BUILDING WEB SERVICES

CS 346: Application Development

WHAT IS A SERVICE?

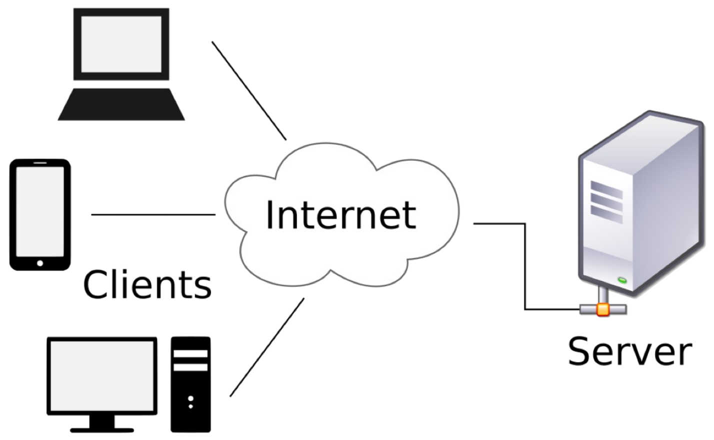
A service is *software that provides runtime capabilities to other software.*

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; WhatsApp.



WHY ONLINE SERVICES?

The advantages of this architecture are significant:

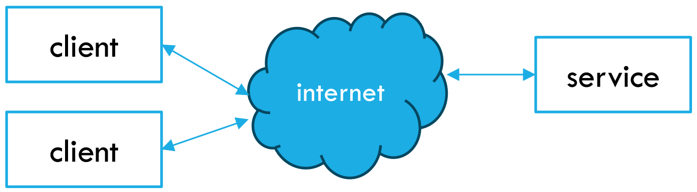
- **Resource sharing.** We often need to share resources or data across users. e.g. customer data is shared across the company. A shared online resource supports this.
- **Reliability.** By moving critical data and computation to a central controlled location, we have a greater ability to manage that data securely. We also can potentially support failover scenarios (i.e. redirect clients to a redundant service 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 to align the processing requirements with our needs if they change. i.e., allocate more hardware as needed for a high-performance task.
- **Performance.** If designed correctly, distributing our work across multiple systems can allow us to grow our system to meet high demand. Amazon for example, needs to ensure that their systems remain responsive, even in times of heavy load (e.g. holiday season). They do this by "spinning up" services as needed and shutting them down again when demand subsides.

HOW DOES THIS AFFECT YOUR APPLICATION?

We split computation across local and remote systems.

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. A ”staging area” for client data. Might also be used to handle sensitive data.
- A shared database could also fit this deployment model (but we tend to reserve the word “service” for systems that do more than store data).

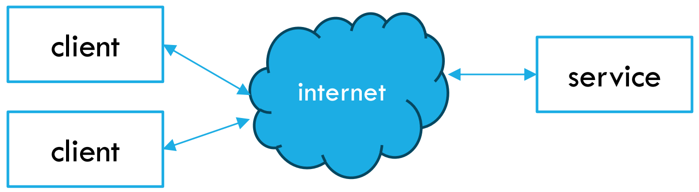
DISTRIBUTED: CLIENT-SERVER

Client-server architectures split processing into front-end and back-end pieces. This is also called a **two-tier architecture**. Unlike layers, which are logical, tiers represent a ***physical*** separation of concerns.

Common two-tiered systems:

- Application (front-end) and database (back-end),
- Web browser (front-end) and web server (back-end).

Three-tier architectures were also popular in the 1990s and 2000s, which included a middle business-logic tier (often a business object/transaction layer that sat in front of a database).



