

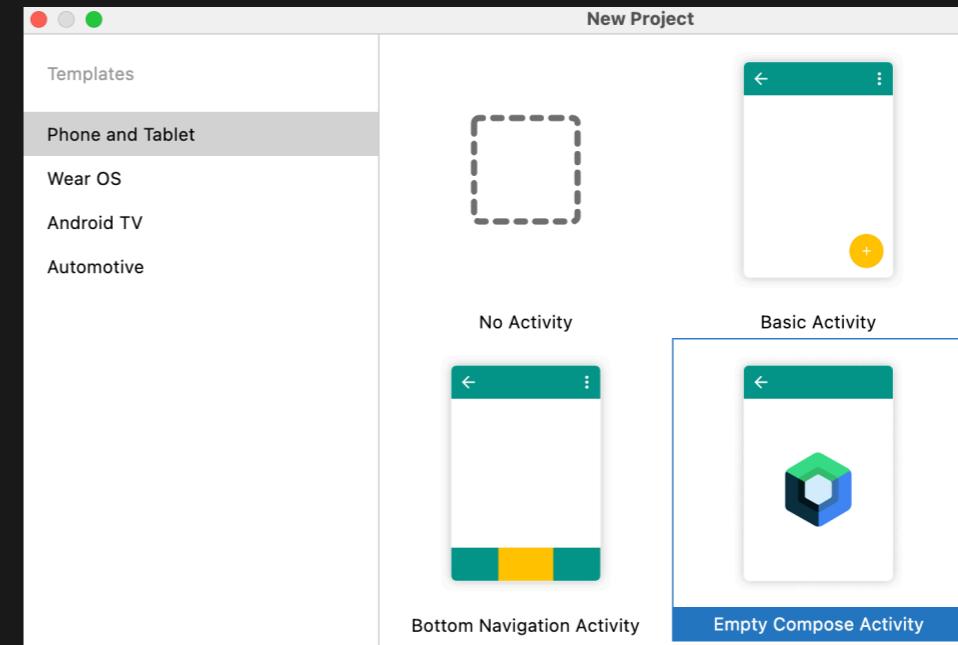
COMPOSE

ANDROID DEVELOPMENT STUDIO

- Android Apps can be developed in different IDEs, most prominent is Android Studio.
- You can download the latest version of Android Studio [here](#).
- Android Studio provides a wizard for generating a simple "Hello World" project.
- During the wizard, you need to provide a "Minimum SDK". The Minimum SDK defines the lowest API level your app will run on (i.e. API 21: Android 5.0 (Lollipop)).

EXERCISE

Open Android Studio and create a new project called TestProject. In the wizard, select the your installed API as minimum SDK and "Empty Activity". Run the project. For installing the emulator, goto Tools -> Device Manager -> Create Device.



VERSIONING

Android versions are managed using so called Api levels.
New os releases often provide new API levels.

Version	Released	API Level	Name
14	October 2023	34	Android 14
13	August 2022	33	Android 13
12	March 2022	32	Android 12L
12	October 2021	31	Android 12
11	September 2020	30	Android 11

↓ more ↓

Version	Released	API Level	Name
10	September 2019	29	Q
9.0	August 2018	28	Pie
8.1	December 2017	27	Oreo
8.0	August 2017	26	Oreo
7.1	October 2016	25	Nougat
7.0	August 2016	24	Nougat

↓ more ↓

Version	Released	API Level	Name
6.0	October 2015	23	Marshmallow
5.1	March 2015	22	Lollipop
5.0	Nov 2014	21	Lollipop
4.4W	June 2014	20	Kitkat Watch
4.4	Oct 2013	19	Kitkat
↓ more ↓			

Version	Released	API Level	Name
4.3	July 2013	18	Jelly Bean
4.2-4.2.2	Nov 2012	17	Jelly Bean
4.1-4.1.1	June 2012	16	Jelly Bean
4.0.3-4.0.4	Dec 2011	15	Ice Cream Sandwich
4.0-4.0.2	Oct 2011	14	Ice Cream Sandwich

Version	Released	API Level	Name
3.2	June 2011	13	Honeycomb
3.1.x	May 2011	12	Honeycomb
3.0.x	Feb 2011	11	Honeycomb

↓ more ↓

Version	Released	API Level	Name
2.3.3-2.3.4	Feb 2011	10	Gingerbread
2.3-2.3.2	Nov 2010	9	Gingerbread
2.2.x	June 2010	8	Froyo
2.1.x	Jan 2010	7	Eclair
2.0.1	Dec 2009	6	Eclair
2.0	Nov 2009	5	Eclair

↓ more ↓

Version	Released	API Level	Name
1.6	Sep 2009	4	Donut
1.5	May 2009	3	Cupcake
1.1	Feb 2009	2	Base
1.0	Oct 2008	1	Base

PLATFORM VERSIONS I

Android defines three different versions:

- `minSdkVersion`: The minimum api level your app will run on, contains the value you defined during the project wizard.
- `compileSdkVersion`: The api level that was used to build your app. Check Tools->Android->SDK Manager to see your installed SDKs.
- `targetSdkVersion`: The api level that was explicitly tested and works with this app. This is used by Android to optimize compatibility code. This is checked at run time.

PLATFORM VERSIONS II

- Calling an Api larger than your minSdkVersion will raise a compile error.
- Android provides a number of support libraries that allow to use new features and still support older API levels (the support libraries have been moved to AndroidX, the concept remains the same).
- Check <http://developer.android.com/tools/support-library/index.html> for an overview.
- Check <https://developer.android.com/jetpack/androidx/migrate/class-mappings> for a list of mappings from support library to AndroidX.

PLATFORM VERSIONS III

The platform versions supported by an app are defined in build.gradle:

```
android {  
    compileSdkVersion 22  
    buildToolsVersion "22.0.1"  
  
    defaultConfig {  
        applicationId "ch.zhaw.testapplication"  
        minSdkVersion 21  
        targetSdkVersion 22  
        versionCode 1  
        versionName "1.0"  
    }  
}
```

ANDROID FILES

- `AndroidManifest.xml`: Defines the name of the app as well as the main entry point, the icons, style and many other things.
- `res/`: Subfolder that contains all resources (layouts, images, strings, ...).
- `build.gradle`: Build script for the Android builder tool gradle.
- `gradle.properties`, `settings.gradle`: Some files to store project settings.

ANDROID PACKAGE

An Android app is always packaged in an apk file (similar to iOS ipa). It is also a zip with a defined structure.

COMPOSE

OVERVIEW

Jetpack Compose:

- is a declarative framework to generate the user interface of an Android app
- is developed by Google
- has first been presented at GoogleIO 2019 and launched in 2021.

FIRST EXAMPLE I

```
@Composable
fun FirstExample(modifier: Modifier = Modifier) {
    var name = remember { mutableStateOf("Paul Newman") }
    Row(verticalAlignment = Alignment.CenterVertically,
        modifier = Modifier.padding(all = 5.dp)) {

        //string output in light gray text color
        Text(name.value, modifier =
            Modifier.background(Color.LightGray))
```



FIRST EXAMPLE II

```
// and a button component (onClick changes  
// the name, this will automatically  
// redraw the screen)  
Button(onClick = { name.value = "John Ford" })  
{ Text("Change Name") }  
}  
}
```



OBSERVATIONS I

- Compose uses components. A component is almost always a function with component annotation:

```
@Composable
inline fun Row(
    modifier: Modifier = Modifier,
    horizontalArrangement: Arrangement.Horizontal = Arrangement.Start,
    verticalAlignment: Alignment.Vertical = Alignment.Top,
    content: @Composable RowScope.() -> Unit
)
```

OBSERVATIONS II

- modifier, horizontalArrangement and verticalAlignment have default values. Thus, they can be omitted.
- content is a function type. If a function type is the last parameter, it can be written after the closing bracket:

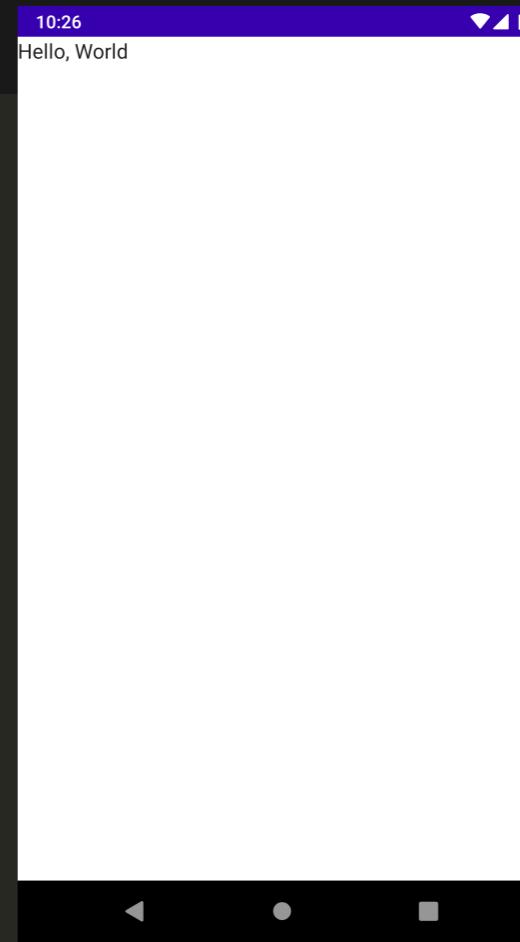
```
//modifier, horizontalArrangement and verticalAlignment  
//are omitted  
Row({Text() Button()})  
//this can be changed to  
Row(){Text(...) Button(...)}  
//and, if no parameter is provided, the round  
//brackets can also be removed  
Row{Text(...) Button(...)}
```

OBSERVATIONS III

- Kotlin can parse all components in a closure (Text(...)
Button(...)) using a special language feature
- Component configurations are done via modifiers or attributes

TEXT COMPONENT

```
setContent {  
    ComposeExampleTheme {  
        // A surface container  
        //using the 'background' color  
        //from the theme  
        Surface(color = MaterialTheme.colors.background) {  
            //The text to present  
            Greeting("Android")  
        }  
    }  
}  
  
@Composable  
fun Greeting(name: String) {  
    Row {  
        Text("Hello, World")  
    }  
}
```



MODIFIERS I

Use Modifiers to change the appearance of a component.

Modifiers can be chained. There are numerous modifiers for each component. These include

- width and height
- foreground color (color)
- background color (background)
- border
- padding
- margin

MODIFIERS II

```
...
Text("Hello, World", modifier =
    Modifier.background(Color.Green)
        .padding(8.dp)
        .border(width=1.dp, color=Color.Blue)
)
...
...
```



ATTRIBUTES I

All components provide a number of additional arguments that can be changed from default values. A Text component has:

- color (ie. Color.White)
- fontSize (ie. 30.sp)
- fontStyle (ie. FontStyle.Italic)
- fontWeight (ie. FontWeight.Bold)
- textAlign (ie. TextAlign.Center)
- fontFamily (ie. FontFamily.SansSerif)

ATTRIBUTES II

```
Text("Hello, World", modifier =  
    Modifier.background(Color.Green)  
    .padding(8.dp)  
    .border(width=1.dp, color=Color.Blue),  
    fontSize = 16.sp,  
    fontWeight = FontWeight.SemiBold  
)
```



DIMENSIONS

px: Pixels on the screen

dp: Density-independent pixels, relative to a 160 dpi (dots per inch) screen

sp: Scale-independent pixels, like dp but also scaled by the user's font size preference

EXERCISE TEXTCHANGEAPP: TEXTVIEW

Create a new Project "TextChangeApp".

