

# IoT Engineering

## 4: Internet Protocols, HTTP and CoAP

CC BY-SA, Thomas Amberg, FHNW  
(unless noted otherwise)

Slides: [tmb.gr/iot-4](http://tmb.gr/iot-4)

# Overview

These slides cover the *Internet protocol suite*.

HTTP and CoAP on the application layer.

TCP and UDP on the transport layer.

And how to write a Web service.

# Prerequisites

Install the Arduino IDE, set up ESP8266, get Wi-Fi:

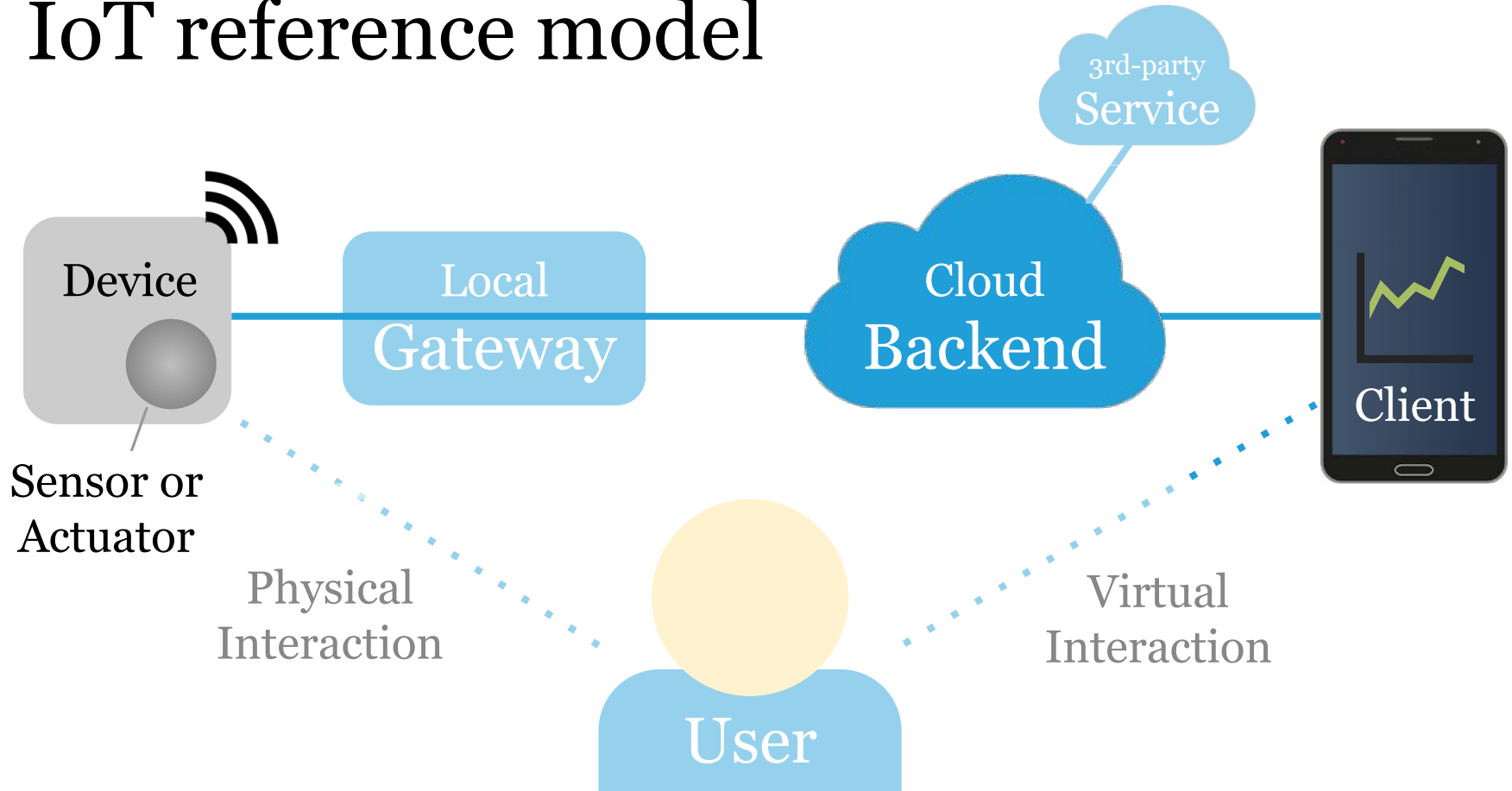
Check the Wiki entry on [Installing the Arduino IDE](#).

