# Web Services

CS 346 Application Development

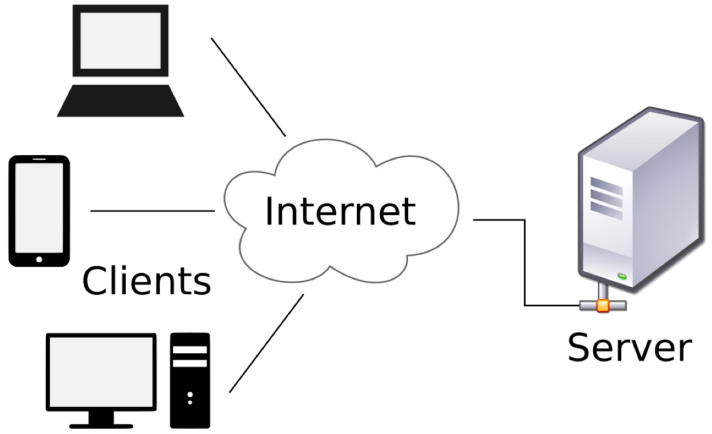
# What is a service?

A service is *software that provides runtime capabilities to other software*.

You already have **local** services on your computer that provide OS-level capabilities. e.g. printer, logging.

We're specifically going to talk about ***remote*** *services, running on a different computer, over a network.*

Most applications use remote services. Your online DB is just a service!




A server might service dozens or hundreds of other systems (clients).

# Benefits of adding more services

- **Data Sharing**. Services that are accessible over the internet allow us to share data between users, or between devices.
  - e.g., a TODO list showing up on my phone and desktop (device sharing, same user)
  - e.g., ability to share TODO items with other users (same device, multiple users).
- **Performance**. If designed correctly, distributing our work across systems can allow us to grow our system to meet high demand
  - e.g., Amazon “spins up” services as needed and shuts them down again when demand subsides.
- **Adding Capabilities**. There are actions that your application cannot perform on its own.
  - e.g., using GenAI to provide capabilities to your application.

# Networking Basics

Quick reminder of some concepts that you need to know.

# Introduction

How do client and server communicate?
We have protocols for packaging, transmitting and interpreting information.

- **Communication protocols**, for transferring data. These include HTTP for web traffic, FTP for sending files over a network, or TCP/IP for data packets.
- **Security protocols** are meant for establishing secure channels for the safe transmission of data. These include SFTP and SSH.
- **Network management protocols** are meant for controlling and managing devices on a network. These include SMNP and ICMP.

# TCP/IP

**TCP/IP** stands for Transmission Control Protocol/Internet Protocol and is a suite of communication protocols used to interconnect network devices on the internet. These two protocols work together:

- Transmission Control Protocol (TCP) defines how applications can create channels of communication across a network. It also manages how a message is assembled into smaller packets before they are then transmitted over the internet and reassembled in the right order at the destination address.
- Internet Protocol (IP) defines how to address and route each packet to make sure it reaches the right destination. Each gateway computer on the network checks this IP address to determine where to forward the message.

**What is TCP/IP?**

https://www.youtube.com/watch?v=614QGgw_FA4

# Addresses

For computers to communicate and transmit data between them, they need to be able to locate one another. How do they do this?

- Every device on a network requires a unique identifier, or IP address. This is usually presented as four numbers (0-255), dot separated.
  - e.g., 192.0.2.1.
- Addresses also include a port, which is an integer label assigned to a specific network channel on the receiver. e.g., 80 web, 21 ftp
- Syntax: address:port
  - e.g., https://www.google.com:80 ← web server monitoring port 80
  - e.g., https://192.168.1.20:1001 ← service monitoring port 1001

# Name Services/Caching

How does your application know the address of some other service?

- You can hard-code the IP address in your application (don't do this).
- Add an alias in your systems `/etc/hosts` file, to use a name locally.
- Rely on a name service like DNS that will let your computer "lookup" an IP address for a system by name – managed by your network

```
~ $ host google.com
google.com has address 142.251.41.78
google.com has IPv6 address 2607:f8b0:400b:804::200e
google.com mail is handled by 10 smtp.google.com.
```

# Web Services

What makes a service a “web service”?