Implement this layout by extending the following code (the Greeting function should already be created):

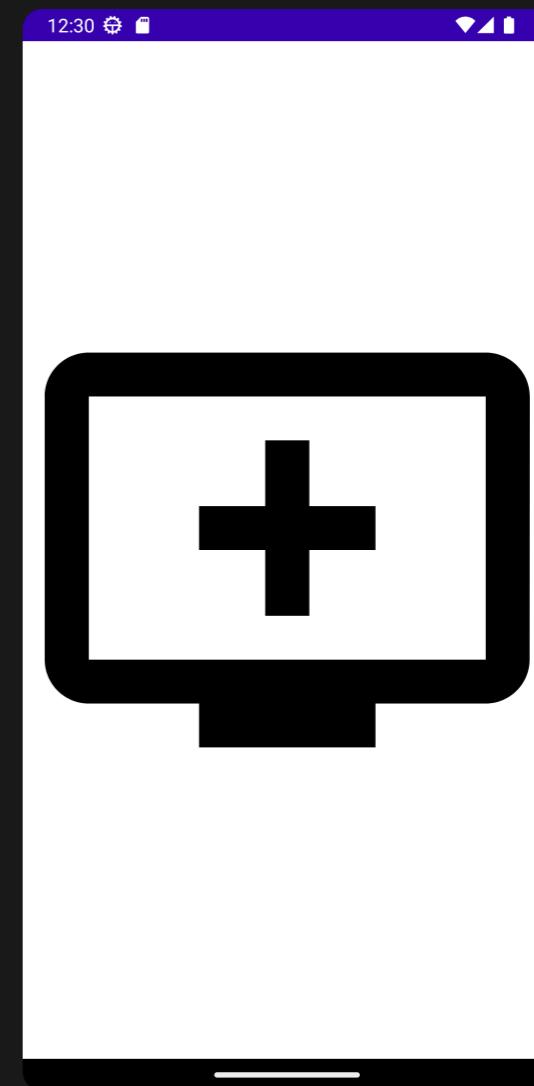
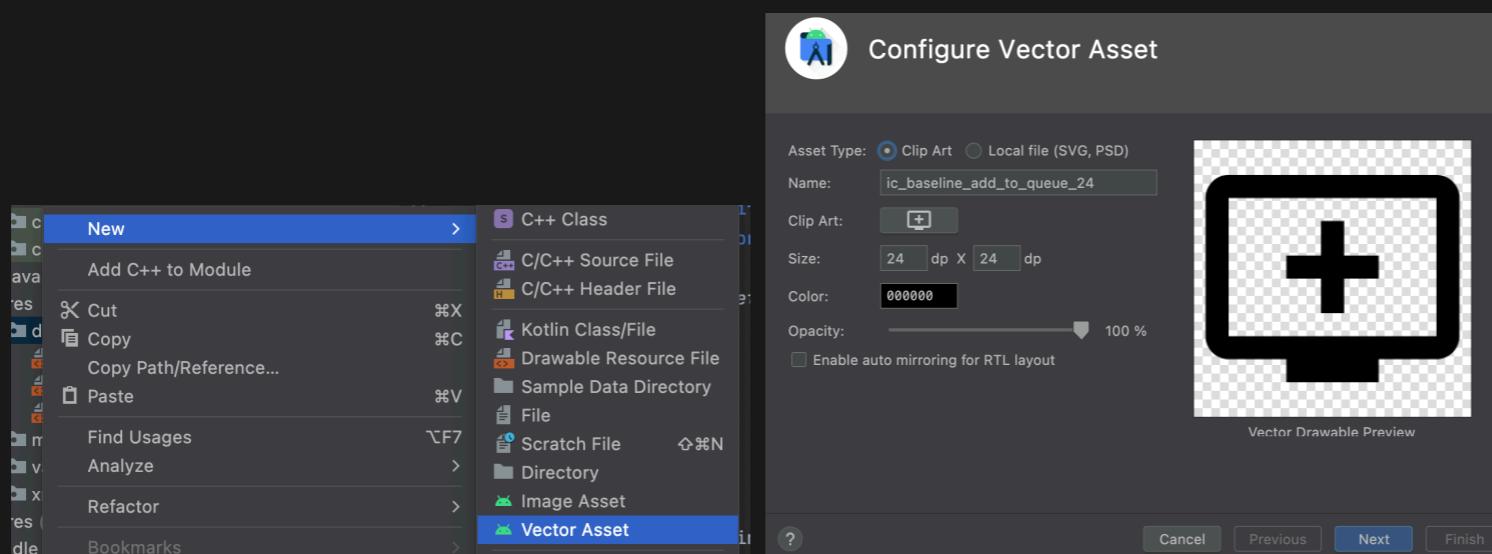


```
@Composable  
fun Greeting(name: String, modifier: Modifier = Modifier) {  
    Column {  
        Row {  
            Text("Hallo Welt", modifier = modifier.fillMaxWidth())  
        }  
    }  
}
```

- Check the official [TextView documentation](#) and use `fillMaxWidth`, `textAlign`, `Color.darkGray` (background) as well as `fontSize 24.sp`

IMAGE COMPONENT I

If you want to load a local vector image, add it first:

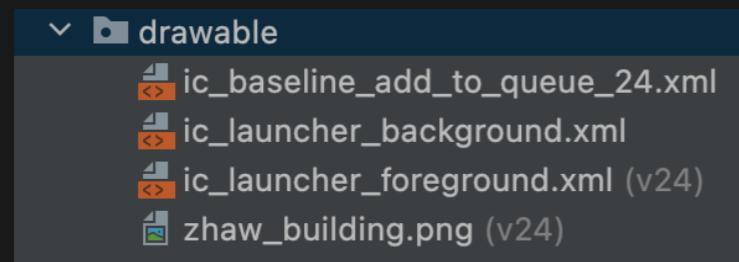


and use it (further examples [here](#)):

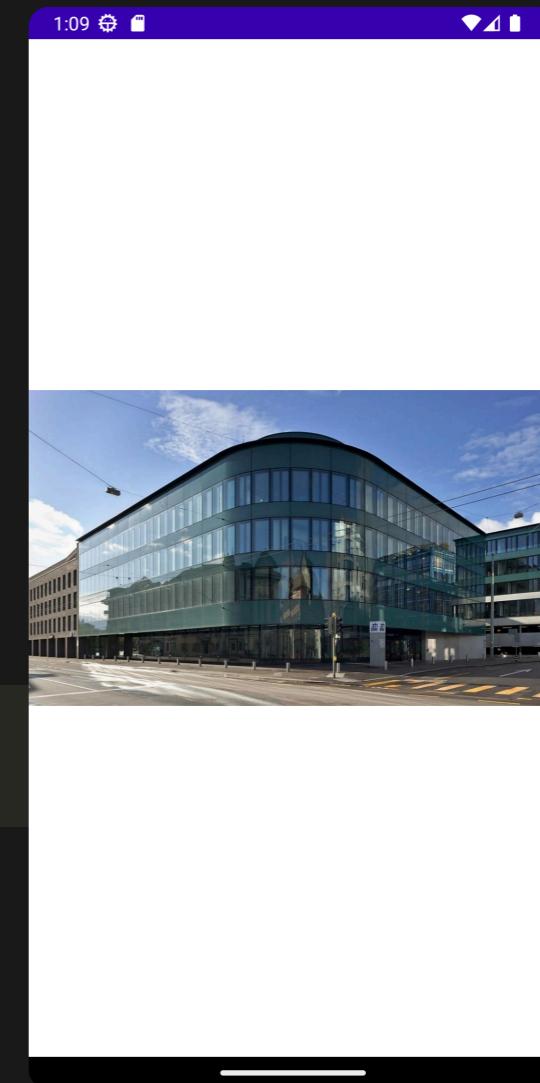
```
Image(painterResource(id =  
    R.drawable.ic_baseline_add_to_queue_24),  
    contentDescription = "vector add to queue")
```

IMAGE COMPONENT II

The easiest way for pixel images is to use copy/paste in the drawable folder



```
Image(painterResource(id = R.drawable.zhaw_building),  
      contentDescription = "zhaw building")
```



BUTTON COMPONENT I

```
//a clickable Button
Button(onClick = {
    //this closure will be called
    //when the Button is clicked
    //the following line logs the string
    //"button clicked" to the console
    Log.i("Button", "button clicked")
})
{
    Text("OK") //The label of the button
}
//by default, the button will take the whole
//available space. When used within a Row or Column
//it will shrink to the expected size
```



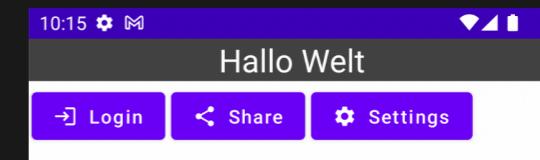
BUTTON COMPONENT II

```
Row {  
    Button(  
        onClick = { /* ... */ },  
        modifier=Modifier.padding(2.dp)  
    ) { Text("OK") }  
    Spacer(Modifier.size(2.dp)) // adds space  
    Button(onClick = { /* ... */ },  
        modifier=Modifier.padding(2.dp)  
    ) {  
        Icon(  
            painterResource(R.drawable.ic_baseline_login_24),  
            contentDescription = "Login",  
            modifier = Modifier.size(ButtonDefaults.IconSize) )  
        Spacer(Modifier.size(ButtonDefaults.IconSpacing))  
        Text("Login") }  
}
```



EXERCISE TEXTCHANGEAPP: THREE BUTTONS

Add three Buttons to your TextChangeApp:



```
@Composable  
fun Greeting(name: String) {  
    Column {  
        Text...  
        Row {  
            Button ...  
            Button ...  
            Button ...  
        }  
    }  
}
```

LAYOUT EXAMPLE

```
Row {  
    Text("Text1", fontSize= 48.sp,  
        color=Color.White,  
        modifier = Modifier.background(Color(0xFF41729f)))  
    Text("Text2", fontSize= 36.sp,  
        color=Color.White,  
        modifier = Modifier.background(Color(0xFF5885af)))  
    Text("Text3", fontSize= 48.sp,  
        color=Color.White,  
        modifier = Modifier.background(Color(0xFF274472)))  
}
```



MAXWIDTH AND MAXHEIGHT

Whenever a composable is within a Column or Row, it will only take minimal space:

```
Column {  
    Text("Text1", modifier = Modifier.background(  
        Color.Blue))  
}
```



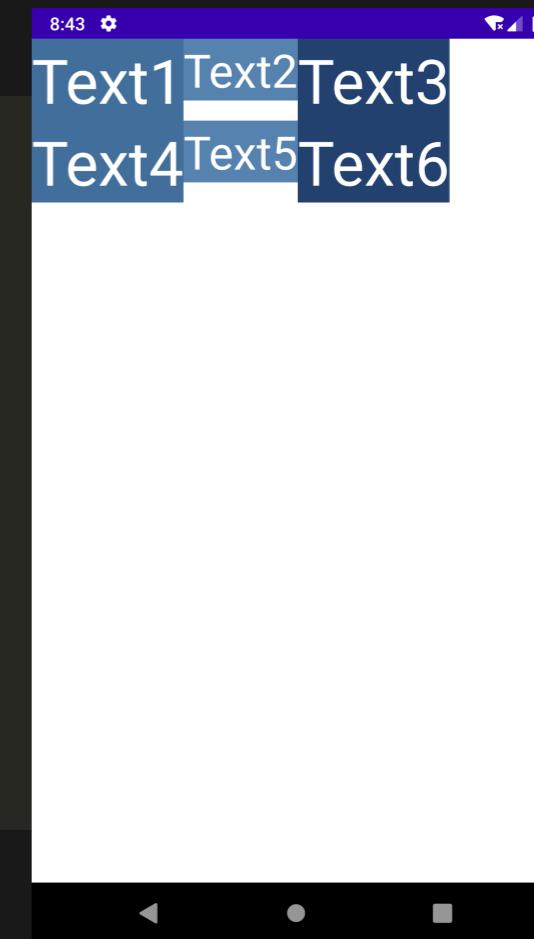
Use .fillMaxWidth() or .fillMaxHeight() to enlarge the composable:

```
Column {  
    Text("Text1", modifier = Modifier.background  
        (Color.Blue)  
        .fillMaxWidth())  
}
```

LAYOUT: ROWS AND COLUMNS

You can nest rows and columns.

```
Column {  
    Row {  
        Text("Text1", ...)  
        Text("Text2", ...)  
        Text("Text3", ...)  
    }  
    Row {  
        Text("Text4", ...)  
        Text("Text5", ...)  
        Text("Text6", ...)  
    }  
}
```



LAYOUT: SPACER I

Spacer is a method:

```
fun Spacer(modifier: Modifier?): Unit
```

that can be used with Modifier.width, Modifier.height, Modifier.size and Modifier.weight modifiers. Use Spacer to separate components.

