1 HTTP

Definition : HTTP is a protocol used for transferring data over the Web,. It operates on a client-server model where a client (user agent) sends a request to a server, which processes it and returns a response.

Underlying Protocols : Operated on TCP (until HTTP/2) and shifted to UDP (from HTTP/3).

URL Components : Protocol, host, optional port, resource path, and optional query parameters.

Additional Elements : Path parameters, subdomains.

URL Encoding : Essential for safe transmission of data with special characters (e.g., space becomes

GET : Fetches data from the server. Idempotent (multiple requests result in the same outcome).

POST : Submits data to the server to create or update a resource. Non-idempotent.

PUT : Replaces all current representations of the target resource with the request payload. Idempotent.

PATCH : Partially modifies a resource, unlike PUT which replaces entirely. Non-idempotent.

DELETE : Removes a specified resource. Idempotent.

1.1 Format

Request : Start Line – Contains the HTTP method, URI, and HTTP version, headers, body

Response : Status Line : Includes the HTTP version, status code, and status message, headers, body

	HTTP/1.1 200 OK Date : Mon, 27 Nov 2023 17:42:47 GMT Server : Apache Last-Modified : Thu, 23 Feb 2023 15:00:12 GMT ETag : "17df-5f55f450264dd" Accept-Ranges : bytes Content-Length : 6111 Vary : Accept-Encoding X-Content-Type-Options : nosniff X-Frame-Options : sameorigin Content-Type : text/html ; charset=ISO-8859-1 {EMPTY LINE} {BODY}
GET / HTTP/1 Host : gaps.heig-vd.ch User-Agent : curl/8.1.2 Accept : */* {EMPTY LINE}	

1.2 Response status codes

1xx (Informational) : Temporary responses indicating the client should continue the request (e.g., 100 Continue).

2xx (Success) : Indicates that the client's request was accepted (e.g., 202 Accepted, 204 No Content for successful requests that don't return data).

3xx (Redirection) : Indicates further action needed to complete the request (e.g., 303 See Other for redirecting with a GET request).

4xx (Client Error) : Errors due to invalid requests from the client (e.g., 406 Not Acceptable, 408 Request Timeout).

5xx (Server Error) : Failure of the server to fulfill a valid request (e.g., 503 Service Unavailable, 507 Insufficient Storage).

1.3 Advanced Parameters

Path Parameters : Variables within the endpoint path, e.g., /users/userId for user-specific operations.

Query Parameters : Key-value pairs appended to the URL with ?, used for filtering, searching, or sorting.

Body : Essential for POST, PUT, PATCH methods, containing data like JSON, XML, or form data.

1.4 HTTP Headers and Content Negotiation

Content Negotiation : Allows clients to specify the format of the response they wish to receive via the `Accept` header.

HTTP **does not transfer objects**, it **transfers representations of objects**. This means that the server can send the same resource in different representations.

1.4.1 Important headers

Content-Type : Specifies the media type of the resource.

Cache-Control : Directives for caching mechanisms in both requests and responses.

1.5 HTTP Sessions & State Management

Statelessness of HTTP : HTTP itself doesn't retain user state between requests.

Session Management : Typically handled via cookies or tokens, providing a way to persist user state across requests.

1.6 Advanced API Design

RESTful Principles : REST APIs should be stateless and resource-based. Each resource is identified by URIs and manipulated through HTTP methods

1.6.1 Principles

1. **Client / server architecture** : client and server are completely separated and only interact through the API
2. **Stateless** : the server does not retain any session information. Requests from the client must include all the information necessary to process it.
3. **Cache-ability** : a REST API should support caching of responses by the client and control which responses can be cached and which not.
4. **Layered system** : it should be able to add intermediate systems (cache, load balancer, security gateway) without any impact for the client
5. **Uniform interface** : Use URIs/URLs to identify resources
   Server responses use a standard format that includes all information required by the client to process the data (modify or delete the resources state). Server responses include **links** that allow the client to **discover how to interact** with a resource
6. **Code on demand (optional)** : responses may include executable code to customize functionality of the client

1.7 Performance Optimizations

Caching : Implementing HTTP caching strategies to reduce load times and server load.

Compression : Using gzip or similar to reduce the size of the payload.

Connection Management : Utilizing HTTP/2 for improved performance through features like multiplexing and server push

1.8 (Non-)Functional Requirements

Functional : User, product, order, payment management, etc.

Non-Functional : Response time, throughput, scalability, availability, maintainability, security.