[Set up the Feather Huzzah ESP8266](#) for Arduino.

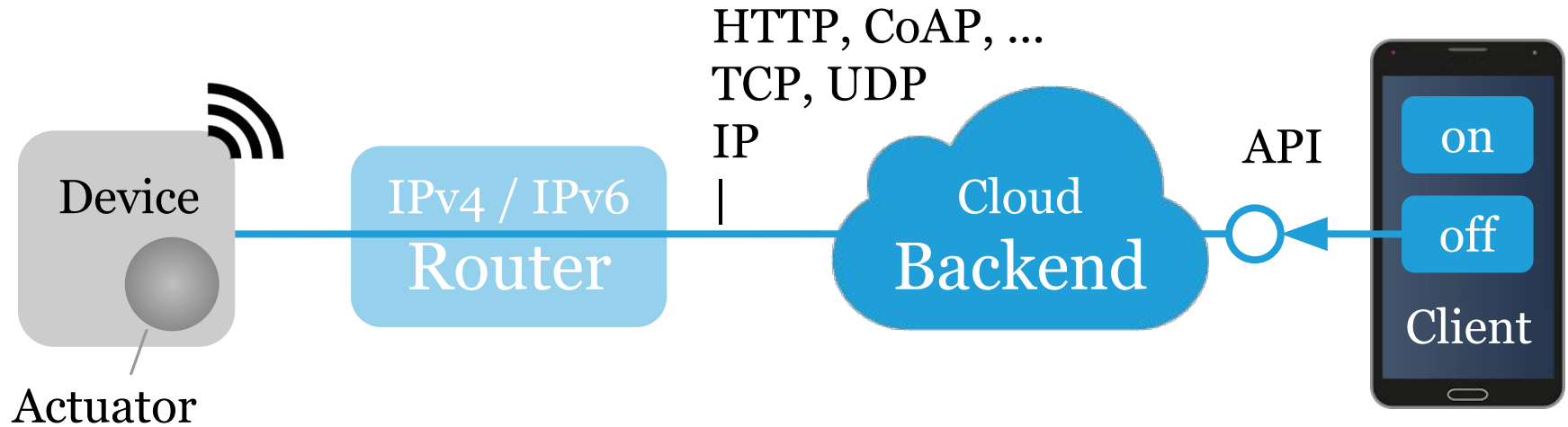
Get access to a Wi-Fi network\* without a portal.

\*In class, try MY\_SSID with MY\_PASSWORD.

# IoT reference model



# Protocols



Internet protocols allow a connected device to send data to a backend or to receive commands sent from a client, as shown here, via a backend Web API.

# Internet protocol suite

**RFC 1122 layers** are loosely based on the **OSI model**:

*Application layer*, process to process, HTTP, CoAP, ...

*Transport layer*, host to (remote) host, UDP or TCP.

*Internet layer*, inter-network addressing and routing.

*Link layer*, details of connecting hosts in a network.

# Internet Protocol (IP)

The **Internet Protocol** is the foundation of the Internet.

It deals with addressing, each host has an **IP address**.

It allows routing datagram packets across networks.

The IP address space is managed by the **IANA**.

# IPv4

Internet Protocol version 4, **IPv4**, **RFC 791**.

Uses 32-bit IP addresses, e.g. 192.168.0.1

The IPv4 loopback address is 127.0.0.1

There are **not enough**\* IPv4 addresses.

\***NAT** is used to conserve addresses.