LAYOUT: SPACER II

```
Row {  
    Text("Text1", fontSize= 48.sp,  
        color=Color.White,  
        modifier = Modifier.background(Color(0xFF41729f)))  
    Spacer(modifier = Modifier.width(16.dp))  
    Text("Text2", fontSize= 36.sp,  
        color=Color.White,  
        modifier = Modifier.background(Color(0xFF5885af)))  
    Text("Text3", fontSize= 48.sp,  
        color=Color.White,  
        modifier = Modifier.background(Color(0xFF274472)))  
}
```



LAYOUT: SPACER II

`weight = 1.0f` will "eat" the available space.

```
Row {  
    //Note that weight will work in the direction of the  
    //Row / Column  
    Text("Text1", fontSize= 48.sp,  
        color=Color.White,  
        modifier = Modifier.background(Color(0xFF41729f)))  
    Spacer(modifier = Modifier.weight(1.0f))  
    Text("Text2", fontSize= 36.sp,  
        color=Color.White,  
        modifier = Modifier.background(Color(0xFF5885af)))  
    Text("Text3", fontSize= 48.sp,  
        color=Color.White,  
        modifier = Modifier.background(Color(0xFF274472)))  
}
```



EXERCISE WEATHERAPP: OUTPUT

Create a new Project "WeatherApp".

Implement this layout. Some hints:

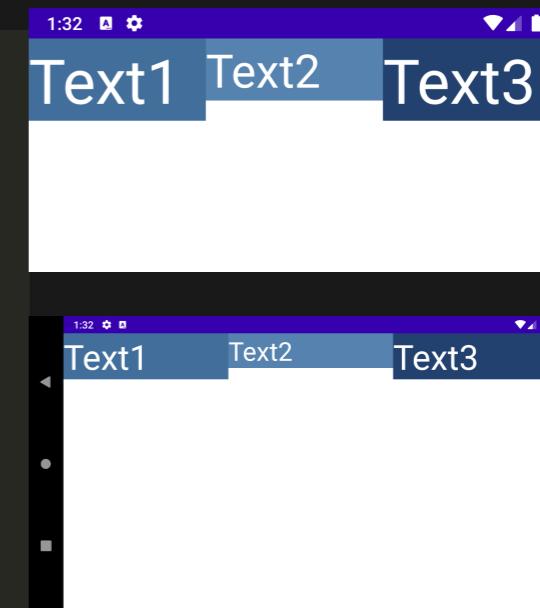


- temperature font size is 96.sp, the other two are 16.sp
- location is italic font style
- all three should be centered horizontally:

```
Column(horizontalAlignment = Alignment.CenterHorizontally)
```

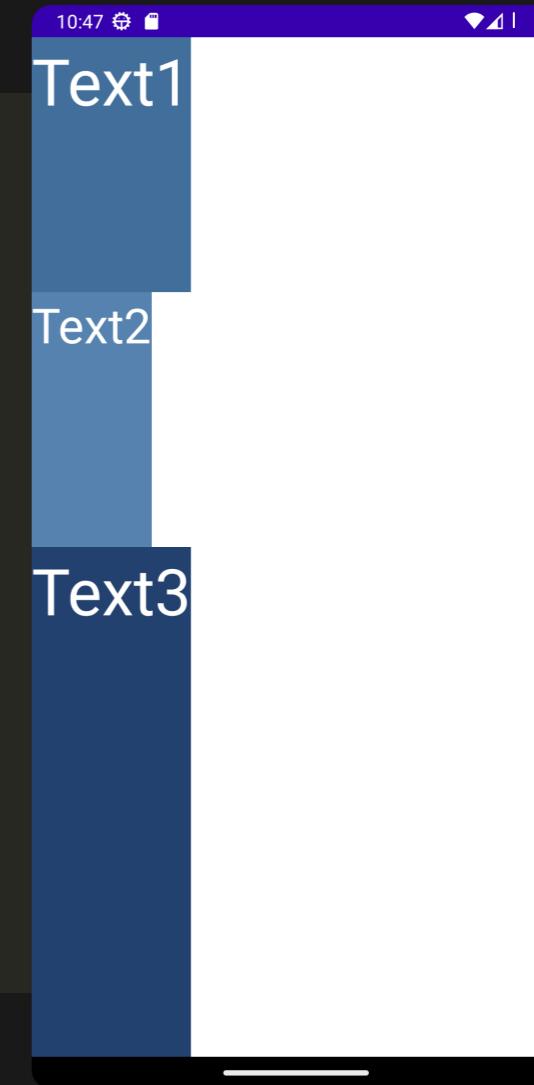
LAYOUT: MODIFIER WEIGHT ON COMPONENTS I

```
//Use weight=1.0 on multiple components for even spacing
Row {
    Text("Text1", fontSize= 48.sp,
        color=Color.White,
        modifier = Modifier
            .background(Color(0xFF41729f)))
            .weight(1f))
    Text("Text2", fontSize= 36.sp, color=Color.White,
        modifier = Modifier
            .background(Color(0xFF5885af)))
            .weight(1f))
    Text("Text3", fontSize= 48.sp, color=Color.White,
        modifier = Modifier
            .background(Color(0xFF274472)))
            .weight(1f))
```



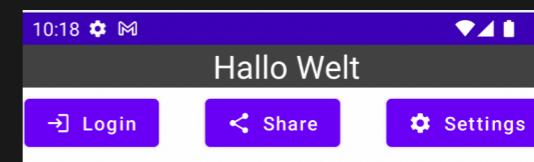
LAYOUT: MODIFIER WEIGHT ON COMPONENTS II

```
//by changing the previous example from Row to Column,  
//it will work as expected. Additionally, weight = 2  
//will double the height of the component  
Column {  
    Text("Text1", fontSize= 48.sp,  
        color= Color.White, modifier = Modifier  
            .background(Color(0xFF41729f)))  
            .weight(1f))  
    Text("Text2", fontSize= 36.sp, color=Color.White,  
        modifier = Modifier  
            .background(Color(0xFF5885af)))  
            .weight(1f))  
    Text("Text3", fontSize= 48.sp, color=Color.White,  
        modifier = Modifier  
            .background(Color(0xFF274472)))
```



EXERCISE TEXTCHANGEAPP: BUTTON LAYOUT

Use Spacer() to separate the Buttons in your
TextChangeApp



LAYOUT INTRINSIC SIZE I

Intrinsics lets you query children before they're actually measured. This is useful in a lot of cases:

Goal: Set all three texts to the same height.

Idea: Modifier.fillMaxHeight() for "Text 2"

LAYOUT INTRINSIC SIZE II

```
Row {  
    Text(  
        "Text1",...)  
    Text(  
        "Text2",...,  
        modifier = Modifier  
            .background(Color.parse("#5885af"))  
            .weight(1f)  
            .fillMaxHeight()  
    )  
    Text(  
        "Text3",...)  
}
```



LAYOUT INTRINSIC SIZE III

Problem: The layout of each row will use the maximum available space. Setting the height of the row to IntrinsicSize.Min will only use the minimum size that is needed (by the highest component of the row). Then everything works as expected.

LAYOUT INTRINSIC SIZE IV

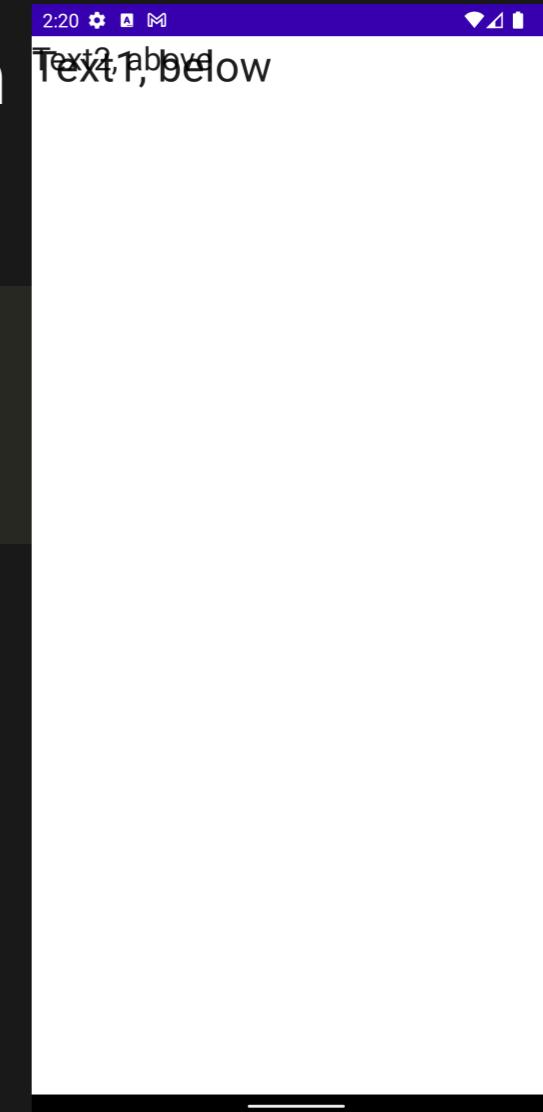
```
Row(modifier = Modifier.height(IntrinsicSize.Min)) {  
    Text(  
        "Text1",...)  
    Text(  
        "Text2",...,  
        modifier = Modifier  
            .background(Color(0xFF5885af))  
            .weight(1f)  
            .fillMaxHeight()  
    )  
    Text(  
        "Text3",...)  
}
```



BOX

Use a **Box** to arrange elements on top of each other:

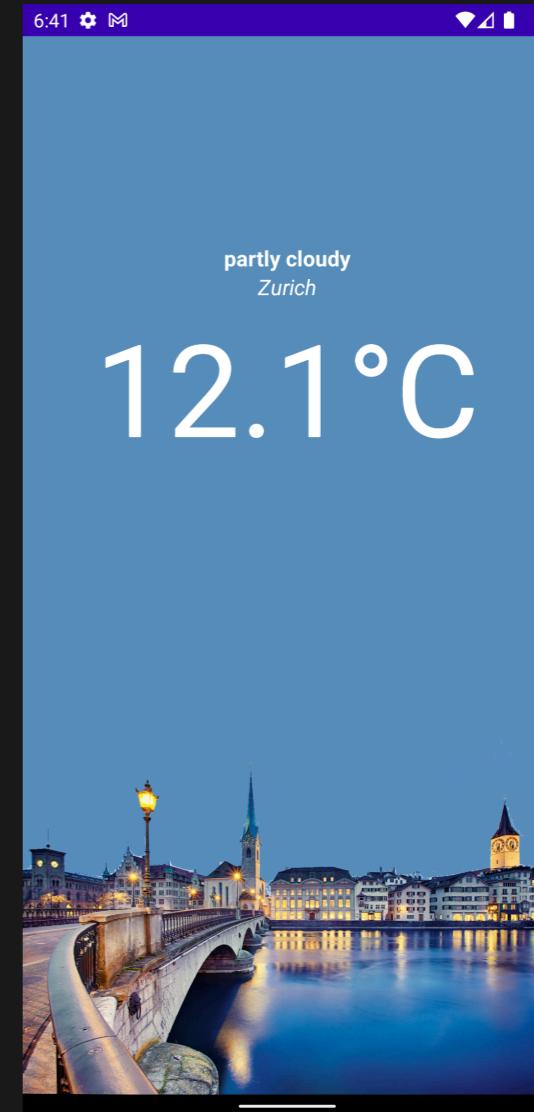
```
Box {  
    Text("Text1, below", fontSize = 32.sp)  
    Text("Text2, above", fontSize = 24.sp)  
}
```



EXERCISE WEATHER APP: BACKGROUND

Use the Box component to add a background image to your Weather App. You can find the background image on Moodle.