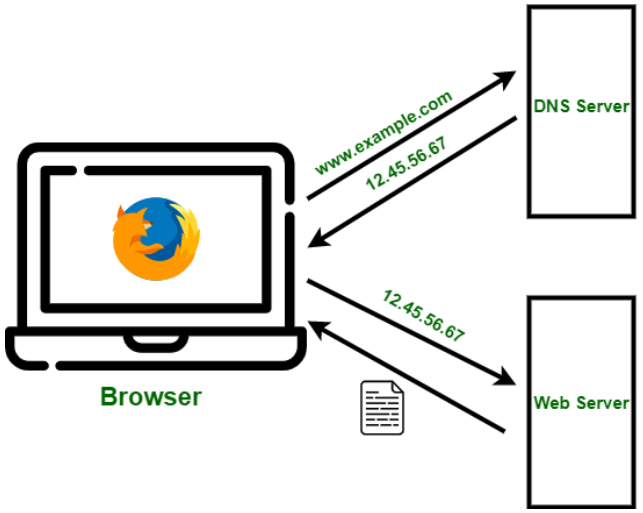
# Service Example: WWW

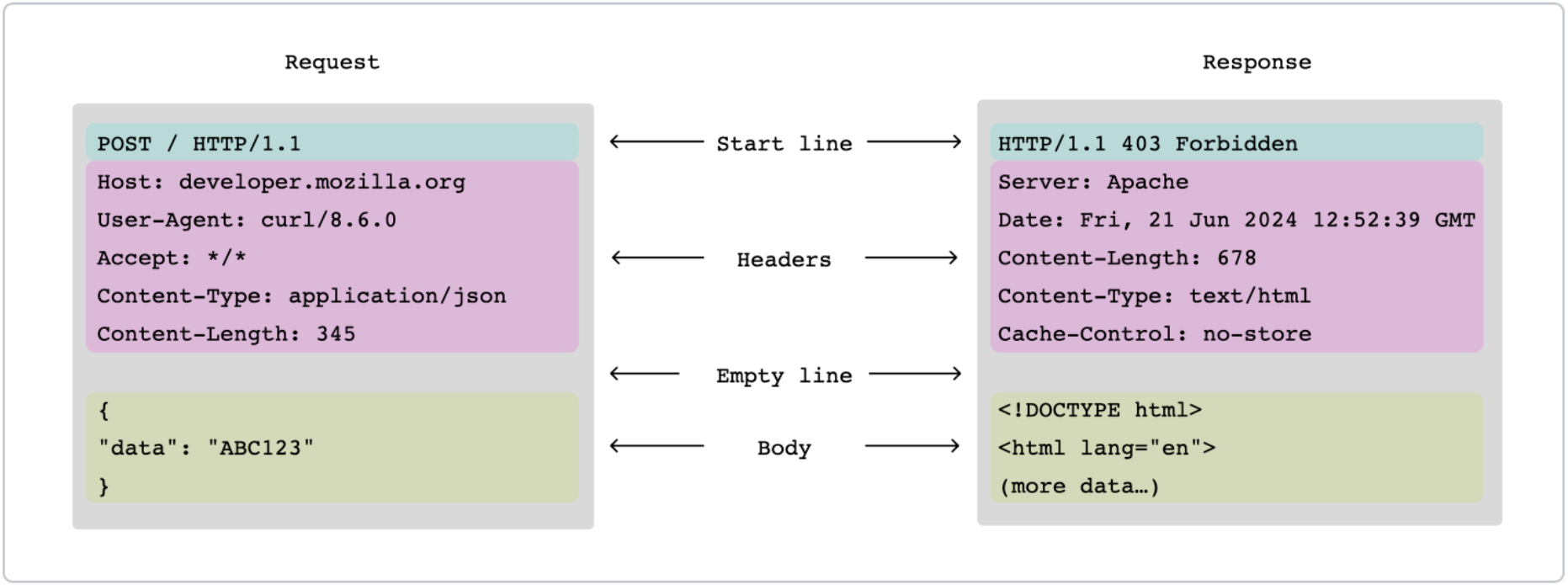
The world-wide-web is a great example of an early client-server (service) design.