# IPv6

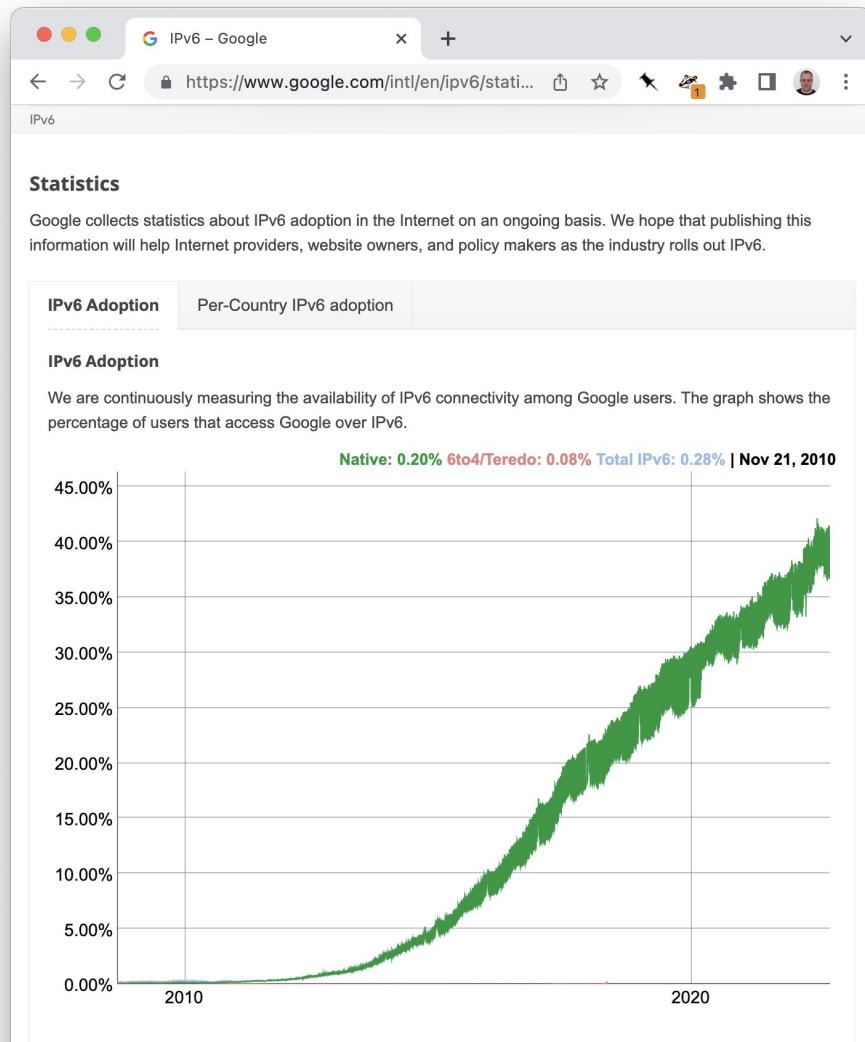
IPv6, RFC 2460.

128-bit IP addresses, e.g.

2001:0db8:85a3:0000:  
0000:8a2e:0370:7334

Loopback address is ::1

Adoption is growing.



# Domain Name System (DNS)

DNS is specified in RFC 1034 (and many other RFCs).

Maps a *domain* name to one or more IP address.

Try, e.g. `$ nslookup www.google.com`

If possible, connect to a domain, IPs can change.

# User Datagram Protocol (UDP)

UDP is specified in RFC 768 and used, e.g. over IP.

UDP is connectionless, it transmits single packets.

UDP is unreliable\*, lost packets are not sent again.

Sent packets can be received in a different order.

UDP allows broadcasting packets (to a subnet).

\*Fine for video, or "fire & forget" messages.

# Transmission Control Protocol (TCP)

TCP is specified in RFC 793 and used, e.g. over IP.

TCP is connection-oriented, host to (remote) host.

TCP is reliable, it provides an ordered byte stream.

Packets are acknowledged, lost ones retransmitted.

# Hypertext Transfer Protocol (HTTP)

[HTTP](#), the "Web protocol", is specified in [RFC 2616](#).

It allows clients & servers to communicate over TCP.

A client sends a request, the server sends a response.