Use Drag'n'Drop to add the png to your project's drawable folder,
ContentScale.FillBounds to enlarge the image and *Modifier.zIndex(-1f)* to move the png to the background.



COMPOSITIONS

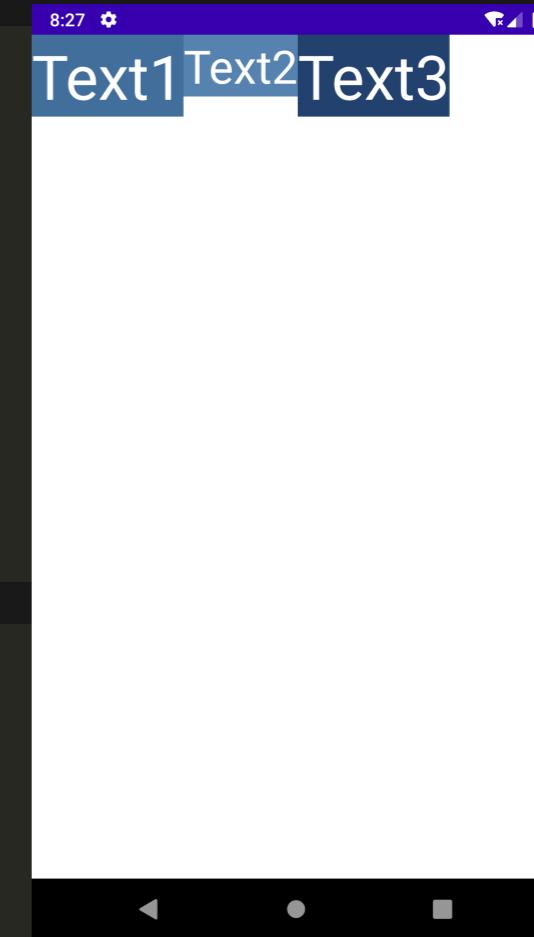
The basic idea of Compose is to create small building blocks that can be nested and reused (so called Compositions):

```
setContent {  
    ComposeExampleTheme {  
        // A surface container  
        //using the 'background' color  
        //from the theme  
        Surface(color = MaterialTheme.colors.background) {  
            TextPresentation()  
        }  
    }  
}
```

```
@Compose  
fun TextPresentation() {  
    Text("Hello, World")  
}
```

COMPOSITION REFACTORING

```
Row {  
    // a composition TextLabel makes it much more readable  
    TextLabel("Text1",  
        Modifier.background(Color(0xFF41729f)))  
    TextLabel("Text2",  
        Modifier.background(Color(0xFF5885af)), 36.sp)  
    TextLabel("Text3",  
        Modifier.background(Color(0xFF274472)))  
}
```



```
@Composable  
fun TextLabel(text: String, modifier: Modifier,  
fontSize: TextUnit = 48.sp) {  
    Text(text, fontSize = fontSize,  
        color = Color.White,  
        modifier = modifier)  
}
```

EXERCISE WEATHER APP: NEXT DAYS

Add next days weather to your layout. You can use Unicode chars (↓ and ↑) for min and max.

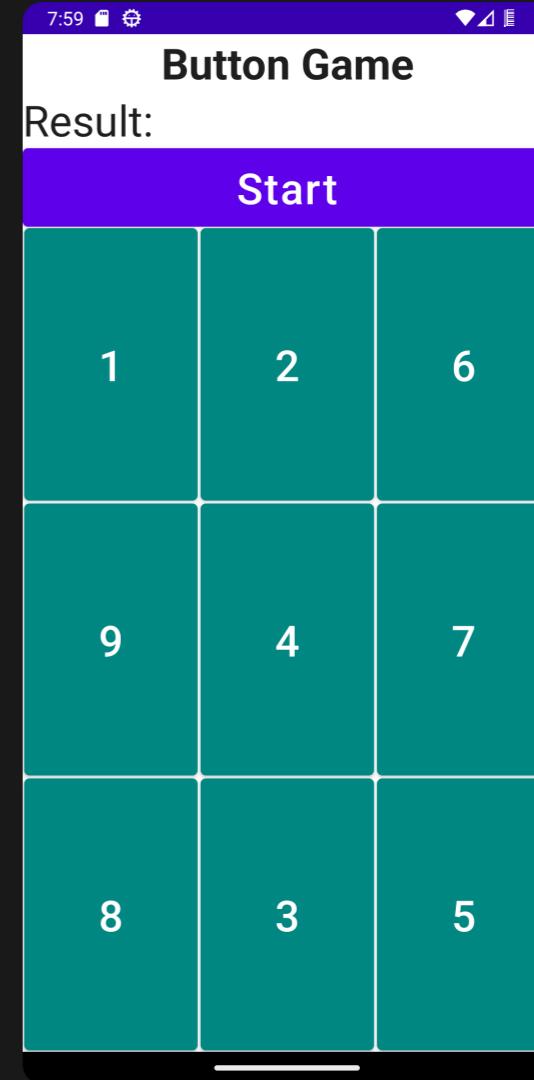


EXERCISE: BUTTON GAME LAYOUT

Create a new project *ButtonGame*.

Implement the following layout. Hints:

- Number buttons use Modifier.weight
- Start button uses fillMaxWidth

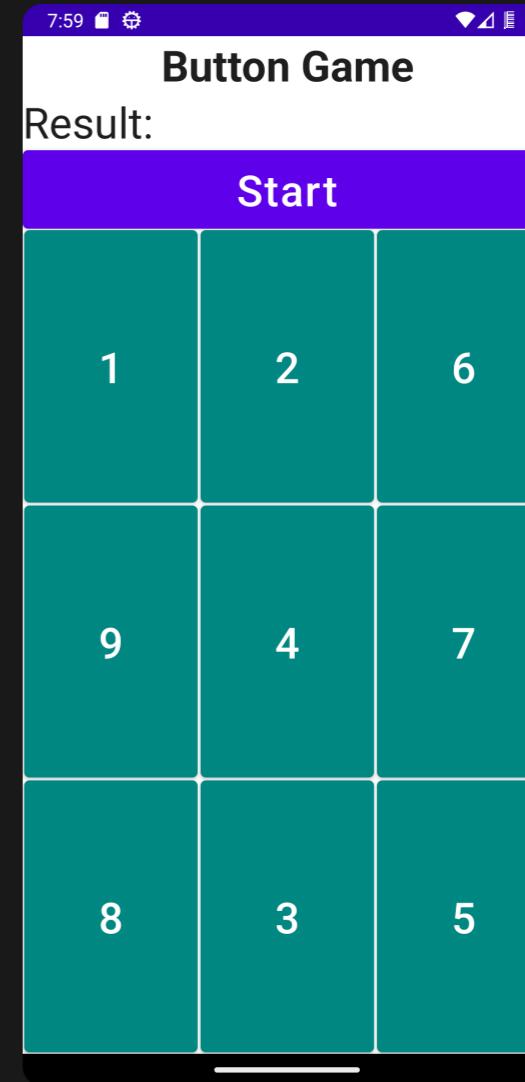


RECAP: BUTTON GAME LAYOUT

Create a new project *ButtonGame*.

Implement the following layout. Hints:

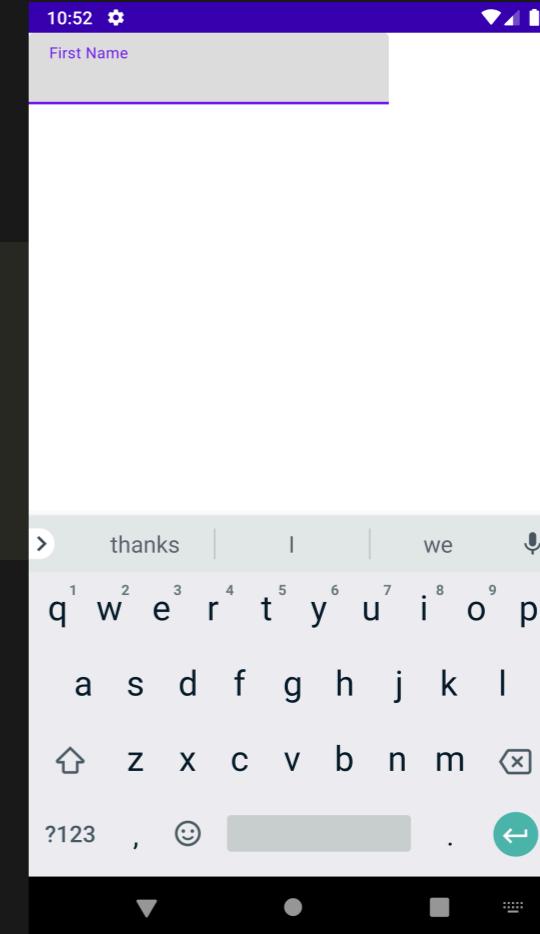
- Number buttons use Modifier.weight
- Start button uses fillMaxWidth



TEXTFIELD I

A TextField can be used to enter Text.

```
TextField(  
    value = "",  
    onValueChange = {},  
    label = {Text("First Name")}  
)
```



TEXTFIELD II

```
TextField(  
    value = "John",  
    onValueChange = {},  
    label = {Text("First Name")},  
    textStyle = TextStyle(color = Color.Blue)  
)
```



STATE MANAGEMENT I

Note that the changing of text won't work in the previous examples! Android Compose uses state management to change the values of an attribute. A state change will automatically trigger a GUI update (recomposition).