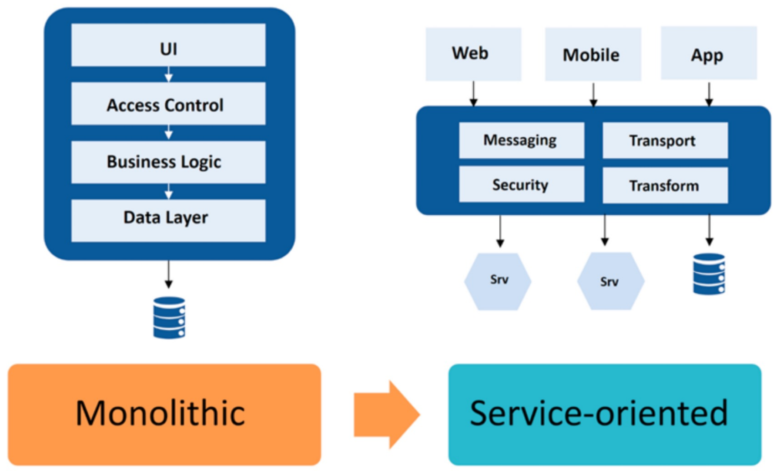
A traditional client-server model allows you to share expensive resources e.g. database. It provides a specific "source of truth for data".

DISTRIBUTED: SERVICE

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

- Each service is independent and separately deployed (i.e. a separate application).
- Each service provides coarse-grained domain functionality.
- Client application 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; this could be processed in its entirely by one service, as a single transaction.



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

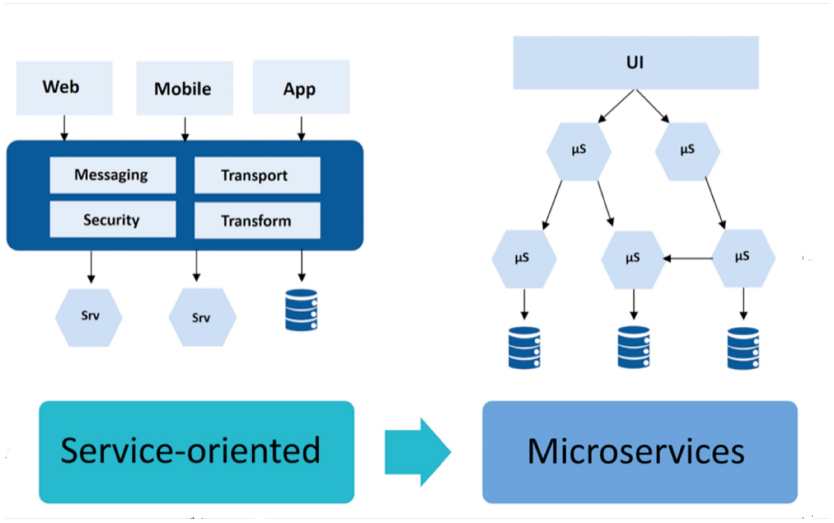
DISTRIBUTED: MICROSERVICES

A microservices architecture is a collection of *loosely coupled services*.

- Services are organized around business capabilities i.e. they provide specialized, domain-specific services to applications.
- Services are small, decentralized, and independently deployable.
- Each micro-service operates independently.

What makes them different from services?

- We allow redundant services! This provides scalability e.g., to handle increased demand or component failures.
- They are smaller and can coordinate tasks.



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 (name lookup returning an IP address).
   - We might need to "know" the port number to connect to on each system.
2. How do we identify the user so 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.

