>` would work too.
  - `tags`:
    - Defines a grouping for the UI interface.
    - All HTTP methods that are defined for the people endpoint will share this tag definition.

# Swagger Configuration File (cont.)

  - `summary` – Defines the UI interface display text for this endpoint.
  - `description` – Defines what the UI interface will display for implementation notes.
- The last section defines the configuration of responses from the URL endpoint:
  - `responses` – Defines the beginning of the expected response section.
  - `200`:
    - Defines the section for a successful response, i.e., HTTP status code 200.
    - Responses for other HTTP status codes can also be configured.

# Swagger Configuration File (cont.)

- `description` – Defines the UI interface display text for a response of 200.
- `schema` – Defines the response as a schema, or structure.
- `type` – Defines the structure of the schema as an `array`.
- `items` – Define the items in the array.
- `properties` – defines the items in the array as objects having key/value pairs:
  - `fname` – Defines the first key of the object.
    - □ `type` – Defines the value associated with `fname` as a `string`.
  - `lname` – Defines the second key of the object.
    - □ `type` – Defines the value associated with `lname` as a `string`.
  - `timestamp` – Defines the third key of the object.
    - □ `type` – defines the value associated with `timestamp` as a `string`.

# Handler for RESTful Web Service Endpoint

- Recall that in `swagger.yml`, we have configured Connexion with:
  - The `operationId` value to call the `people` module.
  - The `read` function within the module is invoked when the API receives an HTTP request for `GET /api/people.`
- This means a `people.py` module must exist and contain a `read()` function.
- Important notes about the sample code:
  - `PEOPLE` is a dictionary data structure:
    - This is a simple names database, keyed on the last name.
    - This is a module variable, so its state persists between method calls:
      - □ Can modify the data structure.

# Handler for RESTful Web Service Endpoint (cont.)

- In a real application, the `PEOPLE` data will exist in a database, i.e., persists the data beyond the web application instance.
- `read()` function:
  - Invoked when an HTTP request to `GET /api/people` is received by the server.
  - Return value of this function is converted to a JSON string (recall the `produces` definition in the `swagger.yml` file).
  - Builds and returns a `list` of people sorted by last name.
- Use the following command to run the Connexion app:

```
python src08.py
```

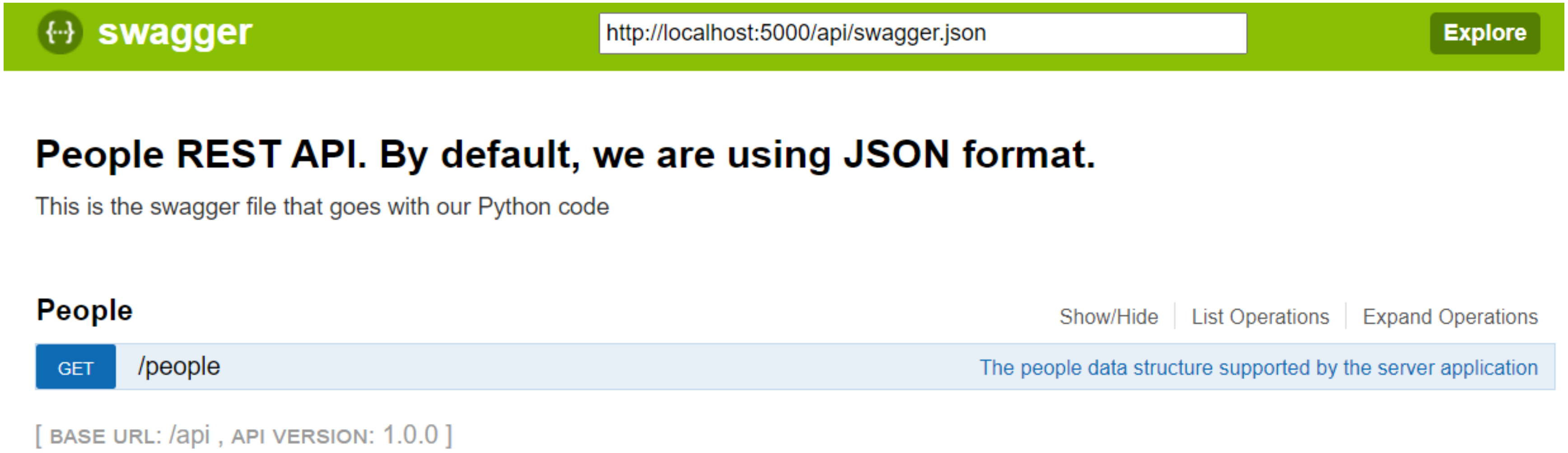
# The Swagger UI

- What have been demonstrated thus far:
  - A simple RESTful web service running with a single URL endpoint.
  - `swagger.yml` provided a definition for the code path connected to the web service endpoint.
- In addition, **Swagger UI** is automatically created for our web service:
  - Requires installation of the `connexion` module with an additional `swagger-ui` option:

```
python -m pip install connexion[swagger-ui]
```