STATE MANAGEMENT II

```
var name = remember { mutableStateOf("") }
```

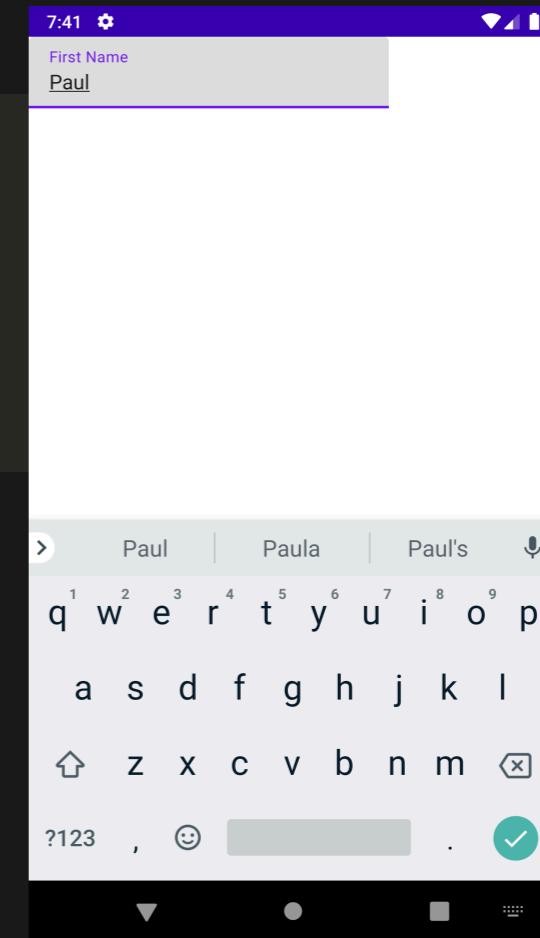
- Needs to be defined inside a composable context
- remember: Value will be saved.
- mutableStateOf: The initialization is provided as parameter

The following import is required:

```
import androidx.compose.runtime.getValue  
import androidx.compose.runtime.setValue
```

STATE MANAGEMENT III

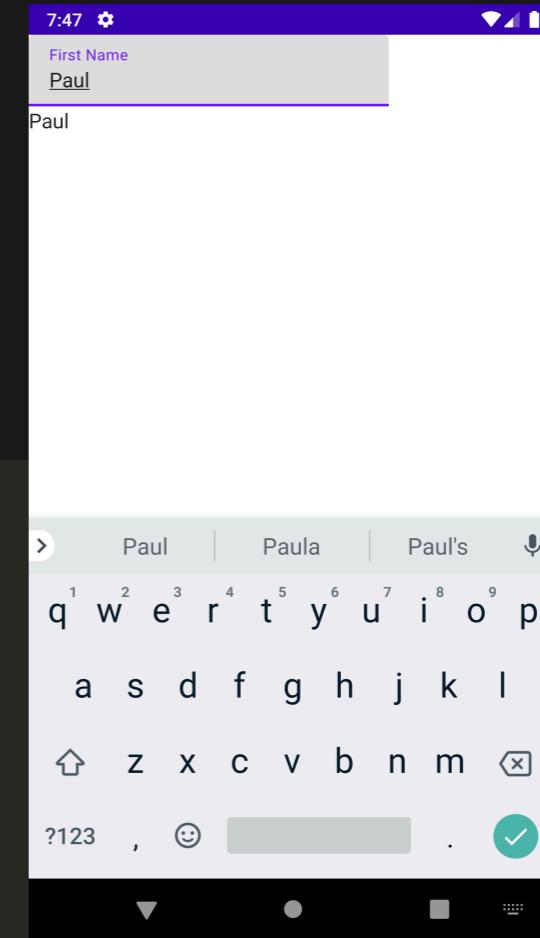
```
var name = remember { mutableStateOf("") }
TextField(
    value = name.value,
    onValueChange = {name.value = it},
    label = {Text("First Name")}
)
```



STATE MANAGEMENT IV

A state variable can be used by multiple components. Whenever the state changes, the component is updated automatically.

```
var name = remember { mutableStateOf("") }  
Column {  
    TextField(  
        value = name.value,  
        onValueChange = {name.value = it},  
        label = {Text("First Name")}  
    )  
    Text(name.value)  
}
```



STATE MANAGEMENT V

```
//it is sometimes necessary to add the type of the variable  
//to mutableStateOf:  
var name = remember { mutableStateOf<String>("") }  
var nameOptional = remember { mutableStateOf<String?>(null) }  
//Note that there are further mutbleState methods for  
//different types are available: mutableDoubleStateOf,  
//mutableDoubleStateOf, mutableIntStateOf,  
//mutableFloatStateOf,...
```

STATE SAVING

when remember is used, the state is only saved during recompositions. If an external event occurs (ie, orientation change), the state is deleted.

Use rememberSaveable to store the state over events:

```
var nameState = rememberSaveable { mutableStateOf("") }  
...
```

TEXTFIELD FOCUS

The TextField won't lose focus automatically. Use a listener on your outer component (ie. Column) to dismiss the keyboard when clicked outside:

```
val focusManager = LocalFocusManager.current
Column(modifier = Modifier
    .fillMaxWidth()
    .fillMaxHeight()
    .pointerInput(Unit) {
        detectTapGestures(onTap = {
            focusManager.clearFocus()
        })
    }
)
```

VIEW MODEL

MOTIVATION

Using the state within composables has a number of drawbacks:

- No separation of ui and business logic
- It is necessary to pass the state variables to other composables. Additionally, changes need also passed down (ie. via closure)

A View Model will solve these problems. Additionally:

- remember / rememberSaveable is not needed anymore (by default, the Lifecycle of a ViewModel is until the app is destroyed)

INSTALLATION

You need to add a library in order to use view models. Open build.gradle.kts (Module:app):

```
dependencies {  
    implementation(  
        "androidx.lifecycle:lifecycle-viewmodel-compose:2.8.6"  
    )  
}
```

Note that a cleaner way to enter additional libraries is using the version catalog.

VIEW MODEL EXAMPLE I

```
//A ViewModel must inherit from ViewModel
class AppViewModel() : ViewModel() {

    var name = mutableStateOf("")

    //it is possible to create methods and
    //change attributes within
    fun changeName() {
        //use .value to access the value of the state
        //variable
        name.value = "New Name from Model State"
    }
}
```

VIEW MODEL EXAMPLE II

```
//access the ViewModel within your activity (or a closure):
val model: UserViewModel by viewModels()
...
//and use it
model.changeName()
```

VIEW MODEL EXAMPLE III

```
//use a model parameter for your component to have immediate  
//access to the ViewModel  
@Compose  
fun AppView{//add a model parameter to your composition  
    //Note that the model is automatically instantiated  
    //Note also that sometimes Android Studio won't find  
    //viewModel(). Import it directly:  
    //import androidx.lifecycle.viewmodel.compose.viewModel  
    model: AppViewModel = viewModel()) { ...  
  
    ...  
    //access the attribute and call methods from the model  
    Text(model.name.value,...)  
    Button(onClick = { model.changeName() },...  
    ...
```

LISTS AND VIEW MODEL

Compose watches changements in state variables and will trigger an update automatically. However, if you change list elements the update won't be triggered automatically.

Solution: Use `mutableStateListOf`:

```
class AViewModel() : ViewModel() {
    var fruits = mutableStateListOf<String>(
        "Apple", "Orange", "Banana") ...

    fun addFruit(name : String) {
        fruits.add(name) // will trigger
    }
}
```

FURTHER VIEW MODEL FACTS

- The ViewModel is available in all! composables. Even better: It is always the same instance!
- It is sometimes necessary to access the context within your View Model class. This can easily be achieved by inheriting from ApplicationViewModel:

```
class WeatherViewModel(application: Application) :  
    AndroidViewModel(application) {  
    private val context =  
        getApplication<Application>().applicationContext  
    ...
```

and use it:

```
@Composable  
fun MyComposable(model: WeatherViewModel = viewModel())
```

CONTEXT

Use the **Context** to access everything in the application environment, ie:

- Ressource Files
- Application Directories (Home and Temporary)
- Application Services

To read a file from the resources, use the following line:

```
//the text file is stored under res/raw/text_file.txt  
val tFile = context.resources.openRawResource(R.raw.textFile)
```

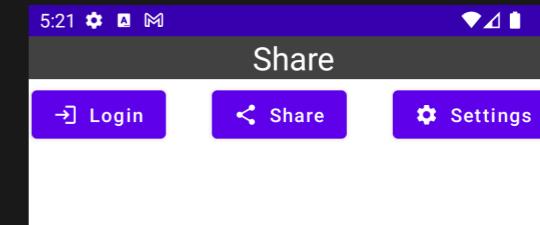
ANDROIDVIEWMODEL

Use an AndroidViewModel when a Context is required.

```
class AppViewModel(application: Application) :  
    //Note that this ViewModel inherits from AndroidViewModel  
    //(and not ViewModel)  
    AndroidViewModel(application) {  
        //Only available in AndroidViewModel  
        private val context =  
            getApplication<Application>().applicationContext  
    }
```

EXERCISE TEXTCHANGE APP: LOGIC

Implement the following logic in your TextChange App: Whenever a button is clicked, the Text changes to the name of the button (Login, Share, Settings). Use a ViewModel:



- Install the necessary library viewmodel-compose
- Create a TextChangeViewModel class that has one attribute buttonText and one method changeButtonText
- Add the model as parameter to your component
- Call changeButtonText when a button is clicked
- Read model.buttonText in the Text composable

SERVER CALLS I

In order to retrieve data from a server, you need to have Internet permission. This is added in your manifest file:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="zhaw.ch.gsontest">

    <uses-permission android:name="android.permission.INTERNET" />

    <application ...
</manifest>
```

SERVER CALLS II

There are different ways to retrieve data from a server:

1. Your own implementation that runs in its own thread.
2. A library like Volley that does the work for you.

SERVER CALLS: VOLLEY

1. Add volley to your dependencies:

```
implementation ("com.android.volley:volley:1.2.1")
```

2. Call the server.
3. Retrieve either the answer from the server or an error.

SERVER CALLS: VOLLEY SERVER CALL

```
//create a request queue
val requestQueue = Volley.newRequestQueue(context)
//define a request.
val request = StringRequest(
    Request.Method.GET, ENDPOINT,
    { response ->
        //find the response string in "response"
        Log.i("Volley", response)
    },
    Response.ErrorListener {
        Log.e("Volley", "Error loading data: $it")
    }
)

//add the call to the request queue
requestQueue.add(request)
```

EXERCISE WEATHER APP: SERVER CALL

Implement a server call to the following [API](#):

```
https://api.open-meteo.com/v1/forecast?latitude=47.37&longitude=8.55&daily=temperature_2m_max,temperature_2m_min&current_weather=true&timezone=Europe%2FZurich
```

Log the received string on the console.

Hint: Implement an `AndroidViewModel` first:

```
class WeatherViewModel(application: Application) :  
    AndroidViewModel(application) {  
    private val context =  
        getApplication<Application>().applicationContext  
    init { //init is called when the class is instantiated  
        loadWeatherData()  
    }...  
}
```

JSON PARSING

Typically, APIs return a JSON structure that can be parsed by:

- an integrated stream parser.
- an external library like Klaxon or GSON that converts the JSON to a set of objects.

KLAXON: PROCESS

1. Add Klaxon to your dependencies:

```
implementation ("com.beust:klaxon:5.5")
```

2. Create a backing object (tree) that matches your JSON structure.
3. Read JSON from file or server.
4. Convert JSON to your backing object.

KLAXON: JSON EXAMPLE

File persons.json:

```
{  
  "persons":  
    [{"name": "Peter Muster", "street": "Musterstrasse 1"},  
     {"name": "Paul Frey", "street": "Bahnhofstrasse 12"},  
     {"name": "Erwin Dobler", "street": "Brückstrasse 16"},  
     {"name": "Walter Loder", "street": "Bachweg 17"},  
     {"name": "Maria Weibel", "street": "Aareweg 5"}]  
}
```

KLAXON: BACKING OBJECT

```
class Person(val name: String, val street : String) {}
```

```
class Persons(val persons : MutableList<Person>) {}
```

Note that you only need to define the properties you are interested in!

KLAXON: PARSING