A **web browser** (front-end) can request content from a **web server** (back-end) hosted on a different system.

Requests are sent using the Hypertext Transfer Protocol (HTTP).

- HTTP is a request–response model, where the client requests data from the server.
- The URL of a website describes the protocol, address, and document. e.g., https://uwaterloo.ca/about/






An example of a HTTP request/response. The request includes a header with information about the request, and a body that contains any required data. The response includes relevant header information and data, usually HTML or JSON.

GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/100.0.4896.127 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Accept-Language: en-US,en;q=0.9
Connection: keep-alive

<!doctype html>
<html lang="en">
<head><title>Example Domain</title>
<meta name="viewport" content="width=device-width, initial-scale=1">
<style>body{background:#eee;width:60vw;margin:15vh auto;font-family:system-ui,sans-serif}h1{font-size:1.5em}div{opacity:0.8}a:link,a:visited{color:#348}</style><body>
<h1>Example Domain</h1>
<p>This domain is for use in documentation examples without needing permission. Avoid use in operations.<p>
<a href="https://iana.org/domains/example">Learn more</a>
</body>
</html>

Another view of an HTTP response. Browsers don’t need syntax highlighting.

# HTTP Request Methods

The HTTP protocol supports the following types of requests aka “methods” :

- **GET:** The GET method requests that the target resource transfers a representation of its state. GET requests only retrieve data.
- **HEAD:** The HEAD method requests that the target resource transfers a representation of its state, like for a GET request, but without the data. Uses include checking if a page is available or finding the size of a file.
- **POST:** The POST method requests that the target resource processes the representation enclosed in the request according to the semantics of the target resource. e.g., post a message on a forum, or complete an online transaction.
- **PUT:** The PUT method requests that the target resource creates or updates its state with the state defined by the representation enclosed in the request.
- **DELETE:** The DELETE method requests that the target resource deletes its state.

# So, what are web services?

A **web service** is a simply a service that is built using web technologies, and which serves up content using web protocols and data formats.

- **We are using HTTP as the basis for a more generalized service protocol** that can serve up a broader range of data than just HTML.

A service can be written in almost any language. The web server e.g. Apache, nginx, handles the actual request and delegate work. This is just an extension of what web servers were originally designed to do.

- We are also leveraging the ability of web servers to handle HTTP requests efficiently, with the ability to scale to very large numbers of requests.

# REST

Representational State Transfer (REST), is a software architectural style that defines a set of constraints for how the architecture of an Internet-scale system, such as the Web, should behave.

- REST was created by Roy Fielding in his doctoral dissertation in 2000*.
- It has been widely adopted and is considered the standard for managing stateless interfaces for service-based systems.
- The term “RESTful Services” is commonly used to describe services built using standard web technologies that adheres to these design principles.

* Roy was also one of the principal authors of the HTTP protocol and co-founded the Apache server project.

# Key REST Principles

1. **Client-Server**. By splitting responsibility into a client and service, we decouple our interface and allow for greater flexibility.
2. **Layered System**. The client has no awareness of how the service is provided, and we may have multiple layers of responsibility on the server. i.e. we may have multiple servers behind the scenes.
3. **Stateless**. The service does not retain state i.e. it's idempotent. Every request that is sent is handled independently of previous requests. That does not mean that we cannot store data in a backing database, it just means that we have consistency in our processing.
4. **Cacheable**. With stateless servers, the client has the ability to cache responses under certain circumstances which can improve performance.
5. **Uniform Interface**. Our interface is consistent and well-documented. Using the guidelines below, we can be assured of consistent behaviour.

# HTTP Request

For your service, you define one or more **HTTP endpoints** (URLs). Think of an endpoint as a function - you interact with it to make a request to the server. e.g.,

https://localhost:8080/messages

https://cs.uwaterloo.ca/asis

To use a service, you format a request using one of these request types and send that request to an endpoint.

