Web Services

CS 346 Application Development

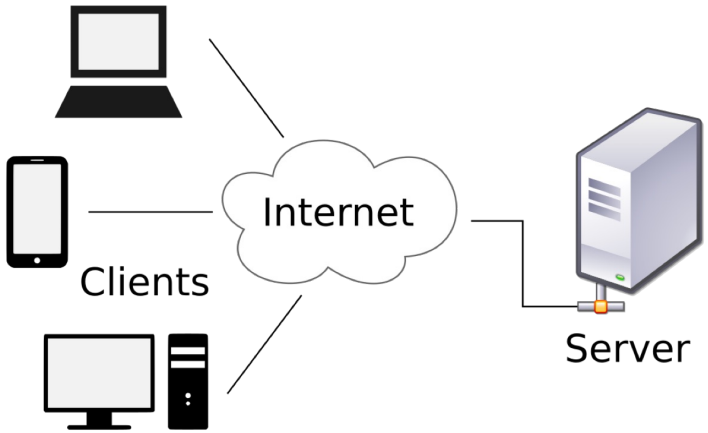
What is a service?

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 spooler, services for logging etc.

We're specifically going to talk about ***online services*** – *those running on a different computer over a network*.

Many applications work with online services to share data or communicate with other applications. e.g. Twitter or email client; Windows update; Overwatch.

Like a local service, these exist to provide services to other software.



Why online services?

The benefits of moving services online are significant:

- **Resource sharing**. We often need to share resources or data across users. e.g. a shared spreadsheet. Support for online resources supports this.
- **Reliability**. By moving critical data and computation to a central controlled location, we can manage that data more securely. We also can potentially support failover scenarios e.g., redirect clients in case of a failure.
- **Parallelization**. It can also be cheaper to spread computation across multiple systems instead of relying on a single local system. Distributed architectures provide flexibility dynamically allocate hardware as needed based on performance characteristics.
- **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 when demand subsides.

How does this affect your application?

We can split computation across systems, based on our specific needs:

Client (local)

- Contains most of the application logic.
- Fetches data or posts data that needs to be shared.

Service (remote)

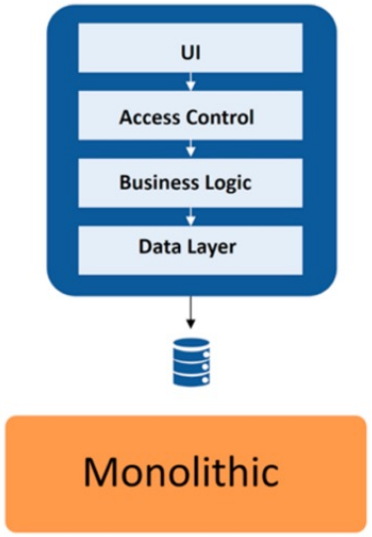
- Accessible to every client (i.e. each client knows how to connect to the server, but not necessarily how to connect to other clients)
- Focused on computation where the results need to be shared with one or more clients. Might also be used to handle sensitive data.
- A shared database could also fit this deployment model (we tend to reserve the word “service” for systems that do more than store data).

1/ Client-Server model

Client-server architectures split processing into front-end and back-end pieces. This is also called a **two-tier architecture**. Tiers represent a ***physical*** separation of concerns.

e.g., web browser/server.

The service just acts as a resource. Over time, architectures moved towards offloading more and more computation to the service.

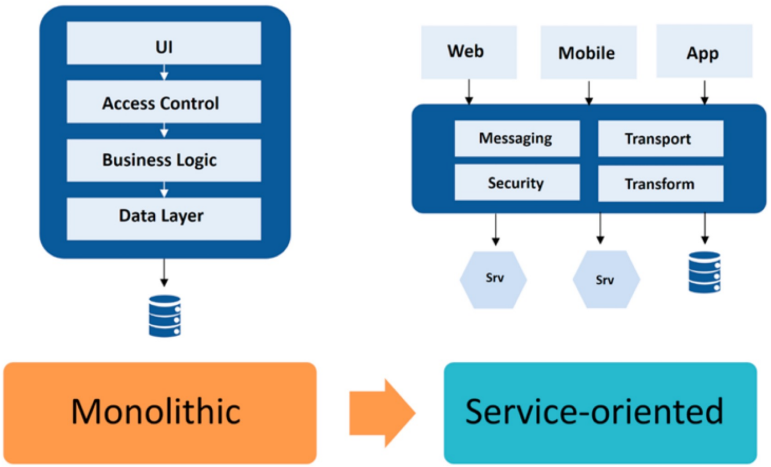


2/ Service Architecture

A services-based architecture splits functionality into small "portions of an application”.

