Introductory Course on Network and Connectivity

High-Level Networking & REST API Basics

Faculdade de Design,
Tecnologia e Comunicação
Universidade Europeia

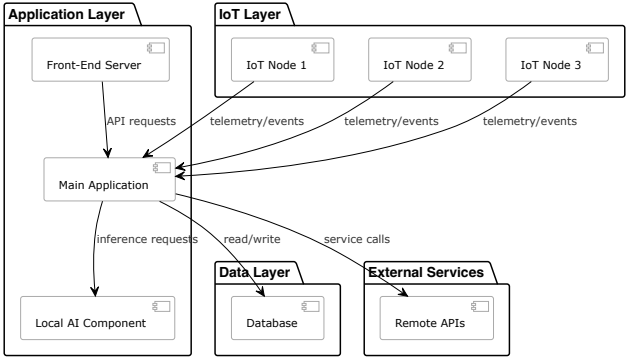
Why networking matters for Creative Technologies?

Does it matter to you?

What is the most networked application you've used recently?

Enabling connectivity

Networking allows creative technologies to connect and communicate with each other, enabling the integration of devices, applications, and services. This connectivity is important for the development of interactive and immersive experiences.



Endgame

The endgame is to build and integrate networked applications using APIs.

Say you want to build a mobile app that tracks fitness data from wearable devices and syncs it with a cloud service for analysis and visualization.

- The mobile app communicates with the wearable device to collect fitness data (e.g., steps, heart rate) using Bluetooth or Wi-Fi.
- The app then sends this data to a cloud-based REST API for storage and processing.
- The cloud service analyzes the data and provides insights, which the app retrieves and displays to the user.

Your job is to make all these components work together seamlessly. To that end, you need to understand the basics of computer networks and how to interact with REST APIs [1].

[1] Eventually, you'll also need to understand authentication, security, and data privacy. And there are more protocols than just REST, like GraphQL, gRPC, and WebSockets.

Technical requirements

To follow along, you need to install Python 3.7 or higher.

On MacOS, you can use Homebrew (`brew install python`). If you don't have Homebrew, install it from https://brew.sh/.

On GNU/Linux, use your package manager. For example, for Ubuntu/Debian, `sudo apt install python3 python3-venv python3-pip`.

On Windows, use the official installer, and make sure to check "Add Python to PATH" during installation. Get the installer at https://www.python.org/downloads/.

Use virtual environments (e.g., `venv`). Create the environment with `python3 -m venv .venv`, and activate it with `source .venv/bin/activate`.

Install a rest API framework. We will use FastAPI, which you can install with `pip install "fastapi[standard]"`

Repository

https://github.com/IADE-Coding/MCCIA2526NetworksClass1

Connecting multiple devices to share resources and information

A network is a collection of computers, servers, mainframes, network devices, peripherals, or other devices connected to one another to allow the sharing of data.

Networks are essential for modern computing, enabling communication, resource sharing, and access to information across the globe. They support various applications, from web browsing and email to cloud computing and IoT.

Key concepts about networks

- **Protocols**: Standardized rules for data exchange (e.g., HTTP, FTP, TCP/IP).
- **IP Address**: A unique identifier for devices on a network.
- **Client-Server Model**: A network architecture where clients request services and resources from centralized servers.
- **Cloud Computing**: Using remote servers hosted on the internet to store, manage, and process data.
- **IoT**: Network of physical devices embedded with sensors and software to connect and exchange data.
- **Network Devices**: Hardware components like routers and switches, that facilitate network connectivity and communication.
- **Network Services**: Essential services like DHCP (Dynamic Host Configuration Protocol) for IP address assignment and NAT (Network Address Translation) for modifying network address information in IP packet headers.

OSI Model

The OSI (Open Systems Interconnection) model is a conceptual framework used to understand and implement network protocols in seven layers. Each layer serves a specific function and communicates with the layers directly above and below it.

1. **Physical Layer**: Deals with the physical connection between devices, including cables, switches, and other hardware.
2. **Data Link Layer**: Responsible for node-to-node data transfer and error detection/correction (e.g., Ethernet, MAC addresses).
3. **Network Layer**: Manages data routing, addressing, and packet forwarding (e.g., IP).
4. **Transport Layer**: Ensures reliable data transfer between end systems (e.g., TCP, UDP).
5. **Session Layer**: Manages sessions and connections between applications.
6. **Presentation Layer**: Translates data formats and handles encryption/decryption.
7. **Application Layer**: Provides network services directly to end-user applications (e.g., HTTP, FTP).