1.9 Web Infrastructure Components

Web Server : Handles HTTP requests.

Reverse Proxy : Manages requests for multiple servers.

Load Balancer : Distributes traffic across servers.

Cache : Stores data for faster future access.

CDN : Network of servers for content delivery.

1.10 The 'Host' Header

Allows multiple domains on the same IP with help of proxy. Key for reverse proxy functionality.

1.11 Proxy Types

Forward Proxy : Between clients and external systems.

Reverse Proxy : Between clients and servers, provides security and load balancing. Can be used to load balance, cache , encrypt/decrypt traffic, protect servers from attack, serve static content from a cache, serve multiple domains on the same IP

1.12 System scalability

Vertical Scaling (Scaling Up) : Add resources to a single server. Limited by hardware

Horizontal Scaling (Scaling Out) : Add more servers. Limited by soft

1.13 Load Balancing Strategies

Round-Robin : Distribute requests evenly.

Sticky Sessions : Bind client sessions to specific servers.

Least Connections : Direct to server with fewest active connections.

Least Response Time : Choose the fastest responding server.

Hashing : Based on request attributes like IP or URL.

1.14 Caching Mechanisms

Client-Side : Browser caches response.

Server-Side : Server or proxy caches data.

Expiration Model : Cache for a set duration (Cache-Control : max-age).HEADER : Cache-Control : max-age=<number of seconds>

Validation Model : Cache until data changes (Last-Modified/ if modified, ETag/If-None-Match).

1.15 CDN

Type of cache that can be used to serve static content to clients.

Geographically distributed network of proxy servers and their data centers. → improve the performance of a system by serving static content to clients from the closest server.

Best way to cache is to cache at different levels and combine the technique

2 JAVA I/Os

2.1 Basic concepts

Binary Data : Direct representation of data in bits ; does not need interpretation.

Text Data : Represents characters, requiring specific encoding to interpret.

2.2 Character encodings

ASCII : 7-bit character set for English.

Extended ASCII : Includes code pages like ISO-8859-1 (Latin-1), Windows-1252.

Unicode : Supports all languages and characters ; UTF-8, UTF-16 are implementations.

UTF-8 : Variable-length (1-4 bytes per character), backward compatible with ASCII.

2.3 End-of-line characters

Unix/Linux/macOS : '\n' (LF)
Windows : '\r\n' (CR+LF)

2.4 Byte order

Little Endian : Least significant byte first.

Big Endian : Most significant byte first. Java default.

2.5 Java Basics

Sources and Sinks : Abstractions for data origins (InputStream) and destinations (OutputStream).

Streams : Channels to read (Reader) or write (Writer) data.

2.6 Java classes

`FileInputStream`, `FileOutputStream` for binary files.
`InputStreamReader`, `OutputStreamWriter` for character files.
`BufferedReader`, `BufferedWriter` for efficient text reading/writing.
`BufferedInputStream`, `BufferedOutputStream` for buffered binary IO.

Buffer Size : Affects performance ; larger buffers may improve throughput but increase memory usage.

```
class StreamReaderWriterExample {
public static void main(String[] args)
throws IOException {
  BufferedReader reader = new BufferedReader(
    new InputStreamReader(
      new FileInputStream("text"),
      StandardCharsets.UTF_8
    )
  );

  BufferedWriter writer = new BufferedWriter(
    new OutputStreamWriter(
      new FileOutputStream("text"),
      StandardCharsets.UTF_8
    )
  );

  int c;
  while ((c = reader.read()) != -1) {
    writer.write(c);
  }

  writer.flush();
  writer.close();
  reader.close();
}
}
```

3 Application protocol

Layers : Application protocols sit on top of transport (TCP/UDP) and network (IP) protocols.

Versions : Protocols can have multiple versions (e.g., HTTP/1.1, HTTP/2).

3.1 Structure of Application Protocols

Messages : Defined exchanges between client and server (e.g., SMTPs HELO, MAIL FROM:).

Format : Specific syntax for each message type.

Sequence : Order of message exchange, often illustrated with sequence diagrams.

Edge Cases : Defined behavior for unexpected or error conditions.

3.2 Defining an Application Protocol

3.2.1 Overview

Describe the protocols purpose and basic operation.

3.2.2 Transport Protocol

Specify underlying protocol (TCP/UDP) and port(s).

3.2.3 Messages

Define messages/actions available to clients and servers. Specify message format and encoding (commonly UTF-8).

Examples : Provide sequence diagrams or examples to illustrate use.