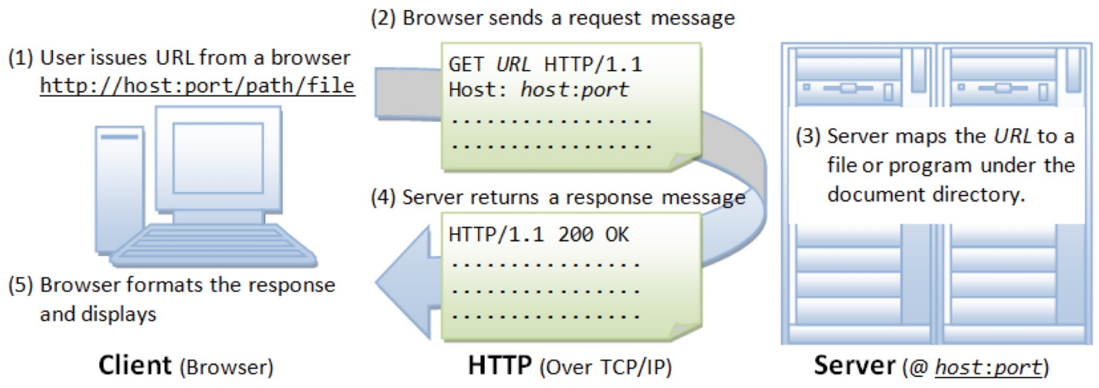
SERVICES > COMMUNICATION

CS 346: Application Development

HYPERTEXT TRANSFER PROTOCOL (HTTP)

The Hypertext Transfer Protocol (HTTP) is an application layer protocol that supports serving documents, and processing links to related documents from a remote service. HTTP is a request–response protocol, where the client requests data from the server. e.g.,

- The web browser requests content, which results in an HTTP request being sent to the server.
- The server returns a response message to the client that includes status, and data.


(1) User issues URL from a browser http://host:port/path/file
(2) Browser sends a request message
GET URL HTTP/1.1
Host: host:port
(3) Server maps the URL to a file or program under the document directory.
(4) Server returns a response message
HTTP/1.1 200 OK
(5) Browser formats the response and displays
Client (Browser)
HTTP (Over TCP/IP)
Server (@ host:port)

HTTP REQUEST 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 representation data enclosed in the response body. Uses include looking whether a page is available through the status code, and quickly finding out 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. For example, it is used for posting a message to an Internet forum, or completing 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.

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 principle 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. Examples:

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 (1/2)

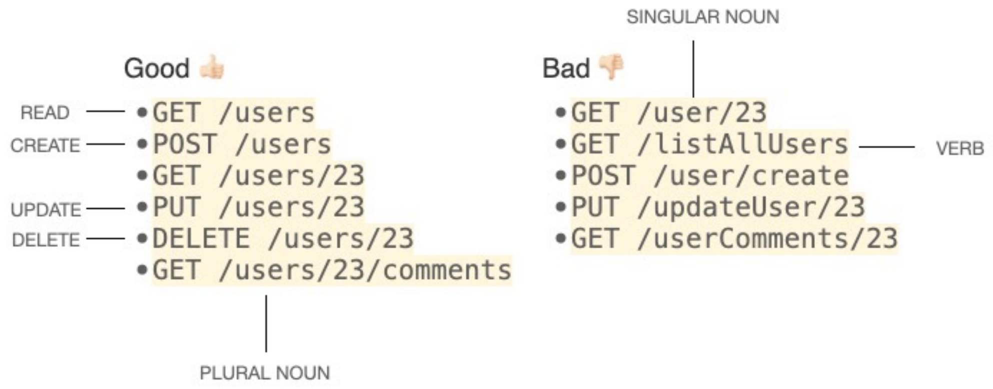
Here's some guidelines on using REST to create a web service [Cindrić 2021].

1. When defining endpoints, use **nouns** instead of verbs and use **plural** instead of 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 e.g. if you make a POST request, use JSON data structure as the payload.
    - It's easier to use, read and write, and it's faster than XML. Every meaningful programming language and toolkit already supports it.

API GUIDELINES (2/2)

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).



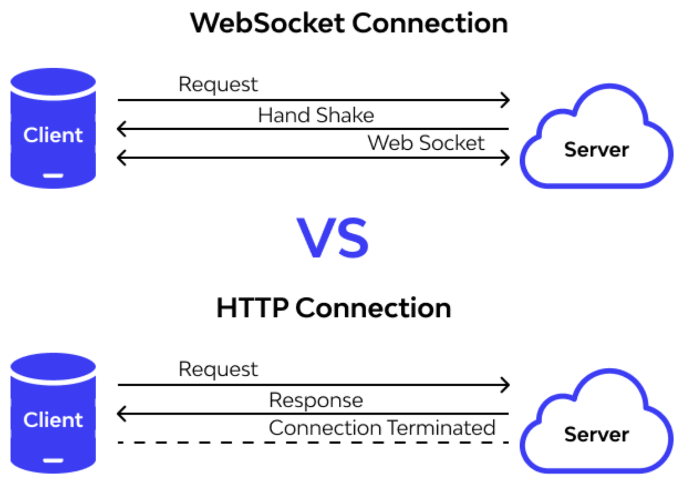
```
• GET /users
• POST /users
• GET /users/23
• PUT /users/23
• DELETE /users/23
• GET /users/23/comments
```

```
• GET /user/23
• GET /listAllUsers
• POST /user/create
• PUT /updateUser/23
• GET /userComments/23
```