TCP/IP (Internet Protocol Suite)

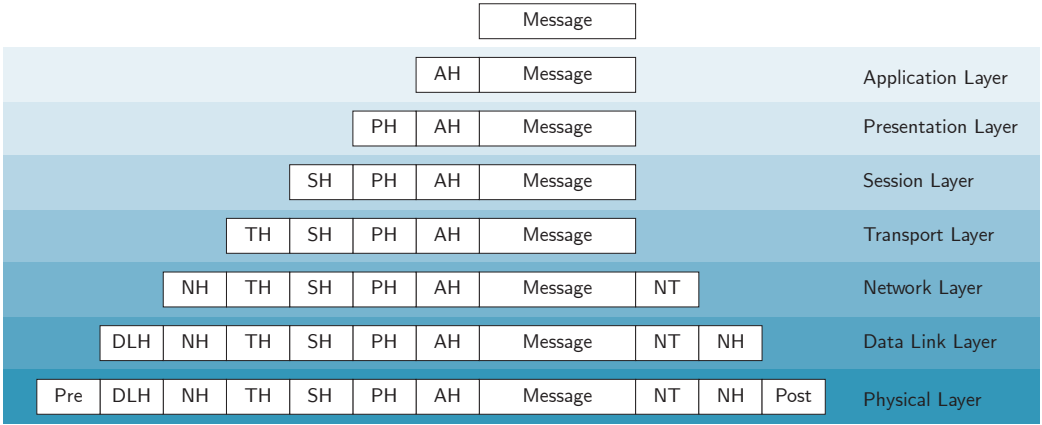
The TCP/IP model is a simplified framework for understanding network communication, consisting of four layers that correspond to the OSI model.

1. **Network Interface Layer**: Combines the OSI's Physical and Data Link layers, handling the physical transmission of data.
2. **Internet Layer**: Corresponds to the OSI's Network layer, responsible for routing and addressing (e.g., IP).
3. **Transport Layer**: Similar to the OSI's Transport layer, ensuring reliable data transfer (e.g., TCP, UDP).
4. **Application Layer**: Encompasses the OSI's Session, Presentation, and Application layers, providing services for end-user applications (e.g., HTTP, FTP, SMTP).

Common Protocols

- **TCP/UDP**: Transmission Control Protocol (reliable) / User Datagram Protocol (unreliable) for data transmission.
- **FTP/SFTP**: File Transfer Protocol / Secure File Transfer Protocol for transferring files.
- **HTTP/HTTPS**: Protocols for web communication (Hypertext Transfer Protocol / Secure).
- **SMTP/IMAP/POP3**: Protocols for email transmission and retrieval.
- **DNS**: Domain Name System for translating domain names to IP addresses.
- **DHCP**: Dynamic Host Configuration Protocol for automatic IP address assignment.
- **NAT**: Network Address Translation for modifying network address information in IP packet headers.

The life of a message



Networking may seem somewhat complicated, but thankfully, you don't need to worry about all the details to build networked applications.

What is a networked application?

A networked application is a software program that relies on a network to function, allowing multiple users or devices to connect and interact with each other. These applications can range from simple web applications to complex distributed systems.

There are several approaches to a networked architecture, each with its own advantages and disadvantages.

Key concepts of networked architectures

- **Client-Server**: Centralized servers provide resources and services to multiple clients.
- **Peer-to-Peer (P2P)**: Decentralized architecture where each device
- **Cloud-Based**: Services and resources hosted on remote servers accessed via the internet.
- **Edge Computing**: Processing data closer to the source (e.g., IoT devices) to reduce latency and bandwidth usage.
- **Microservices**: An architectural style that structures an application as a collection of loosely coupled services, each responsible for a specific functionality.
- **Service-Oriented Architecture (SOA)**: A design pattern where services are provided to other components over a network, promoting reusability and interoperability.
- **Event-Driven Architecture**: A design paradigm where the flow of the program is determined by events, such as user actions or messages from other systems.
- **Message Queuing**: A communication method where messages are sent to a queue and processed asynchronously, allowing for decoupled and scalable systems.
- **Hybrid**: Combines multiple architectures to leverage their strengths.