# The Swagger UI (cont.)

- Navigating to http://localhost:5000/api/ui/ will launch a web page that resembles the screenshot on the next slide.
- Note the trailing slash in the URL.



People REST API. By default, we are using JSON format.

This is the swagger file that goes with our Python code

People

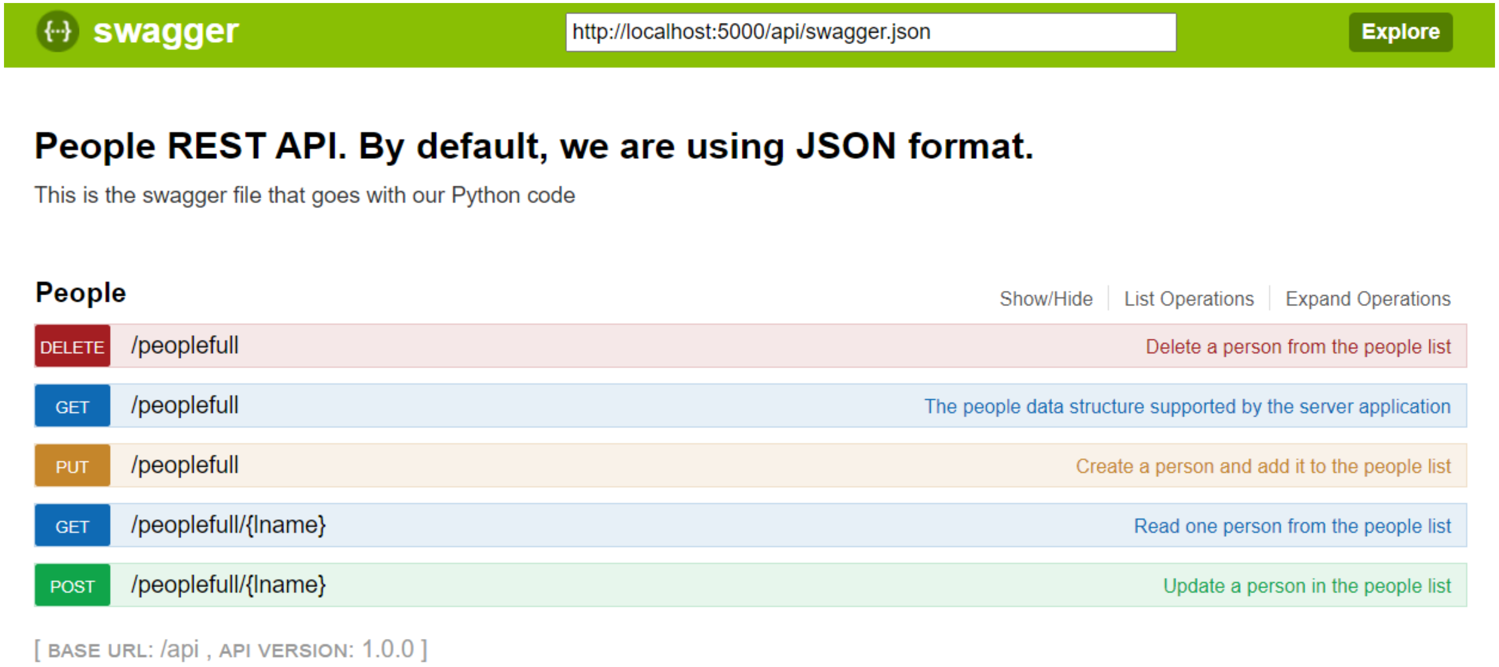
/people — The people data structure supported by the server application

[ BASE URL: /api , API VERSION: 1.0.0 ]

- The above screenshot shows the initial Swagger interface with the list of URL endpoints supported.