COMMUNICATION MODELS

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

- **Bidirectional** communication: either one can initiate, and the other responds to the request.
- **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.

SERVICES > KTOR

CS 346: Application Development

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 almost 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).

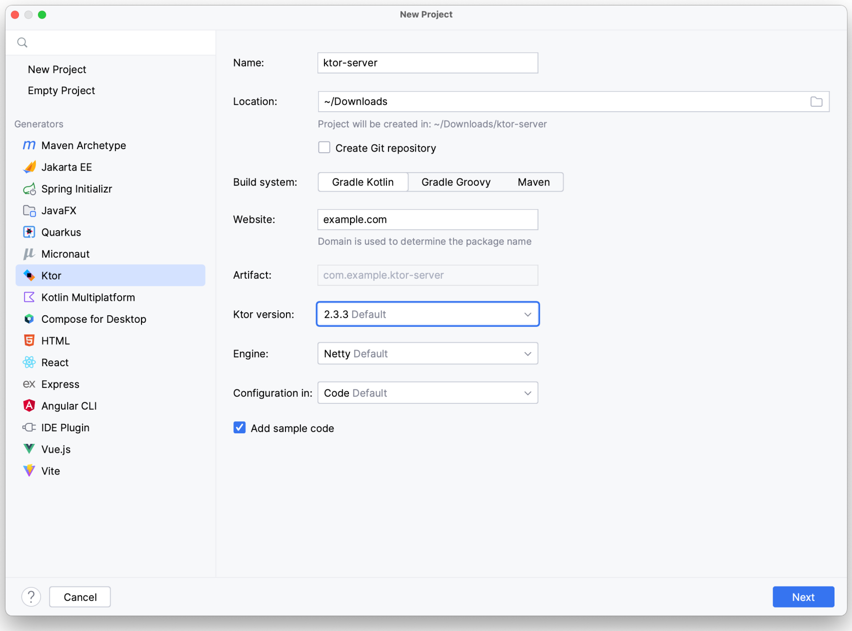
CREATING A KTOR PROJECT

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



Name: ktor-server

Location: ~/Downloads

Project will be created in: ~/Downloads/ktor-server

☐ Create Git repository

Build system: [✓ Gradle Kotlin] Gradle Groovy Maven

Website: example.com

Domain is used to determine the package name

Artifact: com.example.ktor-server

Ktor version: 2.3.3 Default

Engine: Netty Default

Configuration in: Code Default

☑ Add sample code

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.

```kotlin
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.