Real-World Examples

- **Web Applications**: E-commerce platforms, social media, online banking.
- **Mobile Apps**: Messaging apps, ride-sharing services, fitness trackers.
- **IoT Systems**: Smart home devices, industrial sensors, wearable technology.
- **Cloud Services**: Data storage, machine learning platforms, content delivery networks.
- **Gaming**: Multiplayer online games, game streaming services.
- **Collaboration Tools**: Platforms like Slack, Microsoft Teams, and Google Workspace that enable real-time communication and collaboration among users.
- **Streaming Services**: Platforms like Netflix, Spotify, and YouTube that deliver audio and video content over the internet.
- **APIs**: Services like Google Maps API, Twitter API, and payment gateways that allow developers to integrate external functionalities into their applications.

Road to networked applications

With the goal of building and integrating networked applications using APIs, we need to understand what an API is and how to interact with it.

We will build APIs and consume APIs built by others. We need to agree on a set of rules and conventions to make this possible, so that we may rely on some expectations, and allow others to do the same.

What is an API?

An API (Application Programming Interface) is a set of rules and protocols that allows different software applications to communicate with each other. It defines how requests and responses should be formatted, enabling seamless interaction between systems.

APIs are essential for modern software development, enabling integration, automation, and the creation of complex applications by leveraging existing services and functionalities.

This is our goal: build and integrate networked applications using APIs.

Types of APIs

There are several types of APIs, each with its own characteristics and use cases:

- **REST (Representational State Transfer)**: Uses standard HTTP methods and is stateless, making it scalable and easy to use.
  - This is what we will focus on.
- **SOAP (Simple Object Access Protocol)**: A protocol that uses XML for message formatting and is more rigid and complex than REST.
- **GraphQL**: A query language for APIs that allows clients to request only the data they need, reducing over-fetching and under-fetching of data.
- **gRPC**: A high-performance, open-source framework that uses HTTP/2 for transport and Protocol Buffers for serialization, enabling efficient communication between services.

HTTP Basics

HTTP (Hypertext Transfer Protocol) is the foundation of data communication on the web. It defines how messages are formatted and transmitted, and how web servers and browsers should respond to various commands.

The protocol relies on a schema to locate resources. An URI (Uniform Resource Identifier) is a string of characters that identifies a particular resource, while a URL (Uniform Resource Locator) is a specific type of URI that provides the means to access that resource over the internet.

HTTP messages are mainly text-based and consist of requests sent by clients and responses returned by servers.

The REST architectural style uses standard HTTP methods to perform operations on resources identified by URIs.

HTTP Messages

An HTTP message consists of:

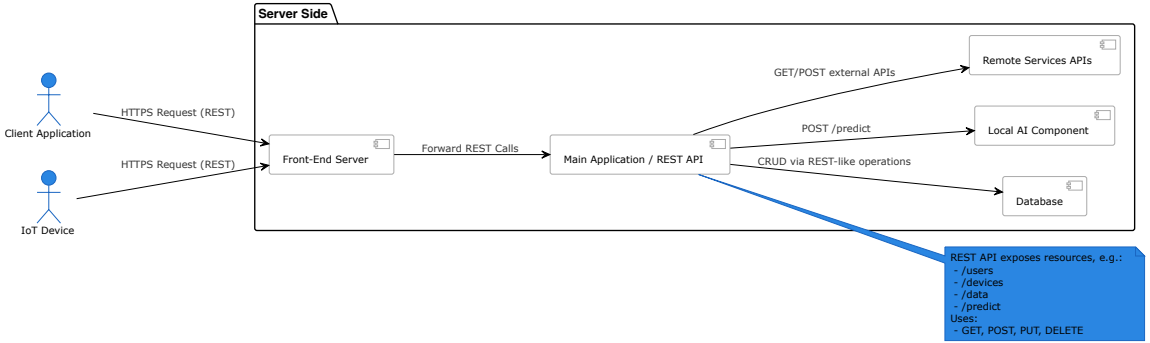
1. **Request Line**: Contains the HTTP method (e.g., GET, POST), the URI, and the HTTP version.
2. **Headers**: Key-value pairs that provide additional information about the request or response (e.g., Content-Type, Authorization)[2].
3. **Body**: Optional data sent with the request or response, typically used for POST and PUT requests.
4. **Status Line**: In responses, it includes the HTTP version, status code (e.g., 200 OK, 404 Not Found), and a reason phrase[3].