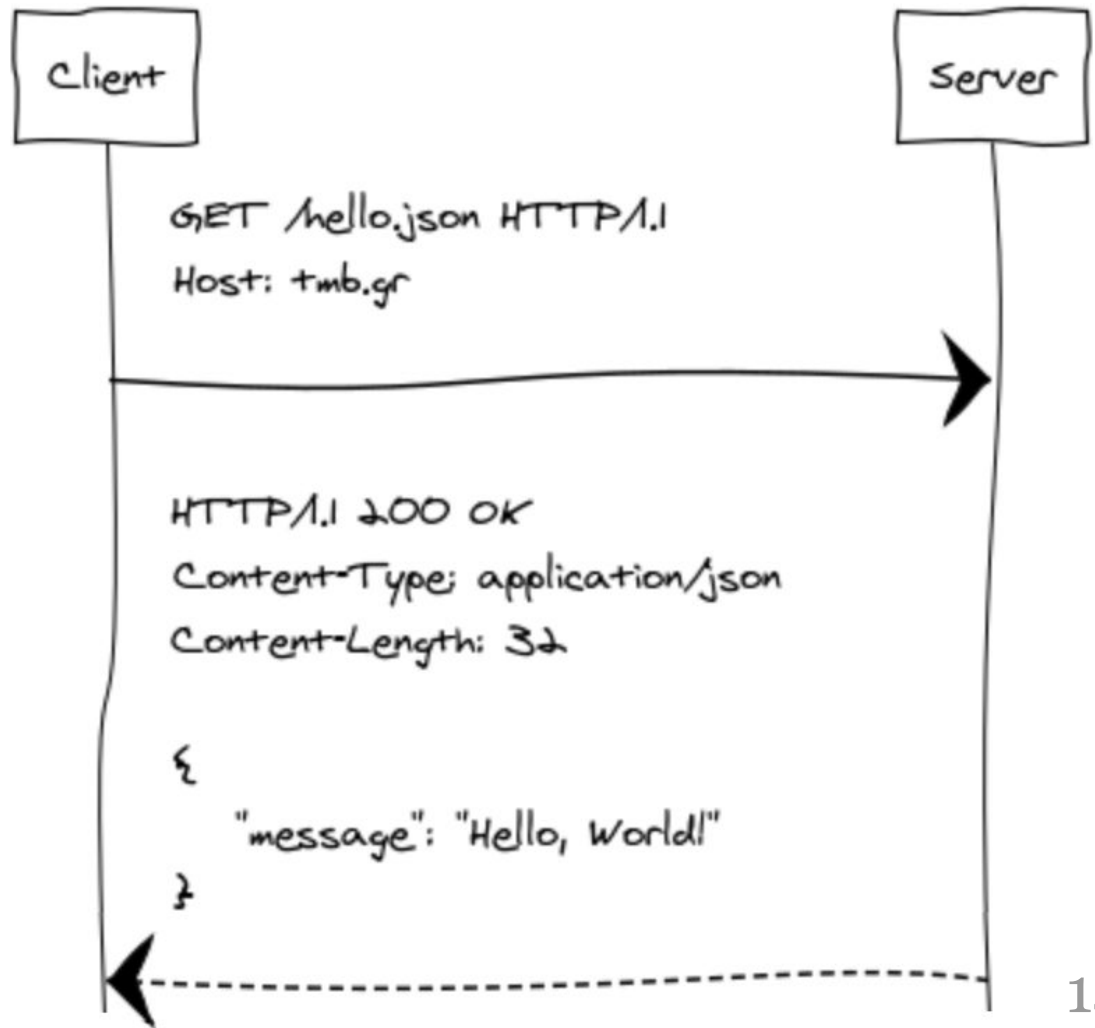
Request and response headers are encoded in [ASCII](#).

The content type and length are declared in headers.

# HTTP

Web request w/  
host header.

Web response  
with headers  
and content.



# HTTP status codes

Each HTTP response contains a **status code**, e.g.

200 OK — the GET, POST, ... request has succeeded.

401 Unauthorized — requires user authentication.

301 Moved Permanently — new permanent URL.

500 Internal Server Error — e.g. an exception.

# Uniform Resource Identifier (URI)

A **URI** is a string of characters to identify a resource.

URI syntax and resolution is specified in **RFC 3986**.

A Unified Resource Locator (**URL**) is a type of URI.

E.g. `https://www.google.com:443/search?q=iot`

has `scheme`, `host`, `port`, `path` and `query` parts.

IoT: URIs refer to physical things/properties.



# Debugging with Curl and PostBin

Curl (<https://curl.haxx.se/>) is a generic Web client.

It's useful to test Web APIs, try this GET request:

```
$ curl -v http://tmb.gr/hello.json
```

Or create a [PostBin](#) and send a POST request with:

```
$ curl --data "t=23" https://postb.in/...
```

Here's the [manual](#) and a book on [Everything Curl](#).

# Web clients

A *Web client* sends Web requests to a Web server.

The basic steps to send a HTTP Web request are:

- Create a client (or a client **socket**, in Unix/Linux)
- Connect to a remote host (or IP) and port, e.g. 80
- Write the client request, read the server response
- Close the connection, or send another request

Web browsers/clients like Curl work the same way. 18

# Web services

A *Web server*, or *service*, responds to client requests.