# Building the Complete RESTful Web Service

- The ultimate goal is to build a RESTful web service that provides full CRUD access to our `PEOPLE` data structure.
- Refer to `swaggerfull.yml` and `peoplefull.py` for the complete implementation of the remaining use cases.



**People REST API. By default, we are using JSON format.**

This is the swagger file that goes with our Python code

**People**

| | | |
|---|---|---|
| DELETE | /peoplefull | Delete a person from the people list |
| GET | /peoplefull | The people data structure supported by the server application |
| PUT | /peoplefull | Create a person and add it to the people list |
| GET | /peoplefull/{lname} | Read one person from the people list |
| POST | /peoplefull/{lname} | Update a person in the people list |

[ BASE URL: /api , API VERSION: 1.0.0 ]

# Building the Complete RESTful Web Service (cont.)

- It is relatively easy to create a comprehensive RESTful web service with Python:
  - Use the `connexion` module and some additional configuration.
  - A useful documentation and interactive system can also be put in place.

# Testing RESTful Web Service with Postman

- Postman (https://www.postman.com/) can be used to test RESTful web service methods:
  - Indicate the required URI of the resource.
  - Select the required HTTP verb corresponding to the web service method.
- Based on the HTTP verb of the resource being called, the following parameters would need to be provided:
  - Header field(s).
  - Query string parameter(s).
  - Path parameter(s).
  - Body of the HTTP PUT and POST request:
    - Mainly for JSON formatted requests.

# Consuming RESTful Web Service in Python

- Python **`requests`** library:
  - `requests` is a Python library that enables the sending of HTTP requests.
  - `requests` library supports the four main HTTP request methods of `get()`, `put()`, `post()` and `delete()`.
  - Use the following command to install the library:

```
python -m pip install requests
```

- The `put()` and `post()` methods allow data to be sent to the server as a JSON object:
  - In Python, a `dict` data structure can be converted into a JSON object using the Python **`json`** library.

# Consuming RESTful Web Service in Python (cont.)

- Configure the headers:
  - Set "content-type" value to "application/json":
  - Matches the `consumes` configuration of the web service.
- The `get()` and `delete()` methods allow `params` representing the path or query string parameters to be passed in as a `dict` object.
- Sample source file `src09.py` demonstrates how to call the people web services using the `requests` library.

# Persisting Data to a Relational Database

- Python provides an elaborate set of libraries for working with relational database management systems (RDBMS):
  - Most major RDBMSs such as SQLite, MySQL and PostgreSQL are supported.
  - MySQL Connector for Python – Driver and API for MySQL.
  - SQLAlchemy – An open-source SQL toolkit and object-relational mapper (ORM) for Python.
- On the Raspberry Pi, due to the lower computational capability, SQLite is preferable to MySQL:
  - SQLite is serverless whereas MySQL is server-based.
  - Bear in mind that we are also running other IoT control applications, `connexion`, etc.

# Persisting Data to a Relational Database (cont.)

- Sample source code `src10.py` demonstrates how to work with MySQL Connector for Python:
  - Need to install the driver with pip:

    ```
    python -m pip install mysql
    python -m pip install mysql-connector-python
    ```

  - Use basic SQL DML (data manipulation language) statements to complement the CRUD use cases for the people structure.
  - We will not be using ORM.