You can inspect HTTP messages using browser developer tools or command-line tools like `curl` and `httpie`.

[2] More about headers at https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers
[3] More about codes at https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status

REST

REST (Representational State Transfer) is an architectural style for designing networked applications. It relies on a stateless, client-server communication model and uses standard HTTP methods to perform operations on resources.



REST API Principles

Resources are the key abstraction in REST. They represent data or objects that can be accessed and manipulated through the API. Each resource is identified by a unique URI.

Other relevant notions include:

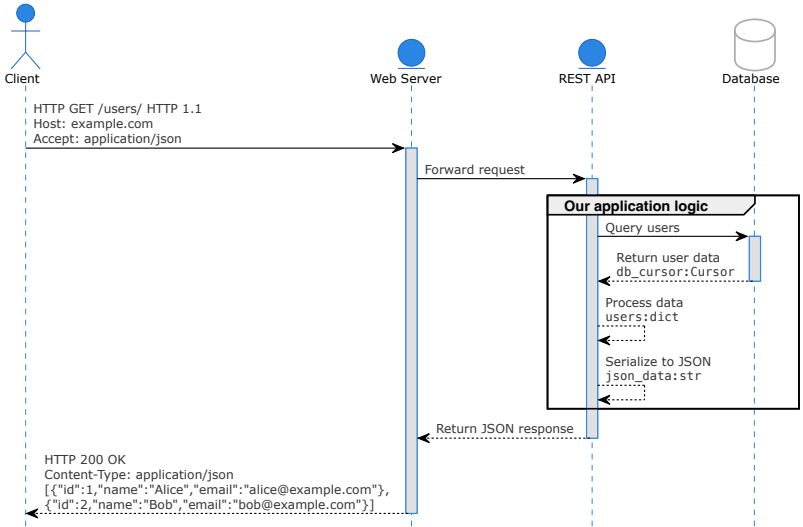
1. **Stateless**: Each request from a client to a server must contain all the information needed to understand and process the request. The server does not store any client context between requests.
   - Which is not to say there is no persistence of data, just that the server does not keep track of client state.
2. **Uniform Interface**: A consistent and standardized way of interacting with the API, typically using HTTP methods (GET, POST, PUT, DELETE) and resource URIs.
3. **Layered System**: The architecture can be composed of multiple layers, with each layer having specific responsibilities, allowing for scalability and flexibility.

Common HTTP Methods

Because HTTP is the foundation of REST, we use its methods to define our (uniform) interface. This interface is composed of method calls that operate on resources. The most common HTTP methods (or *verbs*) used in REST APIs are:

- **GET**: Retrieve data from the server (e.g., fetch a list of users, fetch a single user).
  - This is the only safe and idempotent method. The body of a GET request is usually empty.
    - *safe* means it doesn't modify data.
    - *idempotent* means that making the same request multiple times has the same effect as making it once.
- **POST**: Send data to the server to create a new resource (e.g., create a new user, create a new list of items).
  - Not idempotent, not safe, and usually carries a body with the data to create.
- **PUT**: Update an existing resource on the server (e.g., update user information).
  - Idempotent, not safe, and usually carries a body with the data to update.
- **DELETE**: Remove a resource from the server (e.g., delete a user).
  - Idempotent, not safe, and usually has an empty body.

Example: Get all the records of a collection of resources



Data Formats

As we've seen, HTTP messages are text-based. Other than that, the protocol does not impose a specific data format. However, most REST APIs use common data formats for request and response bodies, including:

- **JSON (JavaScript Object Notation)**: A lightweight, human-readable format for data exchange, widely used in REST APIs.
  - This is the format we use in our examples.
- **XML (eXtensible Markup Language)**: A more verbose format for data exchange, less commonly used in modern REST APIs but still prevalent in SOAP APIs.
- **YAML (YAML Ain't Markup Language)**: A human-friendly data serialization format often used for configuration files and data exchange in some APIs.

It is the responsibility of the API to define which formats are supported, and how to use them. Using know formats is one of expectations we set when building and consuming APIs.

JSON

JSON (JavaScript Object Notation) is the most common data format for REST APIs. It is lightweight, easy to read and write, and widely supported across programming languages.