3.2.4 Reserved Ports

Well-known Ports : 0-1023, require privileges on Unix systems.

Registered Ports : 1024-49151, for user or vendor applications.

Dynamic/Private Ports : 49152-65535, for private or temporary services.

Key Ports and Protocols : FTP : 20, 21, SSH : 22, SMTP : 25, 465, 587, DNS : 53, HTTP/HTTPS : 80, 443, POP3 : 110, 995, IMAP : 143, 993

3.2.5 Design Considerations

Purpose : Clearly define what the protocol aims to achieve.

Transport : Choose TCP for reliability or UDP for speed.

Messages : Carefully design message formats for clarity and efficiency.

Error Handling : Specify responses for all possible error scenarios

Extensibility : Consider future changes or extensions to the protocol.

3.3 Protocol example

Overview

Purpose : Allow clients to securely transfer files from a server.

Operation : Client-server model where clients request files and servers respond with file data or errors.

Transport Protocol

Underlying Protocol : TCP for reliable communication.

Default Port : 1155 (chosen arbitrarily and not registered).

Messages

Client Messages :

CONNECT : Initiate a connection to the server. Format : CONNECT

GET <filename> : Request a file from the server. Format : GET filename.txt

QUIT : Close the connection. Format : QUIT

Server Messages :

WELCOME : Acknowledge connection establishment. Format : WELCOME

FILE <filename> <size> <data> : Send requested file content. Format : FILE filename.txt 1024 [binary data]

ERROR <code> <message> : Indicate an error. Format : ERROR 404 File Not Found

BYE : Acknowledge connection termination. Format : BYE

4 Docker

Images vs. Containers : An image is the blueprint, while a container is an instance created from that blueprint.

Networks : Containers can communicate with each other through Docker-defined networks.

Docker Hub : A cloud-based registry service for building and shipping containerized applications.

Image : A read-only template with instructions for creating a Docker container.

Container : A runnable instance of an image, encapsulating an application and its dependencies.

Dockerfile : A script containing instructions for building an image.

Registry : A storage and content delivery system for managing Docker images. Docker Hub is the default.

Volume : A mechanism for persisting data generated by and used by Docker containers.

4.1 Dockerfile instructions

`FROM [image-name]:[tag]` : Sets the base image.

`RUN [command]` : Executes a command.

`CMD ["executable", "param1", "param2"]` : Provides defaults for executing a container.

`ENTRYPOINT ["executable", "param1", "param2"]` : Configures a container to run as an executable.

`ENV KEY=VALUE` : Sets environment variables.

`EXPOSE [port]` : Indicates the ports on which a container listens for connections.

`VOLUME ["/data"]` : Creates a mount point to externalize data from the container.

`COPY [src] [destination]` : Copies new files or directories into the filesystem of the container.

`WORKDIR` : Sets the working directory inside the container.

`ADD` : Copies new files, directories, or remote file URLs from <src> and adds them to the filesystem of the image at the path <dest>.

4.1.1 Example Java Dockerfile

```
# Start from the Java 17 Temurin image
FROM eclipse-temurin:17 as builder
WORKDIR /app
COPY .mvn .mvn
COPY mvnw mvnw
COPY pom.xml pom.xml
RUN ./mvnw dependency:go-offline
COPY src src
RUN ./mvnw package

FROM eclipse-temurin:17
WORKDIR /app
COPY -from=builder /app/target/app.jar
↪  /app/app.jar
ENTRYPOINT ["java", "-jar", "app.jar"]
CMD ["-help"]
```

4.2 Docker Compose

Docker Compose File (docker-compose.yml) : Defines multi-container Docker applications. The YAML file where you define your applications services, networks, and volumes.

Services : Specifies containers based on Docker images.

Volumes : Configures shared or persistent storage volumes.

Ports : Maps ports between the container and the host.

Networks : Optional section to define the networks.

Environment Variables : Sets environment variables within containers.

4.2.1 Practical tips

`.dockerignore` : Use to exclude files from being copied into the image.

Multi-stage Builds : Optimize Dockerfiles for smaller images by separating build and runtime environments.

Security : Run containers with the least privileges; avoid running as root unless necessary.

4.2.2 Docker compose example

```
networks:
  traefik:
    external: true
services:
  api:
    image: ghcr.io/jonastroeltsch/pw4:latest
    networks:
      - traefik
    expose:
      - 7070
    labels:
      ## Traefik
      - traefik.enable=true
      - traefik.docker.network=traefik
      ## HTTPS
      - traefik.http.routers.
      ↪  whoami.entrypoints=https
      - traefik.http.routers.
      ↪  whoami.rule=Host(`${DOMAIN_NAME}`)
```

5 SMTP & Telnet

5.1 Basic Networking Concepts

IP Addresses : Unique identifiers for devices on a network.