The basic steps to handle HTTP Web requests are:

- Create a server at a specific port
- Begin listening at the local IP address
- Accept connections from clients if available
- Read the client request, write a response
- Close the connection to the client

# Push vs. Pull (or Poll)

Does it make sense for a device to be a server/service?

Yes, to pull data  
from the device.

Or to push com-  
mands to it.

**Device** **===Data==>** **Backend**

Client ---Push--> Server

Server <--Pull--- Client

**Device** **<===Cmds===** **Backend**

Client ---Pull--> Server

Server <--Push--- Client

See [GSIoT](#), p.35

# ESP8266 Web service

.ino

```
#include <ESP8266WiFi.h>
```

```
WiFi.begin(ssid, password);
```

```
WiFiServer server(port);
```

```
server.begin();
```

```
WiFiClient client = server.available();
```

```
if (client && client.connected()) {
```

```
    Serial.println(client.remoteIP());
```

```
...
```

# Reading a Web request

Reading a Web request character by character:

```
int ch = client.read(); // -1 or next char
```

The *Content-Length* header contains the length in number of bytes available to read after the headers, it's usually = 0 for GET and  $\geq 0$  for POST requests.

The *Content-Type* defines the format and encoding of the content, which starts right after the headers.

# Sending a Web response

An HTTP response for "success", on ESP8266:

```
client.print("HTTP/1.1 200 OK\r\n");  
client.print("Content-Length: 0\r\n");  
client.print("Connection: close\r\n");  
client.print("\r\n");
```

The header *Connection: close* tells the client to close the connection after reading the response.

# ESP8266 LED Web service

.ino

For prototyping, "parsing" like this is good enough:

```
// PUT /led/state/1
// PUT /led?state=1
if (client.find("state")) {
    int state = client.parseInt();
    Serial.println(state);
    client.find("\\r\\n\\r\\n"); // skip headers
```

...



# Basic authentication

HTTP **Basic Authentication** is specified in **RFC 7617**.

It uses the *Authorization: Basic credentials* header.

Credentials are **Base64** encoded *user ':' password*.

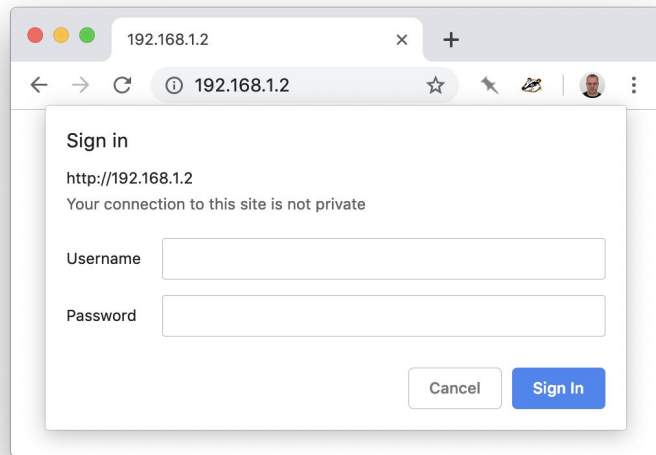
E.g. for user = *tamberg*, password = *oops* this is:

**Authorization: Basic** dGFtYmVyZzpvb3Bz

# Basic authentication

```
> GET / HTTP/1.1\r\n> Host: 192.168.1.2\r\n> \r\n
```

```
< HTTP/1.1 401 Unauthorized\r\n< WWW-Authenticate: Basic\r\n< Content-Type: text/html\r\n< Content-Length: 0\r\n< \r\n
```



# ESP8266 basic auth Web service .ino

```
if (client.find("Authorization: Basic ")) {  
    char creds[64]; // buffer for credentials  
    readStringToEndOfLine(client, creds, 64);  
    client.find("\r\n\r\n"); // skip headers  
    if (strcmp(storedCreds, creds) == 0) {  
        send200Response(client); // auth'ed  
    } else { send401Response(client); }  
} else { send401Response(client); }
```

```
$ curl --user name:password http://192... 27
```

# ESP8266 secure Web service

.ino

```
BearSSL::WiFiServerSecure server(443);  
static const char cert[] PROGMEM = R"EOF( ...  
static const char key[] PROGMEM = R"EOF( ...  
  
server.setRSACert(  
    new BearSSL::X509List(cert),  
    new BearSSL::PrivateKey(key));  
server.begin();  
WiFiClientSecure cInt = server.available();  
  
$ curl --insecure -v http://192...
```

# Hands-on, 15': Web services

Build and run the previous Web service examples.