- Each service is independent and separately deployed.
- Each service provides coarse-grained domain functionality.
- Client communicates with each service using some lightweight protocol.
- Services may share data via a database.

e.g. a service might handle a customer checkout request to process an order, managing the entire transaction.



https://maveric-systems.com/blog/microservices-i-microservices-vs-soa/

How do we make this work?

There are lots of complexities to building a system like this. We need to address fundamental design challenges like:

1. How do the service and client “locate” each other on the network?
   - We assume a name service like DNS exists.
   - We might need to “know” the port number to connect to on each system.
2. How do we ensure that data is only sent to the “correct” clients?
   - We need a secure authentication mechanism e.g., username/password.
3. How do the client and server communicate?
   - We have many different protocols that could be used e.g., TCP/IP, FTP.
   - Current “standard” is HTTP as a lightweight protocol, when possible.

Services > Communication

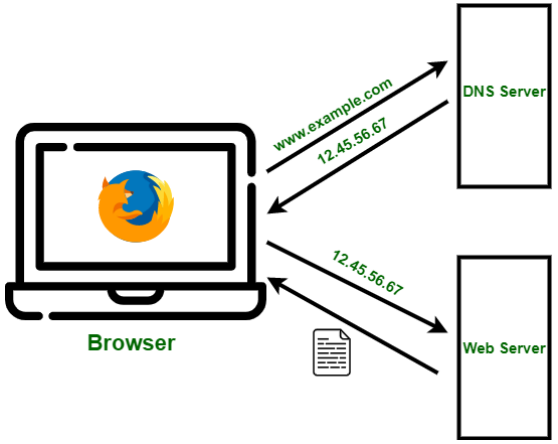
Web protocols in use.

Web architecture

The web is a great example of an early client-server design.

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

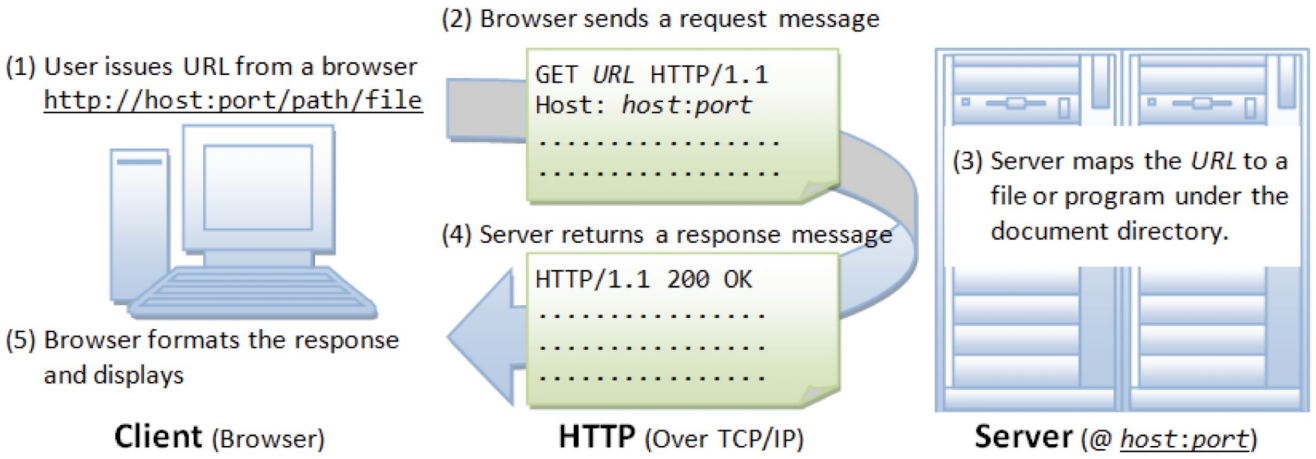
- Requests were sent over the network using a standard request format.
- Results were sent back synchronously.
  - i.e., the web browser "blocks" waiting for content to be served.
- The web server is stateless
  - i.e., each request is served in isolation (ways around this evolved over time).



Request format

The Hypertext Transfer Protocol (HTTP) is an application layer protocol that supports serving documents, and processing links to related documents from a remote service. It's the primarily mechanism for the browser and server to communicate.