# Case Study Walk-through

# Ambient Temperature Case Study

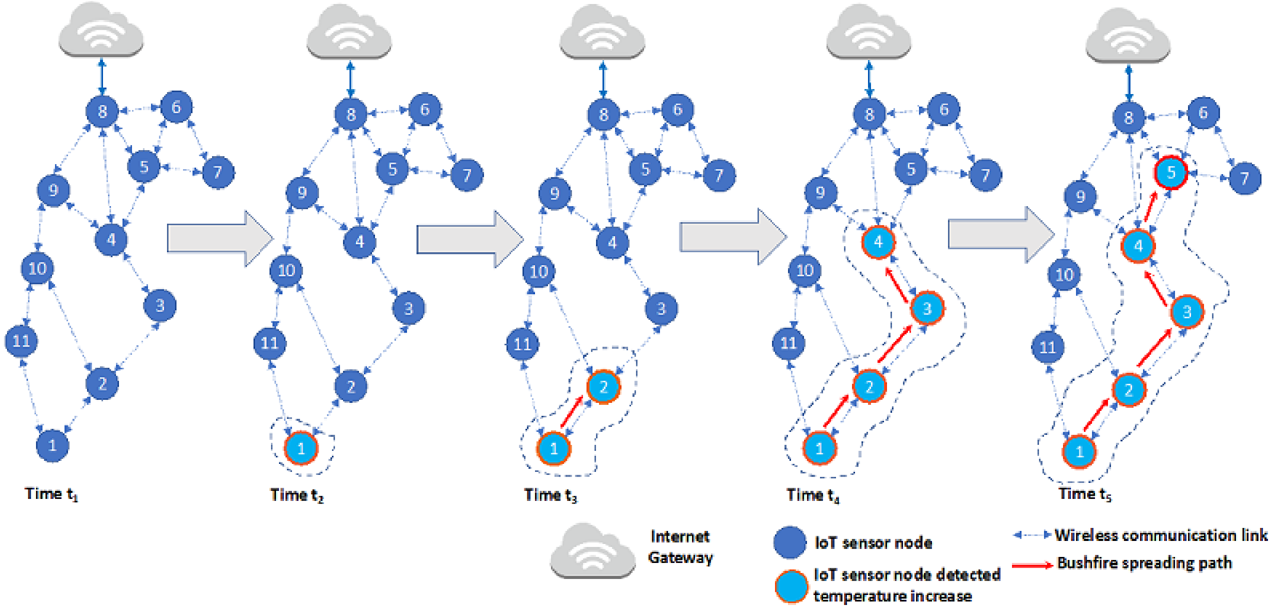
- Suppose we want to track the ambient temperature across a large geographical region:
  - The region is broken down into several areas.
  - Each area is further broken down into several districts.
  - Sensor nodes are deployed across the entire region with multiple nodes monitoring the ambient temperature of each district.
  - Sensor nodes in each district reports their sensor data to a hub and fog processor at the area-level.
  - The fog processor in term relays the sensor data at periodic interval to a cloud server at the region-level.

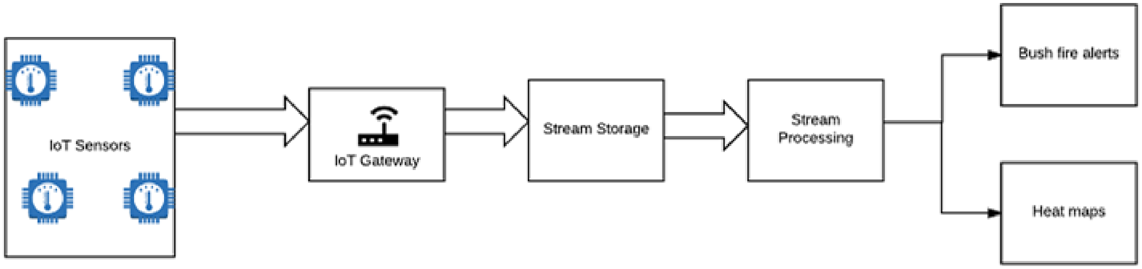
# Ambient Temperature Case Study (cont.)

- Hub:
  - Provides Internet connectivity to the node devices.
- Fog processor:
  - Provides temperature information services to local municipal authority for monitoring temperature changes within its area.
  - Fast response time to dispatch resources.
- Cloud server:
  - Provides temperature information services to central government authority for monitoring temperature changes at the region-level.
  - Sufficient response time to escalate central resources to assist local resources.