Use the *.ino* link on each page to find the source.

Check the serial monitor to see the server IP.

Use your browser and Curl as Web clients.

Done? Here's a bonus [example](#) to study.

# API

An **API**, or application programming interface, is a contract between clients and providers of a service.

Both parties have to agree on:

- How to access the service.
- How to submit data to it.
- How to get data out of it.

Good APIs are documented or self-explanatory.

# RESTful API

Representational state transfer is an API design style.

Uniform methods, e.g. HTTP GET, PUT, POST and DELETE, are used to perform actions on resources.

A resource is anything that can be named/identified: documents, people, or in general data and functions.

E.g. PUT /kitchen/light?state=on

Here's a [REST API tutorial](#).

# Philips Hue API

Philips Hue is a smart home lighting solution, API:

<https://developers.meethue.com/develop/hue-api/>

Reading the API documentation requires an account.

Register for free at <https://developers.meethue.com/>

We'll look at some Hue bulbs and the Hue bridge.



# Hands-on, 15': Philips Hue API

Read the API documentation to find the following:

API endpoints, protocols, data formats, queries.

Try to control the Philips Hue lights in class.

Take notes on links, tools, requests used.

# Constrained Application Protocol (CoAP)

CoAP is specified in RFC 7252, terminology in 7228.

It allows clients & servers to communicate over UDP.

A client sends a request, the server sends a response.

Request and response messages are binary encoded.

DTLS, TLS for UDP, provides security for CoAP.

# CoAP vs. HTTP

CoAP is similar to, but uses less bytes than, HTTP.

GET, PUT, POST, ..., error codes, etc. are encoded.

Resources are discoverable at .well-known/core

*Observe* allows subscribing to resource updates.

Here's a [CoAP cheat sheet](#) by [@markushx](#).

# Hands-on, 15': CoAP

Download a CoAP client and server [implementation](#).

Run it on your laptop or on the ESP8266, if possible\*.

Consider using a testing service like <http://coap.me/>

\*Some libraries include example code.

# IKEA Tradfri

[IKEA Tradfri](#) is another smart home lighting solution.

There is no official API, but some [3rd party libraries](#).

[3rd party docs](#) show how the gateway uses CoAP.

We'll look at Tradfri bulbs, and at the gateway.

# Hands-on, 15': IKEA Tradfri

Search for hints about the API to find the following:

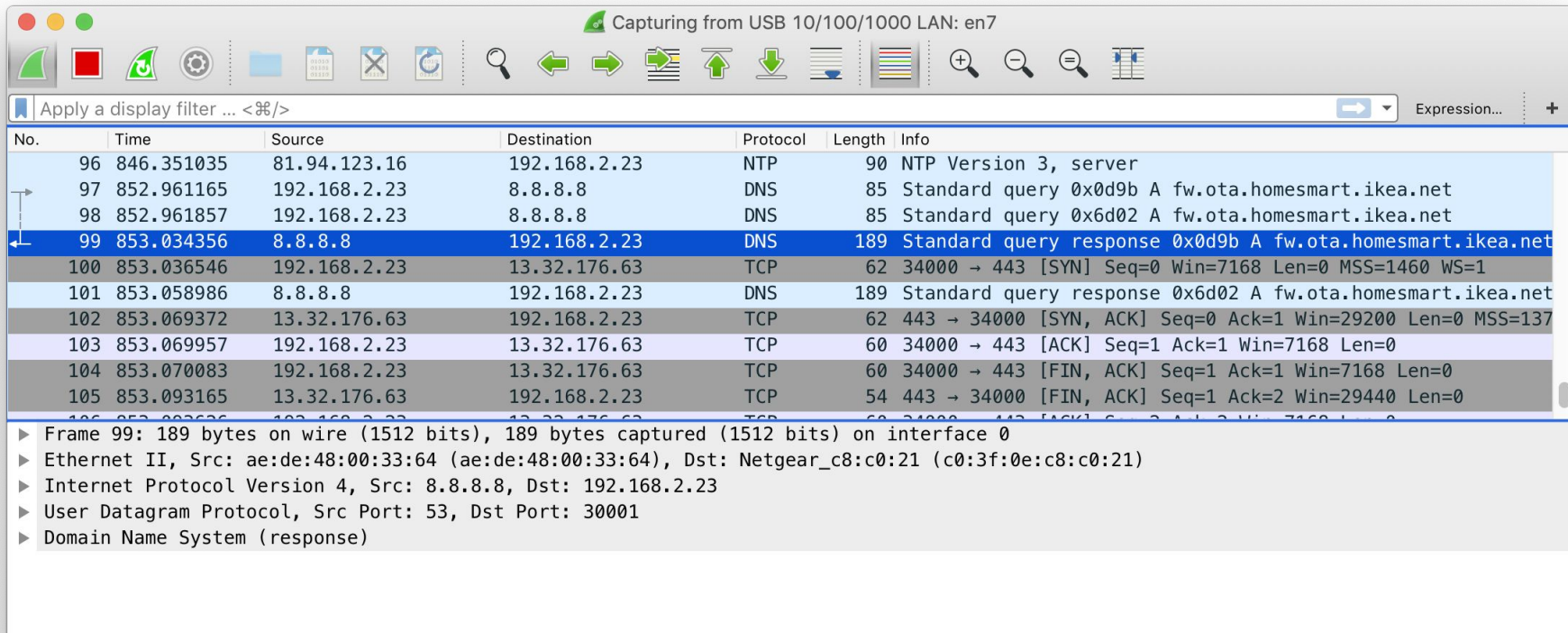
API endpoints, protocols, data formats, queries.