JSON represents data as key-value pairs, arrays, and nested objects.

```
{
  "id": 4,
  "name": "Trudy",
  "email": "trudy@example.com",
  "roles": [
    {"role": "admin", "expires": "2025-09-08"},
    {"role": "user", "expires": null}
  ]
}
```

More about JSON at https://www.json.org/json-en.html.

The same example of request and response

Request:

```
GET /users/ HTTP/1.1
Host: example.com
Accept: application/json
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json
[{
  "id": 1,
  "name": "Alice",
  "email": "alice@example.com"
},
{
  "id": 2,
  "name": "Bob",
  "email": "bob@example.com"
}]
```

Application development

APIs may be developed using various programming languages and frameworks. Popular choices include:

- **Python**: Flask, Django, FastAPI
- **Ruby**: Ruby on Rails, Sinatra
- **JavaScript/Node.js**: Express
- **C#/.NET**: ASP.NET Core, Web API
- **PHP**: Laravel, Symfony

These frameworks provide tools and libraries to simplify the process of building RESTful APIs, handling routing, request parsing, and response formatting.

For our example, we will use FastAPI, a modern web framework for building APIs with Python 3.7+ based on standard Python type hints. The web server used by FastAPI is Uvicorn, an ASGI[4] server implementation, using `uvloop` and `httptools`.

[4] ASGI (Asynchronous Server Gateway Interface). Click for more information.

Source code of the application

```python
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class User(BaseModel):
    id: int
    name: str
    email: str

users_db: dict[int, dict] = {
    1: {"id": 1, "name": "Alice", "email": "alice@example.com"},
    2: {"id": 2, "name": "Bob", "email": "bob@example.com"},
}
```

```python
@app.get("/users/{user_id}", response_model=User)
def get_user(user_id: int):
    user = users_db.get(user_id)
    if user:
        return user
    return {"error": "User not found"}

@app.get("/users", response_model=list[User])
def list_users():
    return list(users_db.values())

@app.post("/users", response_model=User)
def create_user(user: User):
    users_db[user.id] = user.dict()
    return user
```

```
python3 -m venv .venv
source .venv/bin/activate
pip install "fastapi[standard]"
fastapi dev api_server.py
```

```
INFO   Will watch for changes in these directories: ['...']
INFO   Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO   Started reloader process [13684] using WatchFiles
INFO   Started server process [13686]
INFO   Waiting for application startup.
INFO   Application startup complete.
```

Interacting with the API

You can interact with the API using tools like `curl`, Postman, or by writing client-side code in a programming language of your choice.

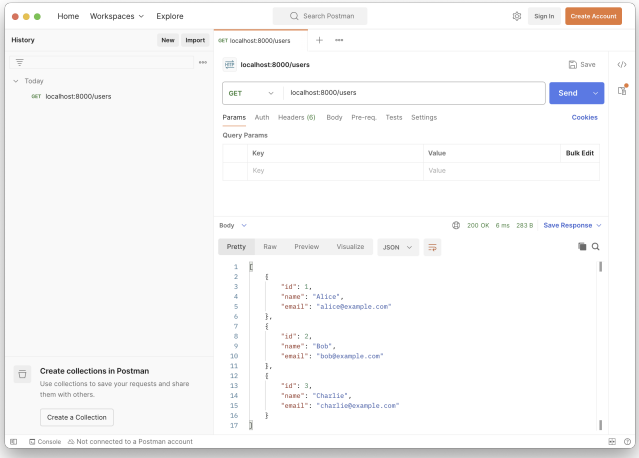
```
# Get all the users
curl localhost:8000/users/

# Get user with ID 1
curl localhost:8000/users/1
```

```
# Create a new user
curl -X POST localhost:8000/users \
-H "Content-Type: application/json" \
-d '{"id":3,"name":"Charlie","email":"
     charlie@example.com"}'
```

Notice that the API server is running on `localhost:8000`, which means it is accessible only from your local machine. In a production environment, the server would be hosted on a public IP address or domain name.

Also, GET requests don't seem to have the `Content-Type` header, which is correct, as they don't have a body. However, they should have an `Accept` header to indicate that the client expects a JSON response. Our web server is lenient and assumes JSON if the header is missing.