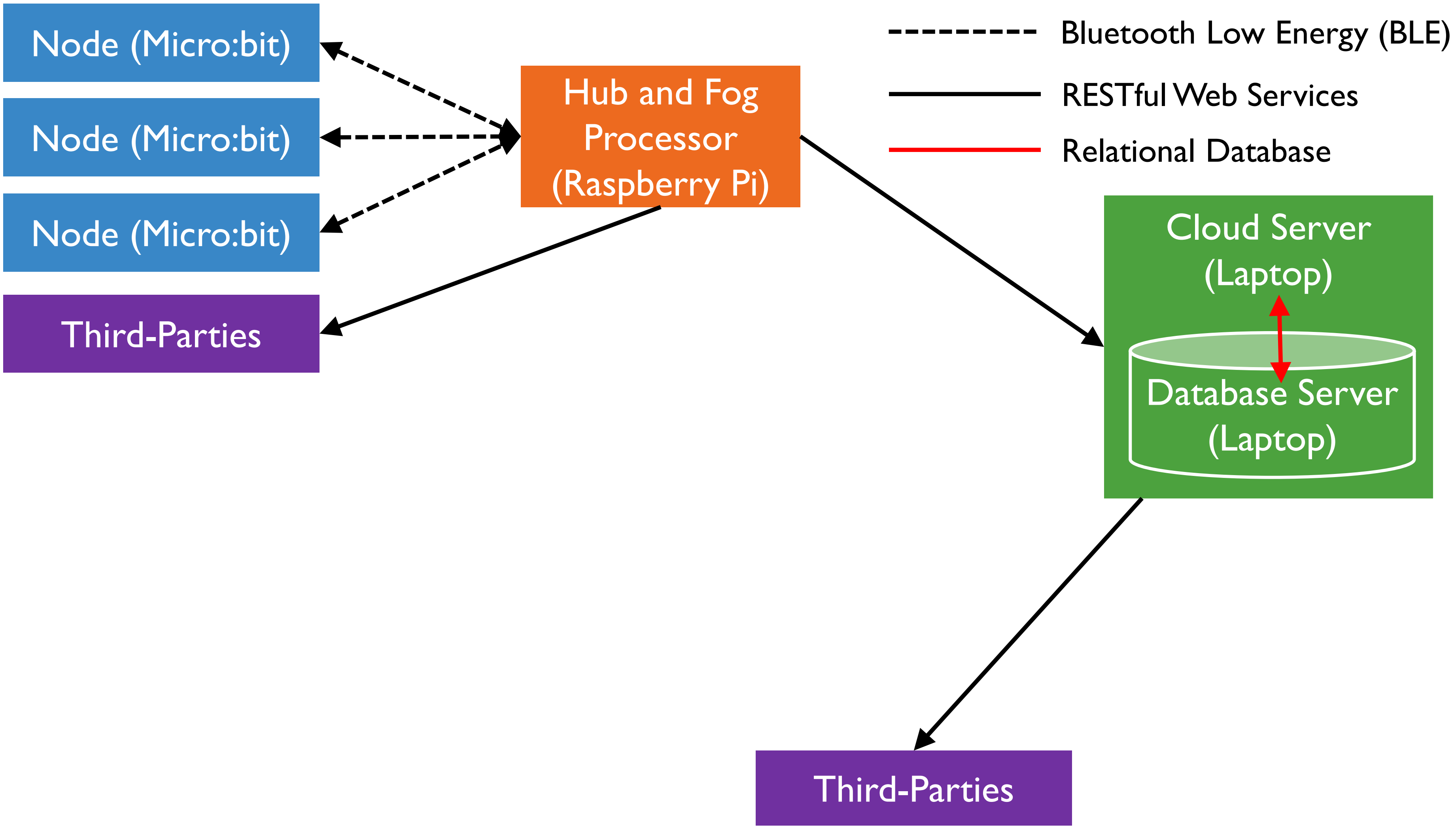
# Ambient Temperature Case Study (cont.)

- A plausible scenario in the real-world:
  - Real-time bushfire alerting with complex event processing.