```kotlin
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

CLIENT: REQUEST

How do we actually use our service? Kotlin (and Java) 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()
}
```

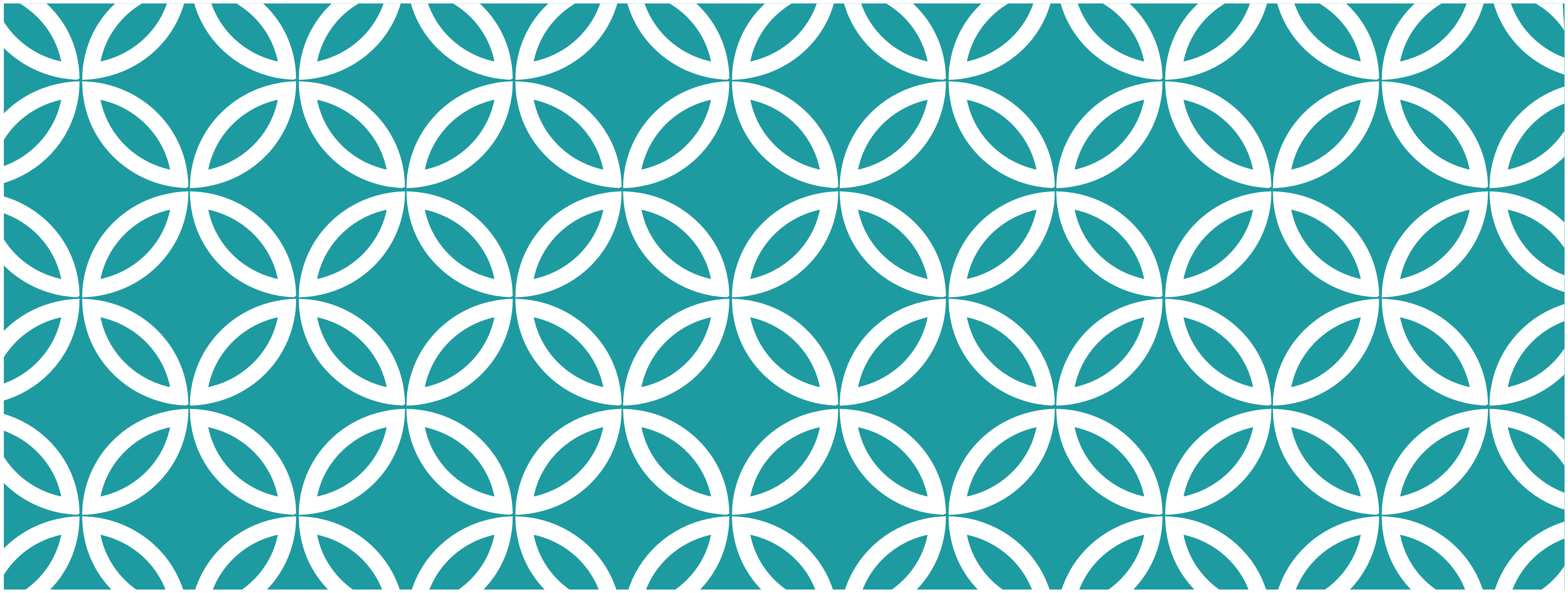
CLIENT: SENDING DATA

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()
}
```



AUTHENTICATION

CS 346 Application Development

AUTHENTICATION/AUTHORIZATION

If you are going to store data on a remote system, you need to be concerned about restricting access to that data!

- **Authentication**: Verifying the identity of a person or system (e.g., username/password).
- **Authorization**: Process where the service determines if the verified user should have access to a resource (and what that access should be). e.g., managing permissions on a database record, or a stored file.

Ktor provides the Authentication plugin to handle authentication and authorization.

Typical usage scenarios include logging in users, granting access to specific resources, and securely transmitting information between parties. You can also use Authentication with Sessions to keep a user's information between requests.

TYPES OF AUTHENTICATION

HTTP

- Basic - use Base64 encoding to provide a username/password. Requires HTTPS to be secure.
- Digest - applies a hash function to the username/password to encrypt before sending.
- Bearer - an authentication scheme that involves security tokens called bearer tokens. The Bearer authentication scheme is used as part of OAuth or JWT, but you can also provide custom logic for authorizing bearer tokens.

OAuth

- OAuth is an open standard for securing access to APIs. The `oauth` provider in Ktor allows you to implement authentication using external providers such as Google, Facebook, Twitter.

Session

- Sessions provide a mechanism to persist data between different HTTP requests. Typical use cases include storing a logged-in user's ID, the contents of a shopping basket.

NEXT STEPS

The Ktor documentation has a *huge* collection of samples.

https://ktor.io/docs/welcome.html

If you are building your own service, we highly recommend reading through the Ktor documentation and reviewing some of these examples.

There are other popular web service solutions e.g., Spring Framework but Ktor is recommended as a well-designed, Kotlin-specific framework.

What if you don't want to build a web service and just want to leverage existing solutions? See the next section on **Cloud Computing.**