- **GET**: Use the GET method to READ data. GET requests are safe and idempotent.
- **POST**: Use a POST request to STORE data i.e. create a new record in the database, or underlying data model.
- **PUT**: A PUT request should be used to UPDATE existing data.
- **DELETE**: Use a DELETE request to delete existing data.

# HTTP Response

The HTTP response needs to return data in a machine-readable format, that can be transmitted as part of the response body (UTF-8).

JSON is often used as a standard data format.

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 51

{
  "status": "success",
  "message": "Data retrieved successfully"
}
```

An HTTP response. The body consists of a JSON object of key: value pairs.

# Using Services with Ktor

Accessing existing services or creating your own.

# What is Ktor?

Ktor is an application framework for building networked applications.

It's also developed and supported by JetBrains.

- https://ktor.io/docs/welcome.html
- https://ktor.io/docs/creating-http-apis.html#prerequisites

- You can use it for anything network and service related. e.g., fetch a web page; connect to a service using HTTP requests (GET, POST).
- You can use it on the client side (to make application requests) or to build a web service (which handles GET/POST requests for clients).

# Client: Simple Request

How do we make requests to a service? Kotlin includes libraries that allow you to structure and execute requests from within your application. e.g., fetches the results of a simple GET request:

```
val response = URL(“https://google.com").readText()
println(response);
```

```
<!DOCTYPE html>
<html lang="en" dir="ltr"><head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" /><meta name="ro
  <link rel="icon shortcut" href="/~cs346/1261/favicon.ico" sizes="32x32" />
<link rel="icon" href="/~cs346/1261/favicon.svg" type="image/svg+xml" id="favicon-svg" /
<link rel="icon" href="/~cs346/1261/favicon-16x16.png" type="image/png" sizes="16x16" />
<link rel="icon" href="/~cs346/1261/favicon-32x32.png" type="image/png" sizes="32x32" />
<title>Course Description – CS 346 Winter 2026</title>
  <meta name="description" content=" Description Introduction to full-stack application
```

# Client: Simple Request

The Ktor `HttpRequest` class uses a builder to let us supply as many optional parameters as we need. Here's a simple GET example:

```
fun get(): String {
    val client = HttpClient.newBuilder().build()
    val request = HttpRequest.newBuilder()
        .uri(URI.create("http://127.0.0.1:8080"))
        .GET()
        .build()

    val response = client.send(request, HttpResponse.BodyHandlers.ofString())
    return response.body()
}
```

# Client: Sending Data

Here's a POST method that sends an instance of our Message class to the service that we've defined and returns the response. We use serialization to encode it as JSON.

```
fun post(message: Message): String {
    val string = Json.encodeToString(message)


    val client = HttpClient.newBuilder().build();
    val request = HttpRequest.newBuilder()
        .uri(URI.create("http://127.0.0.1:8080"))
        .header("Content-Type", "application/json")
        .POST(HttpRequest.BodyPublishers.ofString(string))
        .build()


    val response = client.send(request, HttpResponse.BodyHandlers.ofString());
    return response.body()
}
```

```
data class Message(
    val id: Int,
    val msg: String
)
```

# Processing JSON

```
// The web API returns a response, where the body is a JSON object
// This is from the Domain lecture (Courses demo).

val result = CoroutineScope(IO).async {
    query("https://openapi.data.uwaterloo.ca/v3/ClassSchedules/$term/$courseID")
}

val response = result.await()