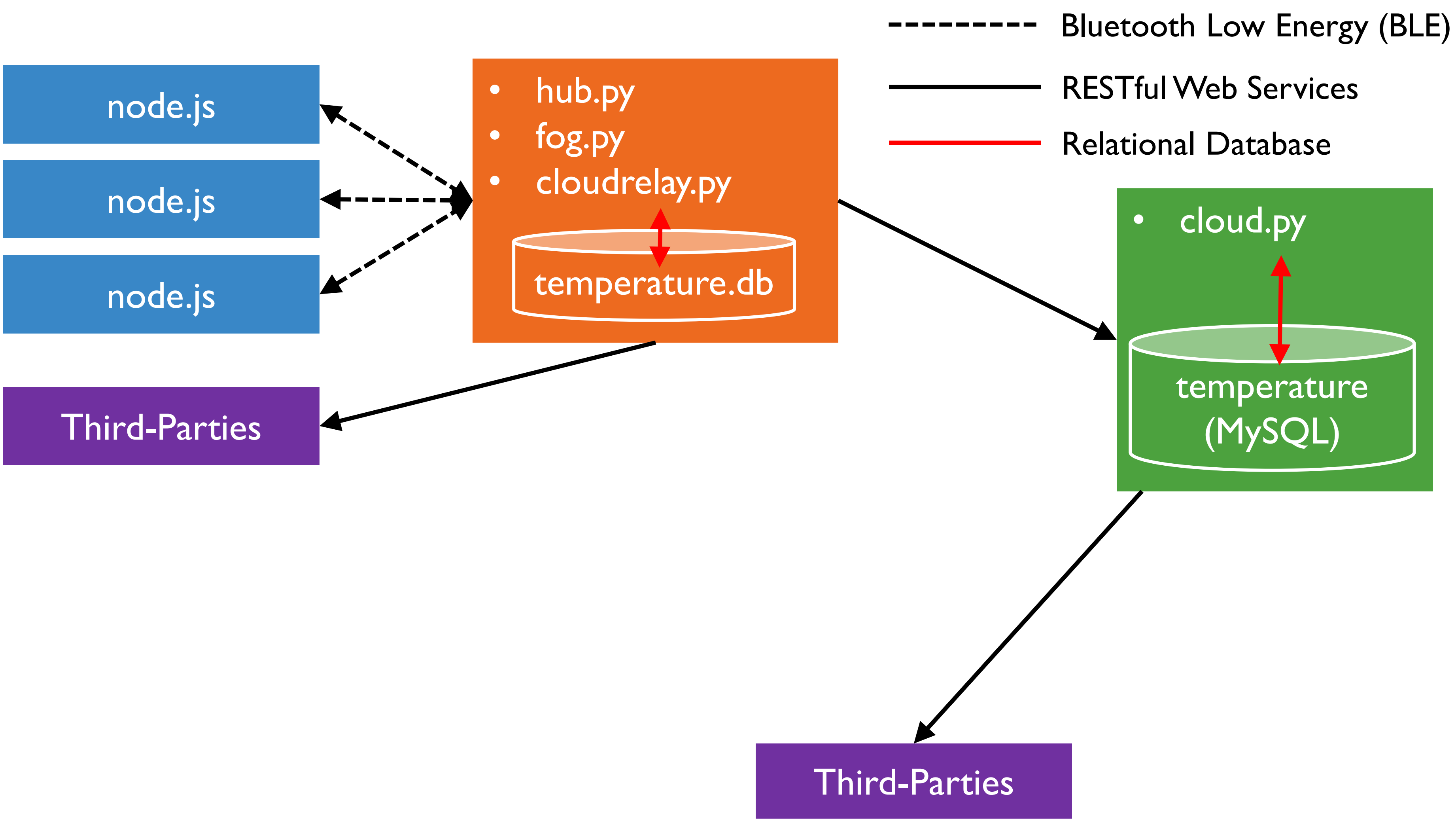




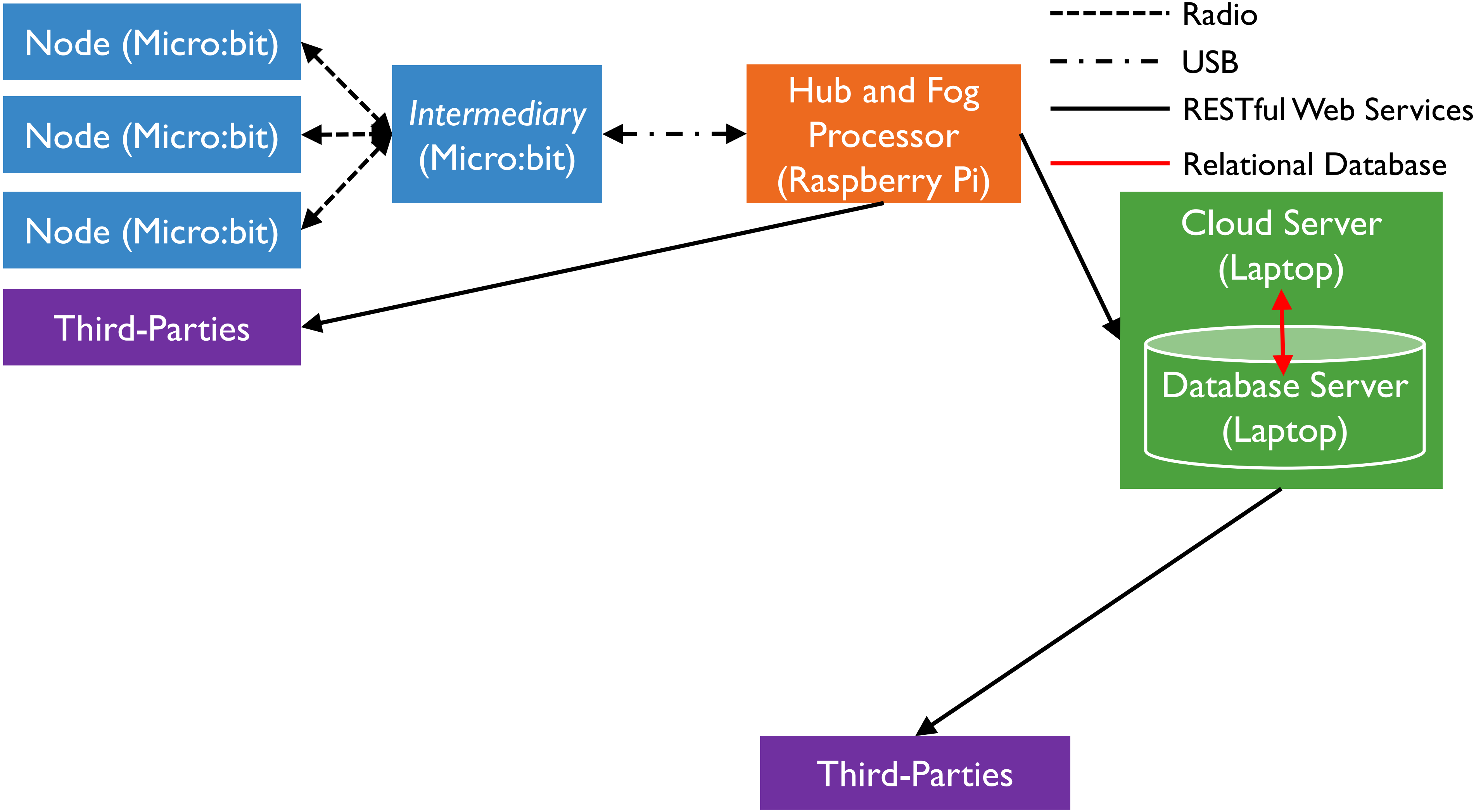