DNS (Domain Name System) : Translates domain names to IP addresses.

5.2 Common Records

`A` : Maps a domain to an IPv4 address.

`AAAA` : Maps a domain to an IPv6 address.

`MX` : Specifies mail exchange servers for a domain.

`TXT` : Holds text information, including SPF records for email validation.

5.3 Email protocols

SMTP (Simple Mail Transfer Protocol) : Used for sending emails. Ports 25 (unencrypted), 465 (SSL), 587 (TLS).

POP3 (Post Office Protocol version 3) : Used for retrieving emails from a server. Ports 110 (unencrypted), 995 (SSL).

IMAP (Internet Message Access Protocol) : Used for retrieving emails from a server, with support for email synchronization. Ports 143 (unencrypted), 993 (SSL).

5.4 SMTP commands

Connect to SMTP Server : telnet [SMTP server address] [port]

Start SMTP Session : EHLO [domain]

Set Sender : MAIL FROM:<sender@example.com>

Set Recipient : RCPT TO:<recipient@example.com>

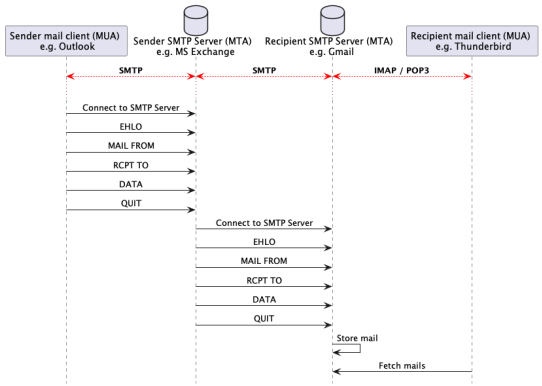
Start Message Body : DATA

Compose Email : Type subject and body. End with a single period on a new line.

Send Email : Press Enter after the single period.

Quit Session : QUIT

5.4.1 Example exchange



5.5 DNS Records for Email

`MX` Record : Defines the mail server responsible for receiving email on behalf of a domain.

`SPF` Record (within `TXT` records) : Lists authorized senders for the domain to prevent email spoofing.

6 TCP in Java

TCP (Transmission Control Protocol) : Connection-oriented, reliable protocol for transmitting data between applications.

UniCast Communication : One-to-one communication between a single sender and receiver.

Connection Establishment : A TCP connection must be established before data can be sent bidirectionally.

6.1 Socket API

`java.net.Socket` : Class used to implement client-side TCP sockets.

`java.net.ServerSocket` : Class used for server-side sockets, listening for incoming connections

6.1.1 Example workflow

Server Workflow

— Create a ServerSocket and bind it to a port.

— Listen for incoming connections with accept().

— Handle the connection with the returned Socket.

— Read/write data through the socket's streams.

— Close the connection.

Client Workflow

— Create a Socket and connect to a server's IP address and port.

— Read/write data through the socket's streams.

— Close the socket.

6.2 Data processing

Use buffered streams (BufferedReader, BufferedWriter) for efficient data reading and writing.

Handle variable length data with delimiters or by sending data length information.

7 UDP in Java

UDP (User Datagram Protocol) : A connectionless, unreliable protocol for sending datagrams without establishing a connection.

Use Cases : Ideal for streaming, gaming, or any application where speed is prioritized over reliability.

7.1 TCP vs UDP

Connection : TCP is connection-oriented ; UDP is connectionless.
Reliability : TCP guarantees delivery ; UDP does not.
Data Flow : TCP is stream-based ; UDP uses individual messages (datagrams).
Usage : TCP for accuracy (web, email) ; UDP for speed (streaming, gaming).

7.2 Working with UDP datagrams

`DatagramSocket` : Used for sending or receiving packets of data.
`DatagramPacket` : Represents data packets sent or received over the network.

7.3 UDP Communication types

Unicast : One-to-one communication.
Broadcast : One-to-all devices on a network.
Multicast : One-to-many, for specified group members.
Reliability and UDP : does not ensure data delivery, order, or duplication protection.

7.4 Messaging Patterns

Fire-and-Forget : Send data without awaiting a response.
Request-Response : Manually implemented pattern for two-way communication.