```
val jsonString = "{\"persons\" : [{\"name\" : ...}]}"  
//convert  
val persons = Klaxon().parse<Persons>(inputStream)  
//use persons...
```

EXERCISE WEATHER APP: PRESENT DATA

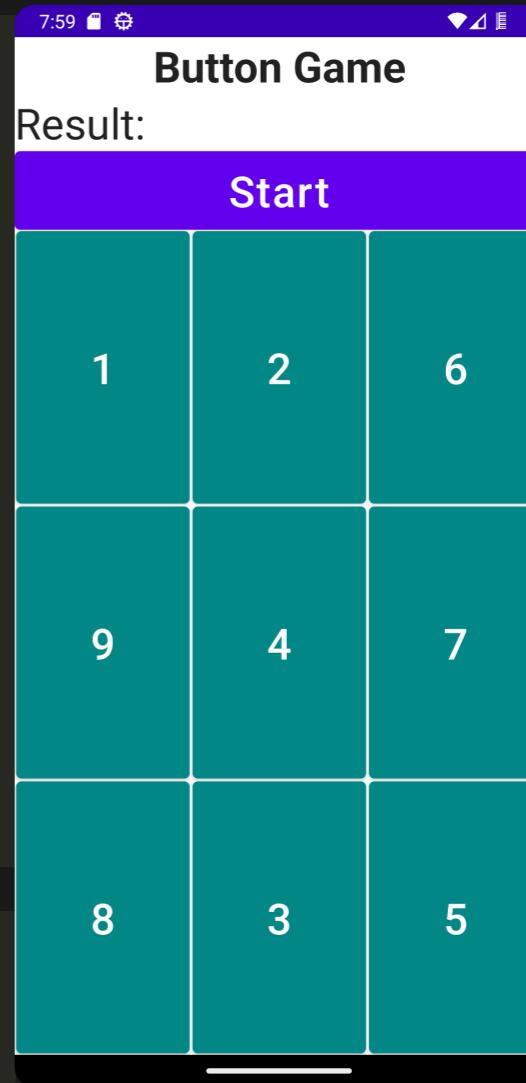
Parse and show the received JSON Data:

```
fun weatherCodeTitle(weatherCode : Int?) : String {  
    return when (weatherCode) { //use weathercode  
        0 -> "Clear sky"  
        1 -> "Mainly clear"  
        2, 3 -> "Partly Cloudy"  
        in 40..49 -> "Fog or Ice Fog"  
        in 50..59 -> "Drizzle"  
        in 60..69 -> "Rain"  
        in 70..79 -> "Snow Fall"  
        in 80..84 -> "Rain Showers"  
        85, 86 -> "Snow Showers"  
        87, 88 -> "Shower(s) of Snow Pellets"  
        89, 90 -> "Hail"  
        in 91..99 -> "Thunderstorm"  
        else -> "unknown $weatherCode"    } }
```



RECAP: BUTTON GAME

```
//Implement the Button Game Logic: After the user has  
//clicked "START", one button is randomly chosen and the  
//background color put to green. The user must click the  
//chosen button fast. Show the elapsed time.  
  
//random number between 1 and 9 (inclusive)  
colorButtonNum = Random.nextInt(1, 9)  
//get time in millis  
val time = System.currentTimeMillis()  
//converting double d to String with two decimal places  
val dStr = "%,.2f".format(d)  
//use a mutableStateList to manage the background color:  
var backgroundColor = mutableStateListOf<Color>(  
    Color.Green, Color.Green, ...)  
  
//setting the background color of a button  
Button(onClick = {model.buttonClicked(num)},  
    colors = ButtonDefaults.buttonColors(  
        containerColor = model.backgroundColor[num] ))  
    {Text("1")} }
```



EXERCISE SBB APP I

Create a new Project SBB App. This app should present the next trains leaving Zurich SBB station. The data can be retrieved from the following URL:

```
https://transport.opendata.ch/v1/stationboard?station=Zurich
```

Create the backing objects for the JSON parsing first. The list of trains can be found under key "stationboard". Each entry has a number of fields, try to read "category", "number", "to" and "platform". Note that number can be null!

EXERCISE SBB APP II

Enhance your SBB app by:

- Adding viewmodel-compose, Klaxon and Volley libraries
- Create an AndroidViewModel (don't forget permissions):

```
class SBBModel(application: Application) : AndroidViewModel(application) {  
    private val context = getApplication<Application>().applicationContext  
    //why do we need a mutable list (and not an immutable)?  
    var entries : MutableList<StationboardEntry> = mutableStateListOf()  
    init {  
        loadJSON()  
    }  
    fun loadJSON() {...  
        val sb = Klaxon().parse<Stationboard>(response)  
        //why do we need to call addAll (and not entries = sb!!.stationboard)?  
        entries.addAll(sb!!.stationboard.toMutableList())  
    }  
}
```

- Init your model in your layout:

```
setContent {  
    val model : SBBModel by viewModels()  
    ...  
    Greeting("Android ${model.toString()}")  
    ...  
}
```

NAVIGATION I

Unfortunately, Navigation is not included in the standard libraries that are installed automatically. Instead, add the following lines to your build.gradle:

```
implementation(  
    "androidx.navigation:navigation-compose:2.6.0")
```

NAVIGATION II

- In Compose, a page is simply a composable.
- Each composable needs a label.
- A navigation controller must be instantiated:

```
val navController = rememberNavController()
```

- All composables should be wrapped in a NavHost that orchestrates the different pages (definition of routes):

```
NavHost(navController = navController, startDestination = "start") {  
    composable("start") { ... }  
    composable("detail") { ... }  
}
```

NAVIGATION III

Right now, navigation will always create a new model and navController. Therefore, as these components will most likely be shared, they need to be passed down:

```
val navController = rememberNavController()
val model : AViewModel = AViewModel(this.application)
NavHost(navController = navController, startDestination = "list") {
    composable("list") {
        ListView(navController = navController, model = model)
    }
    composable("detail") {
        DetailView(navController = navController, model = model)
    }
}
```

NAVIGATION IV

The following line will navigate between composables:

```
navController.navigate("detail")
```

navigate can be called from a clickable element (ie. Button).

NAVIGATION PARAMETERS I

```
//add parameters to your route definition:  
composable("detail/{index}",  
    arguments = listOf(navArgument("index") { type = NavType.IntType })  
    { backStackEntry ->  
        val index = backStackEntry.arguments!!.getInt("index")  
        StationboardEntryDetail(index=index, model = model)  
    }  
}
```

NAVIGATION PARAMETERS I

```
//and provide them:  
navController.navigate("detail/123")  
  
//Hint: Don't pass complex objects, instead, use a ViewModel
```

LISTS I

Use a column and forEach to present all elements of a list:

```
val names = mutableListOf(  
    "Henry Fonda", "John Wayne",  
    "Cary Grant", "Steve McQueen",  
    "Paul Newman", "Robert Redford")
```

```
Column {  
    names.forEach { name ->  
        Text(name)  
    }  
}
```



LISTS II

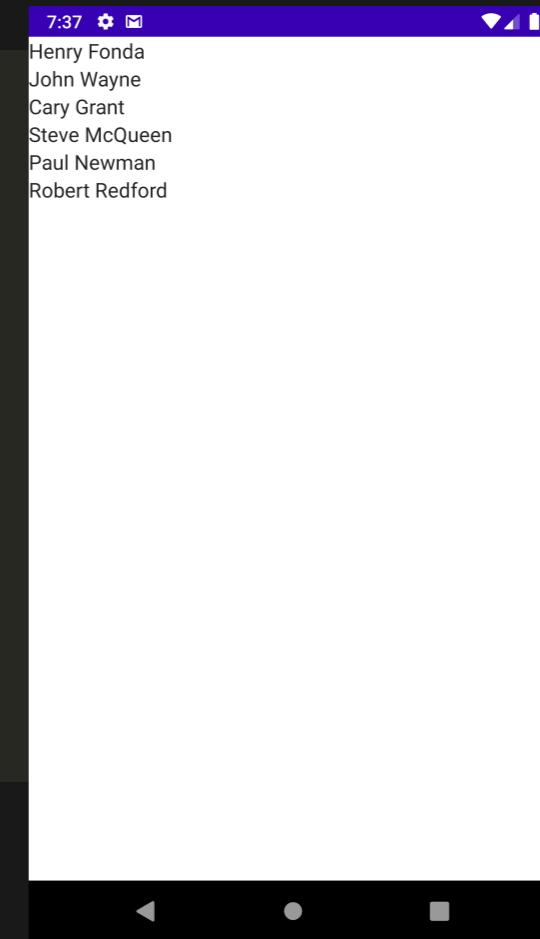
The previous approach works well for short lists, however, when using larger lists, the performance is bad as all rows are calculated before presenting.

Idea: Use LazyRow or LazyColumn. These components only calculate the visible rows/columns.

LazyRow and LazyColumn require the method `itemsIndexed()` (`androidx.compose.foundation.lazy`) to parse a list (instead of `forEach`).

LISTS III

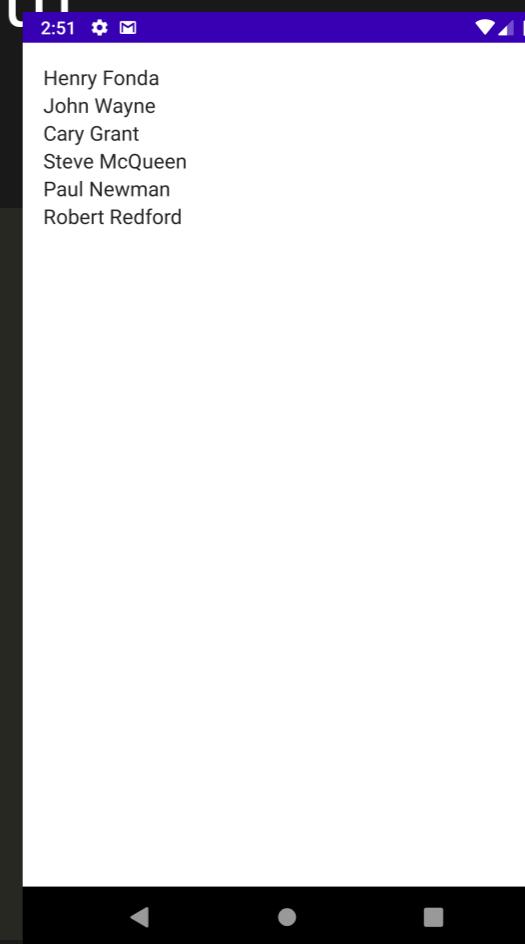
```
val names = mutableListOf(  
    "Henry Fonda", "John Wayne",  
    "Cary Grant", "Steve McQueen",  
    "Paul Newman", "Robert Redford")  
  