HTTP is a request–response model: the client requests data from the server.



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 should only retrieve data and should have no other effect.
- **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 shopping 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.

HTTP Request Format

An HTTP request consists of:

- An endpoint i.e., the URL describing the full path to a service.
- An HTTP method e.g., GET, PUT.
- A header
  - Contains metadata about the request
  - e.g., token authorizing access, or a cookie
- The request body
  - A block of text (often in JSON!)

Web Services

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

- A web service responds to HTTP requests! **We can can 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, handle 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. To do this, we need some guidelines on how to structure HTTP for generic 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.

Request Methods

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.

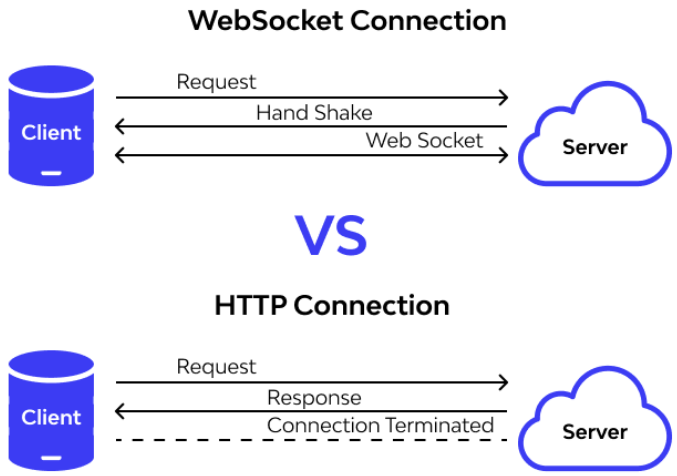
API Guidelines

1. When defining endpoints, use **nouns** over verbs, **plural** over singular form. e.g.
   - GET /customers should return a list of customers
   - GET /customers/1 should return data for Customer ID=1.
2. Use JSON as the data format. i.e. send and receive JSON objects.
3. Be consistent
   - If you define a JSON structure for a record, you should always use that structure: avoid doing things like omitting empty fields (instead, return them as named empty arrays).

Communication Models

For maximum flexibility, we need to support two different models:

- **Bidirectional** communication: either one can initiate, and the other responds.
- **Unidirectional** communication: the client initiates and the service responds.



Bidirectional: can be initiated by either client or server.

Unidirectional: must be initiated by the client.

Using services with Ktor

Accessing 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: 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()
```

The `HttpRequest` class uses a builder to let us supply as many optional parameters as we need:

```
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.

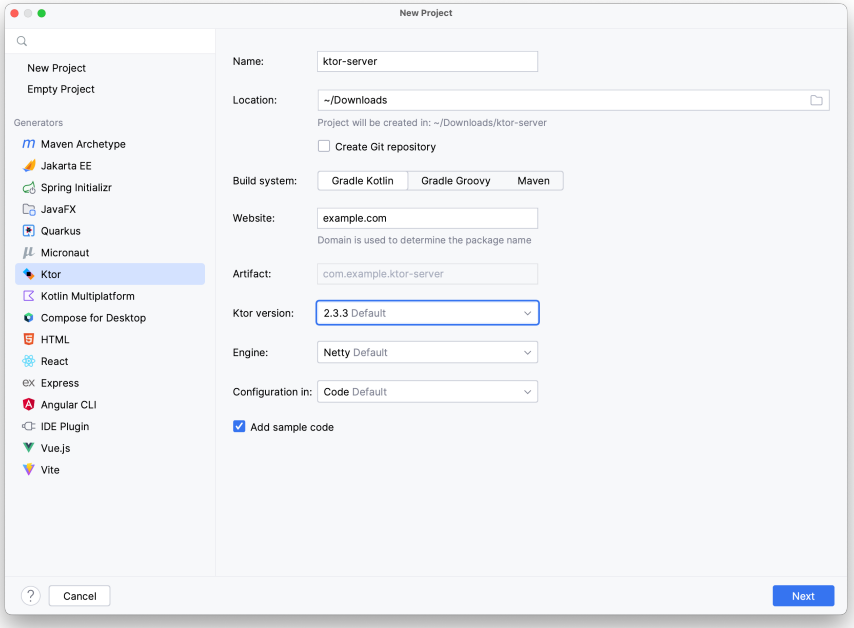
```
fun post(message: Message): String {
    val string = Json.encodeToString(message)


    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()
}
```

Creating a Ktor Project

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