7.5 Service Discovery Protocols

Facilitate discovering services on a network without prior knowledge of IP addresses.
Utilize UDP for announcing services (advertisement) or querying for services (active discovery).

8 TCP & UDP examples in Java

8.1 TCP multi-threaded example

Server :

```
public static void main(String[] args) {
  try (ServerSocket serverSocket = new
  ↪  ServerSocket(PORT);) {
    while (true) {
      Socket clientSocket =
      ↪  serverSocket.accept();
      Thread clientThread = new Thread(new
      ↪  ClientHandler(clientSocket));
      clientThread.start();
    }
  } catch (IOException e) {
    System.out.println("[Server " + SERVER_ID +
    ↪  "] exception: " + e);
  }
}

static class ClientHandler implements Runnable {
  private final Socket socket;
  public ClientHandler(Socket socket) {
    this.socket = socket;
  }
  @Override
  public void run() {
    try (
      socket;
      BufferedReader in = new BufferedReader(
        new InputStreamReader(
          socket.getInputStream(),
          StandardCharsets.UTF_8
        )
      );
      BufferedWriter out = new BufferedWriter(
        new OutputStreamWriter(
          socket.getOutputStream(),
          StandardCharsets.UTF_8
        )
      )
    ) {
      out.write(TEXTUAL_DATA + "\n");
      out.flush();
    } catch (IOException e) {
      System.out.println("[Server " + SERVER_ID
      ↪  + "] exception: " + e);
    }
  }
}
```

Client

```
private static final String HOST = "localhost";
private static final int PORT = 1234;
private static final int CLIENT_ID = (int)
↪  (Math.random() * 1000000);
private static final String TEXTUAL_DATA =
↪  "from Client " + CLIENT_ID;
public static void main(String args[]) {
  try (
    Socket socket = new Socket(HOST, PORT);
    BufferedReader in = new BufferedReader(
      new InputStreamReader(
        socket.getInputStream(),
        StandardCharsets.UTF_8
      )
    );
    BufferedWriter out = new BufferedWriter(
      new OutputStreamWriter(
        socket.getOutputStream(),
        StandardCharsets.UTF_8
      )
    );
  ) {
    out.write(TEXTUAL_DATA + "\n");
    out.flush();
  } catch (IOException e) {
    System.out.println("[Client " + CLIENT_ID +
    ↪  "] exception: " + e);
  }
}
```

8.2 UDP example

```
// Broadcast Receiver
try (var socket = new DatagramSocket(PORT)) {
  byte[] buffer = new byte[1024];
  var packet = new DatagramPacket(
    buffer, buffer.length);
  socket.receive(packet);
  var message = new String(
    packet.getData(), 0,
    packet.getLength(), UTF_8);
} catch (IOException ex) {
  System.out.println(ex.getMessage());
}

// Broadcast Sender
String IPADDRESS = "255.255.255.255";
int PORT = 44444;
try (var socket = new DatagramSocket()) {
  socket.setBroadcast(true);
  String message = "Hello everybody!";
  byte[] payload = message.getBytes(UTF_8);
  var dest = new InetSocketAddress(IPADDRESS,
  ↪  44444);
  var packet = new DatagramPacket(
    payload, payload.length, dest);
  socket.send(packet);
} catch (IOException ex) {
  System.out.println(ex.getMessage());
}

// Multicast Receiver
String IPADDRESS = "239.1.2.3";
int PORT = 44444;
try (var socket = new MulticastSocket(PORT)) {
  var group =  new InetSocketAddress(
    IPADDRESS, PORT);
  var netif =
  ↪  NetworkInterface.getByName("eth0");
  socket.joinGroup(group, netif);
  byte[] buffer = new byte[1024];
  var packet = new DatagramPacket(
    buffer, buffer.length);
  socket.receive(packet);
  String message = new String(
    packet.getData(), 0,
    packet.getLength(), UTF_8);
  socket.leaveGroup(group, netif);
} catch (IOException ex) {
  System.out.println(ex.getMessage());
}

// Multicast Sender
String IPADDRESS = "239.1.2.3";
int PORT = 44444;
try (var socket = new DatagramSocket()) {
  String message = "Hello group members!";
  byte[] payload = message.getBytes(UTF_8);
  var dest = new InetSocketAddress(
    IPADDRESS, 44444);
  var packet = new DatagramPacket(
    payload, payload.length, dest);
  socket.send(packet);
} catch (IOException ex) {
  System.out.println(ex.getMessage());
}
```