# Demonstration Setup



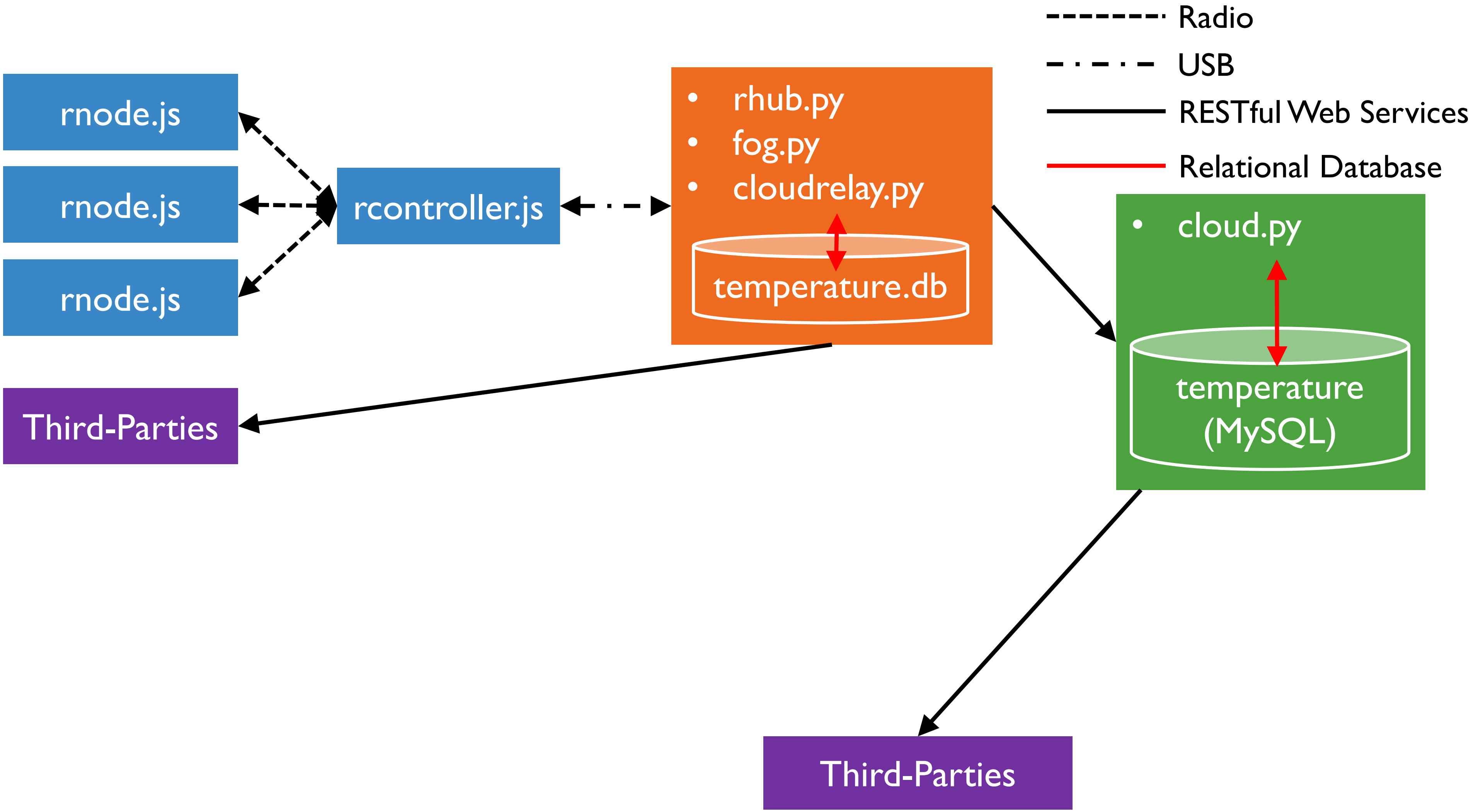
# Demonstration Setup (cont.)