Postman


Home
Workspaces
Explore
Search Postman
Sign In
Create Account
History
New
Import
GET localhost:8000/users
Today
GET localhost:8000/users
localhost:8000/users
Save
GET
localhost:8000/users
Send
Params
Auth
Headers (6)
Body
Pre-req.
Tests
Settings
Cookies
Query Params
Key
Value
Bulk Edit
Key
Value
Body
200 OK
6 ms
283 B
Save Response
Pretty
Raw
Preview
Visualize
JSON
{
[
"id": 1,
"name": "Alice",
"email": "alice@example.com"
},
{
"id": 2,
"name": "Bob",
"email": "bob@example.com"
},
{
"id": 3,
"name": "Charlie",
"email": "charlie@example.com"
}
]
Create collections in Postman
Use collections to save your requests and share them with others.
Create a Collection
Console
Not connected to a Postman account

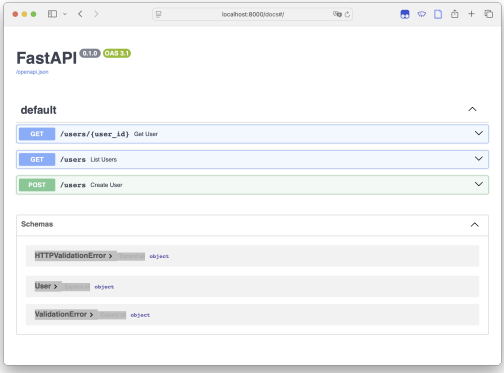
API Documentation

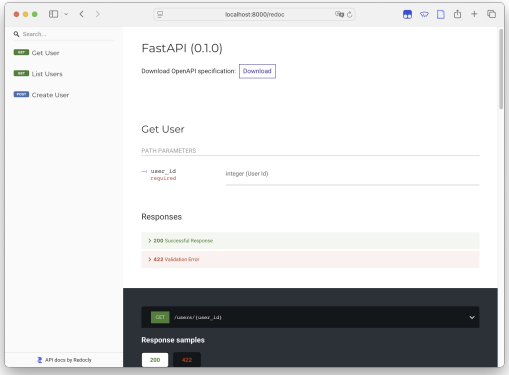
API documentation is essential for developers to understand how to use the API effectively. It typically includes:

- **Endpoint Descriptions**: Information about each API endpoint, including the URL, HTTP method, and purpose.
- **Request Parameters**: Details about the parameters that can be sent with requests, including their
- **Response Formats**: Examples of the data returned by the API, including success and error responses.
- **Authentication**: Information about how to authenticate and authorize requests, if applicable.
- **Examples**: Sample requests and responses to illustrate how to use the API.
- **Error Handling**: Information about common error codes and their meanings.
- **Rate Limiting**: Details about any limits on the number of requests that can be made in a given time period.

Back to our example

FastAPI automatically generates interactive API documentation using Swagger UI and ReDoc.





Try it at `http://localhost:8000/docs` and `http://localhost:8000/redoc`.

Summary of key points

- Networking is essential for modern computing, enabling communication, resource sharing, and access to information across the globe.
- The OSI and TCP/IP models provide frameworks for understanding network communication, and common network protocols include TCP/UDP, HTTP/HTTPS, FTP/SFTP, SMTP/IMAP/POP3, DNS, DHCP, and NAT.
- Networked applications can follow various architectures, such as client-server, peer-to-peer, cloud-based, edge computing, microservices, service-oriented architecture, event-driven architecture, message queuing, and hybrid approaches.
- REST APIs are a popular way to build networked applications, using standard HTTP methods and stateless communication. We can build it in many programming languages and frameworks, including Python with FastAPI.
- JSON is the most common data format for REST APIs.
- Tools `curl` can be used to interact with APIs, and proper documentation is important for developers to understand how to use the API effectively.

References

Tanenbaum, A. S., & Wetherall, D. J. (2011). *Computer Networks* (5th ed.). Pearson.

- Fundamental book on computer networking.

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley.

- Covers various architectural patterns, including those relevant to networked applications.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures* (Doctoral dissertation, University of California, Irvine).

- The original dissertation that introduced REST.
- Available at https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Further reading

- RESTful API Design - https://restfulapi.net/
- FastAPI Documentation - https://fastapi.tiangolo.com/
- HTTP Status Codes - https://developer.mozilla.org/en-US/docs/Web/HTTP/Status
- Understanding the OSI Model - https://www.cloudflare.com/learning/ddos/glossary/open-systems-interconnection-model-osi/
- TCP/IP Model Explained - https://www.cloudflare.com/learning/ddos/glossary/tcp-ip/