if (response.status == HttpStatusCode.OK) {
    val body = response.body<String>()
        Json.decodeFromString<List<SectionDto>>(body)
    } else {
        emptyList<SectionDto>()
    }
}
```

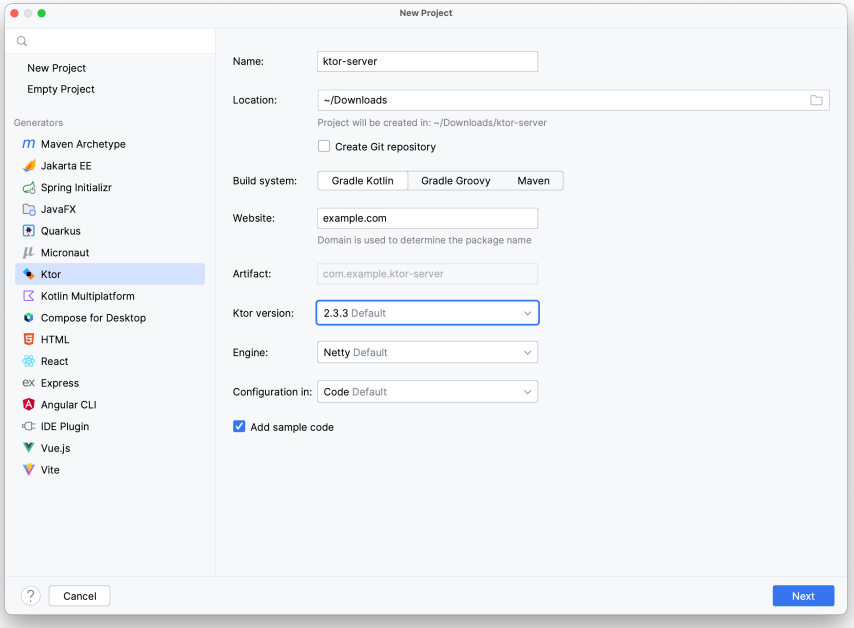
```
You need a data class with
fields that match the API

@Serializable
data class SectionDto(
var courseId: String,
var courseOfferNumber: Int,
var sessionCode: String,
var classSection: Int,
var termCode: Int,
// other fields
)
```

# Creating a Ktor Project (Server)

The Ultimate edition has support for Ktor.

- Install the Ktor plugin
- New project wizard
- Ktor
  - Ktor version: 3.0
  - Engine: Netty
  - Configuration: code
  - Add sample code (check)

- Plugins
  - Routing (*required*)
  - kotlinx.serialization (for JSON payloads)
  - Websockets (bidirectional communication)
  - Authentication - see later section



# Server: Main Method

The main method launches a specific web server (Netty below) and starts listening at the IP address and port listed. For debugging, this corresponds to 127.0.0.1:8080.
e.g.,

```
import io.ktor.server.engine.*
import io.ktor.server.netty.*
import com.example.plugins.*


fun main() {
    embeddedServer(Netty, port = 8080, host = "0.0.0.0") {
        configureSerialization()
        configureRouting()
        configureWebsockets()
    }.start(wait = true)
}
```

# Server: plugins/Routing.kt (1/2)

Standard routing is meant for handling HTTP requests (GET, PUT, POST, DELETE) that are initiated by the client. You create routes for each end point that you want to support.

```
import io.ktor.server.application.*
import io.ktor.server.response.*
import io.ktor.server.routing.*

fun Application.configureRouting() {

    routing {
        get("/") {
            call.respondText("Hello World!")
        }
    }
}
```

/ endpoint

# Server: plugins/Routing.kt (2/2)

Standard routing is meant for handling HTTP requests (GET, PUT, POST, DELETE) that are initiated by the client. You create routes for each end point that you want to support.

```
import io.ktor.server.application.*
import io.ktor.server.response.*
import io.ktor.server.routing.*

fun Application.configureRouting() {

    routing {
        get("/customer/{id}") {
            val id = call.parameters["id"]
            val customer: Customer = customerList.find { it.id == id!!.toInt() }!!
            call.respond(customer)
        }

    }
}
```

/ endpoint

https://github.com/ktorio/ktor-samples

# Server: WebSockets.kt

WebSocket is a protocol which provides a full-duplex communication over a single TCP connection. This is useful when you want to maintain a connection and allow either client or server to send data (e.g. data on the server changes and you want to notify clients).

```
routing {
    webSocket("/echo") {
        send("Please enter your name")
        for (frame in incoming) {
            frame as? Frame.Text ?: continue
            val receivedText = frame.readText()
            if (receivedText.equals("bye", ignoreCase = true)) {
                close(CloseReason(CloseReason.Codes.NORMAL, "Client said BYE"))
            } else {
                send(Frame.Text("Hi, $receivedText!"))
            }
        }
    }
}
```

https://ktor.io/docs/websocket.html#websocket-api