# Demonstration Setup (cont.)



# Demonstration Setup (cont.)



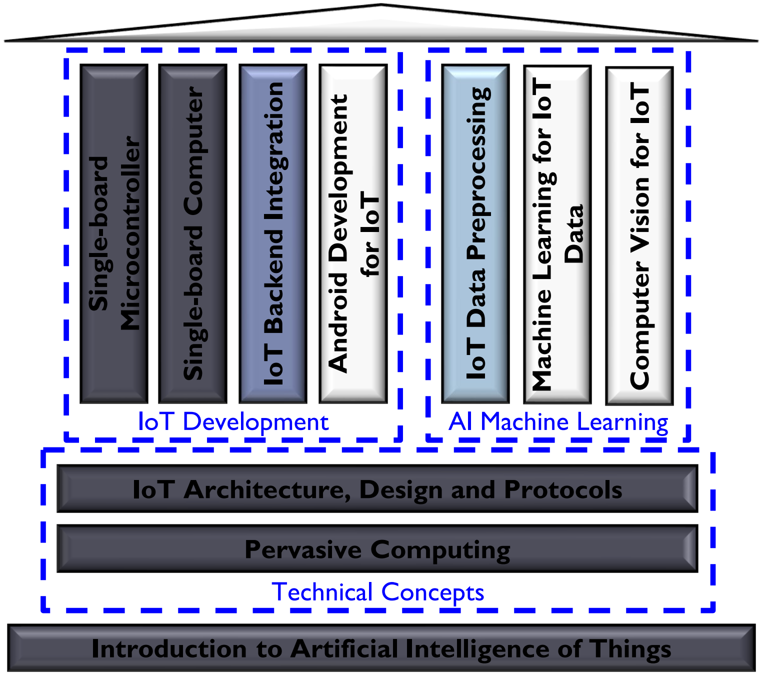
# Summary

- RESTful web services provide a lightweight and flexible approach for software elements in an IoT system to interact with each other.
- Single-board computer such as the Raspberry Pi can act as an integrated hub and fog processor by running RESTful web services and relational databases on it.

# Q&A

# Next Lecture…

- Learn about:
  - More about machine learning.
  - How to perform data preparation with Pandas.
  - How to perform data visualisation with Matplotlib.


Single-board Microcontroller
Single-board Computer
IoT Backend Integration
Android Development for IoT
IoT Data Preprocessing
Machine Learning for IoT Data
Computer Vision for IoT
IoT Development
AI Machine Learning
IoT Architecture, Design and Protocols
Pervasive Computing
Technical Concepts
Introduction to Artificial Intelligence of Things