LazyColumn {  
    //if you need the index and the value, use  
    //itemsIndexed. Otherwise, use items (next slide)  
    itemsIndexed(names) { index, name ->  
        Text(name)  
    }  
}
```



LIST CELL FORMAT I

Adding a padding on the outside of the list with contentPadding:

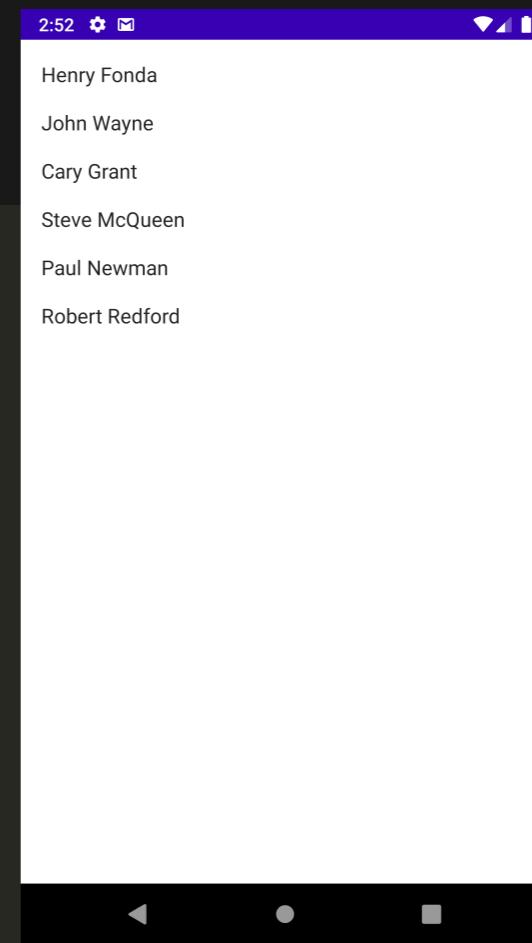
```
val names = mutableListOf(  
    "Henry Fonda", "John Wayne",  
    "Cary Grant", "Steve McQueen",  
    "Paul Newman", "Robert Redford")  
  
LazyColumn(contentPadding =  
    PaddingValues(horizontal = 16.dp, vertical = 16.dp))  
{  
    items(names) { name ->  
        Text(name)  
    }  
}
```



LIST CELL FORMAT II

Adding a spacing between the cells with verticalArrangement:

```
val names = mutableListOf(  
    "Henry Fonda", "John Wayne",  
    "Cary Grant", "Steve McQueen",  
    "Paul Newman", "Robert Redford")  
  
LazyColumn(contentPadding =  
    PaddingValues(horizontal = 16.dp, vertical = 16.dp),  
    verticalArrangement = Arrangement.spacedBy(16.dp))  
{  
    items(names) { name ->  
        Text(name)  
    }  
}
```



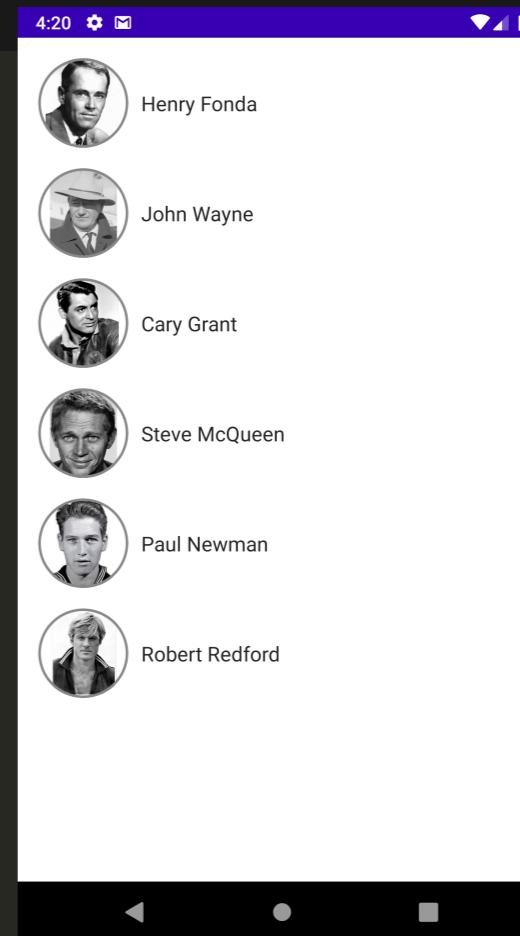
LIST CELL FORMAT III

A list cell can show text and images:

```
class Actor(val firstName : String,  
           val lastName : String) {  
    val fullName = "$firstName $lastName"  
  
    @Composable  
    fun getImage(context: Context) : ImageBitmap {  
        //use the last name in lowercase to retrieve the  
        //corresponding image from resources/raw folder  
        val name: Int =  
            context.resources.getIdentifier(  
                lastName.lowercase(), "raw",  
                context.packageName)  
  
        return ImageBitmap.imageResource(name)  
    }  
}
```

LIST CELL FORMAT IV

```
items(actors) { actor ->
    //each list cell is one row that centers the
    //content vertically
    Row(verticalAlignment = Alignment.CenterVertically) {
        //the image is scaled to fit with size 70 dp,
        //a circle is clipped and a round border is added
        Image(actor.getImage(applicationContext),
            "${actor.fullName}",
            contentScale = ContentScale.Fit,
            modifier = Modifier.size(70.dp)
                .clip(CircleShape)
                .border(2.dp, Color.Gray, CircleShape))
        Spacer(modifier = Modifier.width(10.dp))
        Text(actor.fullName)
    }
}
```



EXERCISE SBB APP III

Add a layout to your SBB App and present:

- "category" and "number"
- "to"
- "platform"

8:15	
S16	43/44
Herrliberg-Feldmeilen	
S12	43/44
Schaffhausen	
S16	41/42
Zürich Flughafen	
S2	32
Ziegelbrücke	
S7	41/42
Winterthur	
S14	31
Affoltern am Albis	
S19	33
Effretikon	
S24	4
Zug	
S15	41/42
Niederweningen	
S5	43/44
Pfäffikon SZ	
S8	34
Winterthur	
S3	41/42

PERMISSIONS

PERMISSIONS IN ANDROID I

- Every app runs in a sandbox, special capabilities need permissions.
- All permissions are identified by a unique **label**.
- You can grant/deny permissions for an app in the settings (Settings->Apps->(Choose app)).

PERMISSIONS IN ANDROID II

Using capabilities without permissions will result in an exception:

```
Caused by: java.lang.SecurityException: Permission Denial: opening
provider com.android.providers.contacts.ContactsProvider2 from
ProcessRecord{2c165c26436:ch.zhaw.contactapplication/u0a112}
(pid=6436, uid=10112) requires
android.permission.READ_CONTACTS or
android.permission.WRITE_CONTACTS
```

DECLARING PERMISSIONS

- Until Android 5.1, permissions were only defined in the manifest file. That changed to a dynamic approach where users have to allow a permission during runtime.
- However, you still need to declare your permissions in Android manifest with a element! If you forget the manifest declaration, your permission is automatically declined during runtime.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
    package="ch.zhaw.contactapplication">  
    <uses-permission android:name="android.permission.READ_CONTACTS" />  
    <application ...>
```

DANGEROUS PERMISSIONS

- Android manages permissions in groups. If you already permit access to READ_CONTACTS, Android will automatically grant WRITE_CONTACTS as both are in the same group.
- Until Android 6, permissions were only declared in the manifest. Since then, Android distinguishes between normal and dangerous protection levels. Normal permissions (ie INTERNET) will be granted automatically when the permission is declared in the manifest, dangerous permissions need to be allowed by the user explicitly.

PERMISSIONS AND COMPOSE

Use the (experimental) library [Accompanist](#) to implement a permission routine in compose.

READ CONTACTS EXAMPLE I

```
@OptIn(ExperimentalPermissionsApi::class)
@Composable //the contentResolver is needed for reading contacts
fun ContactsView(contentResolver : ContentResolver) {
    var contacts = remember { mutableStateListOf<String>() }
    //the closure will be called after the allow/deny
    //dialog has been shown to the user. If the result
    //is true (permission granted), the contacts can be read
    val contactPermissionState = rememberPermissionState(
        android.Manifest.permission.READ_CONTACTS
    ) {
        if (it) {
            readContacts(contentResolver, model.contacts)
        }
    }
}
```

READ CONTACTS EXAMPLE II

```
Column {  
    //the button click will call permissionReadContacts  
    Button(onClick = {permissionReadContacts(contactPermissionState,  
        contentResolver, contacts) }) {  
        Text("Read Contacts")  
    }  
    //the text shows the current permission state  
    Text(getPermissionStateText(contactPermissionState))  
    LazyColumn { //all contacts will be shown in a list  
        items(contacts) { contact ->  
            Text(contact) } } }
```

READ CONTACTS EXAMPLE III

```
//returns the permission state as a string
@OptIn(ExperimentalPermissionsApi::class)
fun getPermissionStateText(  
    contactPermissionState : PermissionState) : String {  
    if (contactPermissionState.status.isGranted) {  
        return "Contact permission Granted"  
    }  
    else {  
        return "No Permission to read Contacts"  
    }  
}
```

READ CONTACTS EXAMPLE IV

```
//reads the contacts (if permission is granted)
//or launches a permission request (allow/deny dialog)
@OptIn(ExperimentalPermissionsApi::class)
fun permissionReadContacts(
    contactPermissionState : PermissionState,
    contentResolver: ContentResolver,
    contacts : MutableList<String>) {

    if (!contactPermissionState.status.isGranted) {
        contactPermissionState.launchPermissionRequest()
    }
    else {
        readContacts(contentResolver, contacts)
    }
}
```

READ CONTACTS EXAMPLE V

```
fun readContacts(contentResolver: ContentResolver,  
    contacts : MutableList<String>) {  
    val uri: Uri = ContactsContract.Contacts.CONTENT_URI  
    //the array defines the fields to be returned  
    val cursor: Cursor? =  
        contentResolver.query(  
            uri, arrayOf(  
                ContactsContract.Contacts._ID,  
                ContactsContract.Contacts.DISPLAY_NAME,  
                ContactsContract.Contacts.HAS_PHONE_NUMBER  
            ),  
            null, null, null  
        )  
}
```

READ CONTACTS EXAMPLE VI

```
if (cursor != null) {
    var res: Boolean = cursor.moveToFirst()
    while (res) {
        //read the id (id == 0) first
        val id = cursor.getString(0)
        //the name (id == 1)
        var text: String = cursor.getString(1)
        //unfortunately, there is no "getBool"
        if (cursor.getString(2) == "1") {
            text += " " + getPhoneNumberForUser(contentResolver, id)
        }
        Log.i("ContactExample", text)
        contacts.add(text)
        res = cursor.moveToNext()
    }
}
```

READ CONTACTS EXAMPLE VII

```
fun getPhoneNumberForUser(contentResolver: ContentResolver,  
    contactId: String): String? {  
    var number = ""  
    val cursor = contentResolver.query(  
        ContactsContract.CommonDataKinds.Phone.CONTENT_URI, null,  
        ContactsContract.CommonDataKinds.Phone.CONTACT_ID + " = " +  
        contactId, null, null)  
    if (cursor!!.moveToFirst()) {  
        while (!cursor.isAfterLast) {  
            val index = cursor.getColumnIndex(  
                ContactsContract.CommonDataKinds.Phone.NUMBER)  
            number += cursor.getString(index) + " "  
            cursor.moveToNext()  
        }  
        cursor.close(); return number}  
}
```

EXERCISE SBB APP IV

Finally, display the departure timestamp.

You could use the following function to convert the epoch time to hh:mm:

```
fun getDateTime(l: Long): String {
    try {
        val sdf = SimpleDateFormat("HH:mm")
        sdf.timeZone = TimeZone.getDefault()
        val netDate = Date(l * 1000)
        return sdf.format(netDate)
    } catch (e: Exception) {
        return e.toString()
    }
}
```



SCHEDULED TASKS WITH WORKMANAGER

Periodic and scheduled tasks (backup, synchronizing of data with a server, news/message downloading, etc.) can be done with WorkManager.

WorkManager is not intended for in-process background work that can safely be terminated if the app process goes away (use Coroutines for this).

WorkManager manages a task that is defined in Worker and configured with a WorkRequest.