Try to access the IKEA Tradfri gateway in class.

Take notes about links, tools, requests used.

# Inspecting network traffic

Tools like **Wireshark** help analyze network protocols:



Capturing from USB 10/100/1000 LAN: en7

Apply a display filter ... <⌘/>

No.	Time	Source	Destination	Protocol	Length	Info
96	846.351035	81.94.123.16	192.168.2.23	NTP	90	NTP Version 3, server
97	852.961165	192.168.2.23	8.8.8.8	DNS	85	Standard query 0x0d9b A fw.ota.homesmart.ikea.net
98	852.961857	192.168.2.23	8.8.8.8	DNS	85	Standard query 0x6d02 A fw.ota.homesmart.ikea.net
99	853.034356	8.8.8.8	192.168.2.23	DNS	189	Standard query response 0x0d9b A fw.ota.homesmart.ikea.net
100	853.036546	192.168.2.23	13.32.176.63	TCP	62	34000 → 443 [SYN] Seq=0 Win=7168 Len=0 MSS=1460 WS=1
101	853.058986	8.8.8.8	192.168.2.23	DNS	189	Standard query response 0x6d02 A fw.ota.homesmart.ikea.net
102	853.069372	13.32.176.63	192.168.2.23	TCP	62	443 → 34000 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=137
103	853.069957	192.168.2.23	13.32.176.63	TCP	60	34000 → 443 [ACK] Seq=1 Ack=1 Win=7168 Len=0
104	853.070083	192.168.2.23	13.32.176.63	TCP	60	34000 → 443 [FIN, ACK] Seq=1 Ack=1 Win=7168 Len=0
105	853.093165	13.32.176.63	192.168.2.23	TCP	54	443 → 34000 [FIN, ACK] Seq=1 Ack=2 Win=29440 Len=0

▶ Frame 99: 189 bytes on wire (1512 bits), 189 bytes captured (1512 bits) on interface 0

- ▶ Ethernet II, Src: ae:de:48:00:33:64 (ae:de:48:00:33:64), Dst: Netgear\_c8:c0:21 (c0:3f:0e:c8:c0:21)
- ▶ Internet Protocol Version 4, Src: 8.8.8.8, Dst: 192.168.2.23
- ▶ User Datagram Protocol, Src Port: 53, Dst Port: 30001
- ▶ Domain Name System (response)

# Summary

We looked at the layers of the Internet protocol suite.

At the Internet layer, we saw the Internet protocol.

At the transport layer, we looked at UDP and TCP.

At the application layer, we met HTTP and CoAP.

Next: Local Connectivity with Bluetooth LE.



# Challenge

Write a connected display service on the ESP8266.

Create a RESTful Web API for the **4-digit display**.

Document the HTTP calls your API can handle.

Print the service IP address to the console.

Commit to the hands-on repo.

# Feedback or questions?

Write me on Teams or email

[thomas.amberg@fhnw.ch](mailto:thomas.amberg@fhnw.ch)

Thanks for your time.