PREPARATION

Define if the type of work should be run once or periodic.
Periodic workers can be configured with constraints (ie once per day) and are executed automatically (execution may be delayed because WorkManager is subject to OS battery optimizations).

Import the library:

```
implementation("androidx.work:work-runtime-ktx:2.8.1")
```

WORKER

Implement a worker that executes the task:

```
//this TestWorker only runs once (but can be restarted)
class TestWorker(context : Context,
    workerParams : WorkerParameters) :
    Worker(context, workerParams) {
    override fun doWork(): Result {
        Log.i("WORKER", "Starting TestWork")
        //calculate a result and send it back to the app
        val outputData: Data = Data.Builder()
            .putString("MESSAGE", "This is the calculated result")
            .build()
        return Result.success(outputData)
    }
}
```

WORKMANAGER DEFINITION

The WorkManager starts the worker:

```
val workRequest =  
    OneTimeWorkRequestBuilder<TestWorker>()  
        .build()  
  
    //connect an observer to the workRequest (to retrieve  
    //the result)  
    WorkManager.getInstance(context)  
        .getWorkInfoByIdLiveData(workRequest.id)  
        .observeForever(Observer {  
            if (it.state == WorkInfo.State.SUCCEEDED) {  
                Log.i("SERVICE", "${it.outputData.getString("MESSAGE")}")  
            }  
        } )
```

WORKMANAGER START

Finally, start the WorkManager

```
//make a call somewhere in your app, ie. in your ViewModel
val continuation = WorkManager.getInstance(context)
    .enqueueUniqueWork(
        "TESTWORKER",
        ExistingWorkPolicy.REPLACE,
        workRequest
    )
```

WORKMANAGER DISCUSSION

A unique Worker is terminated after 10 minutes.

A periodic worker is automatically started within the provided time interval. However, the minimum time interval is 15 minutes!

COROUTINES

- It is not allowed to access certain resources (internet, database, ...) in the main thread.
- AsyncTask is the old way of dealing with this problem. However, AsyncTask is cumbersome. Coroutines solve this task in a more elegant way.

COROUTINE DEFINITION

Coroutine functions are marked *suspend*:

```
suspend fun writeInitData(context : Context) {
```

Coroutine functions have to be called from a different thread by using a scope:

```
//call the coroutine suspend function on the IO thread
//this won't block the UI
viewModelScope.launch(Dispatchers.IO) {
    writeInitData(context)
}
//Note that you could omit the (Dispatchers.IO) launch parameter.
//Then, the coroutine would be executed on the main thread.
```

COROUTINE SCOPES

One of the major problems when dealing with multithreading is the correct termination of threads. Coroutines solve this by attaching them to a scope. In the example from the last slide, any coroutine launched in the `viewModelScope` scope is automatically canceled if the `ViewModel` is cleared.

There is also a `LifecycleScope` and `LiveDataScope` [available](#).

PERSISTENCE

SHARED PREFERENCES I

The easiest way to store data locally is by using
SharedPreferences:

```
val settings = context.getSharedPreferences("prefsfile",  
    Context.MODE_PRIVATE)  
  
//it is important that you get an editor reference!  
val editor = settings.edit()  
  
//save strings, booleans, floats, ints, longs, stringsets  
editor.putString("USER_NAME", "john.ford@hollywood.com")  
  
//and commit your changes  
editor.commit()
```

SHARED PREFERENCES II

Reading preferences is a two liner:

```
val settings = getSharedPreferences("prefsfile",
    Context.MODE_PRIVATE)

val defaultUser = settings.getString("USER_NAME",
    "no-user@no-domain.com")
```

SQLITE

SQLite is an SQL database developed for smaller devices/scenarios:

- It only needs one file to store the data
- It is self-contained and serverless
- It supports transactions

ROOM

- Google recommends to use the library Room for accessing SQLite.
- Insert the following entry to your build.gradle:

```
plugins {  
    ...  
    id("kotlin-kapt")  
    ...  
}  
  
dependencies {  
    implementation("androidx.room:room-runtime:2.5.2")  
    annotationProcessor("androidx.room:room-compiler:2.5.2")  
    kapt ("androidx.room:room-compiler:2.5.2")  
}
```

ROOM: NEWER JAVA VERSION FOR KAPT

Room requires kapt (Kotlin Annotation Processing Tool). This tool runs on a newer Java Version (Java 17) than the default one (Java 8 (version string 1.8)).

Therefore, in order to use kapt, it is required to update to a newer Java version:

```
compileOptions {  
    sourceCompatibility = JavaVersion.VERSION_17  
    targetCompatibility = JavaVersion.VERSION_17  
}  
kotlinOptions {  
    jvmTarget = "17"  
}
```

ROOM: ENTITY

Define an entity to store in the database:

```
@Entity  
class Person(  
    @PrimaryKey(autoGenerate = true) val uid: Int = 0,  
    @ColumnInfo(name = "first_name") val firstName: String?,  
    @ColumnInfo(name = "last_name") val lastName: String?  
)
```

ROOM: DAO I

```
//DAO: Database Access Object
@Dao
interface PersonDao {
    @Query("SELECT * FROM person")
    fun getAll(): List<Person>

    @Query("SELECT * FROM person WHERE uid IN (:userIds)")
    fun loadAllByIds(userIds: IntArray): List<Person>

    @Query("SELECT * FROM person WHERE first_name IN (:names)")
    fun loadAllByFirstName(names: List<String>): List<Person>
}
```

ROOM: DAO II

```
@Query("SELECT * FROM person WHERE first_name LIKE :first AND " +  
        "last_name LIKE :last LIMIT 1")  
fun findByName(first: String, last: String): Person  
  
@Insert  
fun insertAll(vararg persons: Person)  
  
@Delete  
fun delete(person: Person)  
}
```

ROOM: APPDATABASE

```
@Database(entities = arrayOf(Person::class),  
          version = 1, exportSchema = false)  
abstract class AppDatabase : RoomDatabase() {  
    abstract fun personDao(): PersonDao  
}
```

```
val db = Room.databaseBuilder(  
    applicationContext,  
    AppDatabase::class.java, "database-name"  
).build()
```

ROOM: INSERT METHOD

Use keyword suspend:

```
suspend fun insertUsers() {
    db?.personDao()?.insertAll(
        Person(firstName="Peter", lastName="Muster"),
        Person(firstName="Paul", lastName="Muster"))
}

suspend fun readUsers() {
    //we are in a background thread and need to call postValue
    persons.postValue(db?.personDao()?.getAll()?.toMutableList())
}
```

COROUTINES: CALL ASYNC

Call this method from our ViewModel:

```
viewModelScope.launch(Dispatchers.IO) {  
    model.insertUsers()  
}  
//readData is now executed in a different thread and won't block  
//the main ui thread
```

EXERCISE DB APP IV

Implement a DB to the DB App:

- Define a DAO where movies can be retrieved from and written to the database (getAll, InsertAll)
- Implement the AppDatabase class
- Add three methods in the ViewModel: initDB, readMovies and writeMovies

```
//writing all movies from a list can be done with one line:  
db?.movieDao()?.insertAll(*movies.toTypedArray())  
  
//after you have read the data, you need to post them (because you)  
//are in your own thread (movies is a LiveData object)  
movies.postValue(allMovies)
```

EXERCISE DB APP V

Finally, download the movies.json file from moodle, read it with Klaxon and write all movies to the database. Note that it will take a minute or two for this process.

ANDROID SERVICES

SERVICE ARCHITECTURE I

*Source: Google Android Play Service
Documentation*

SERVICE ARCHITECTURE II

- The Play service is executed as background task.
- New service versions are delivered by the Play Store.
This ensures that the newest version is available
(Google controls updates).
- Lately, Play Services have been criticized for sending
personal data to Google servers regularly (see [here](#))

AVAILABLE SERVICES

Google Mobile Ads, Android Advertising, Google Analytics,
Google Sign In, SMS Receiver, Google Awareness, Google
Cast, Google Fit Api, Google Play Games, Google Maps SDK,
ML Kit, Nearby Device, Mobile Vision, Tensor Flow, Google
Pay, ...

SETUP

- Add the desired service to your build.gradle, ie:

```
implementation 'com.google.android.gms:play-services-vision:11.8.0'
```

- Check if the service is available using

```
val googleAPI = GoogleApiAvailability.getInstance()
val isAvailable =
    googleAPI.isGooglePlayServicesAvailable(applicationContext)
if (isAvailable != ConnectionResult.SUCCESS) {
    //show an error message if it is not available
}
```

- Use the Service Client Api (some Apis require authentication)

BARCODE SCANNER I

```
//CAMERA permission is needed!
val detector = BarcodeDetector.Builder(activity.applicationContext)
    .setBarcodeFormats(Barcode.EAN_13)
    //Barcode.DATA_MATRIX | Barcode.QR_CODE)
    .build()

val cameraSource =
    CameraSource.Builder(activity.applicationContext, detector)
    .setFacing(CameraSource.CAMERA_FACING_BACK)
    .setRequestedFps(15.0f)
    .setAutoFocusEnabled(true)
    .build()

if (!detector.isOperational) {
    return "Could not set up the detector!"
}
```

Barcode Scanner II

```
detector.setProcessor(object : Detector.Processor<Barcode> {
    override fun release() {}
    override fun receiveDetections(detections: Detections<Barcode>) {
        if (detections.detectedItems.size() > 0) {
            val barcode = detections.detectedItems.valueAt(0)
            val value = barcode.rawValue
            Log.i("MainActivity", "found barcode: $value")
            activity.runOnUiThread { //stop the camera
                cameraSource.stop()
                Log.i("MainActivity", "camera stopped")
            }
        } else {
            Log.i("MainActivity", "scanning...")
        }
    }
}
```

ANDROID ACTIVITY RECOGNITION

- Recognizes the user's activity changes, ie. driving, walking or biking.
- Uses device sensors to detect the activity. The data is processed using machine learning models.
- Provides two different APIs: The Activity Recognition Transition API and the Activity Recognition Sampling API. Transition is the preferred way, Sampling can be used to finegrain the frequency of the recognition (used in the example on the next slides simply because it is a better demo).

SETUP I

- Add the service to your build.gradle, ie:

```
implementation 'com.google.android.gms:play-services-location:1'
```

- Add the needed permission:

```
<= API 28:  
com.google.android.gms.permission.ACTIVITY_RECOGNITION  
> API 28:  
android.permission.ACTIVITY_RECOGNITION
```

- Check if the service is available:

```
val googleAPI = GoogleApiAvailability.getInstance()  
val isAvailable =  
googleAPI.isGooglePlayServicesAvailable(applicationContext)  
if (isAvailable != ConnectionResult.SUCCESS) {  
//show an error message if it is not available  
}
```

SETUP II

```
//store the client as member variable so that it won't be deleted
activityRecognitionClient = ActivityRecognition.getClient(context)
//create a request by specifying all transitions (next slide)
val request = ActivityTransitionRequest(createAcitivtyTransitions())
//create an Intent and set a unique action
//string (RECOGNITION_ACTION)
val intent = Intent("RECOGNITION_ACTION")
//start the recognition
val task = activityRecognitionClient?.requestActivityUpdates(0,
PendingIntent.getBroadcast(context, 0, intent, 0))
```

SETUP III

```
fun createActivityTransitions() : MutableList<ActivityTransition> {
    val transitions = mutableListOf<ActivityTransition>()
    transitions +=  
        ActivityTransition.Builder()  
            .setActivityType(DetectedActivity.IN_VEHICLE)  
            .setActivityTransition(  
                ActivityTransition.ACTIVITY_TRANSITION_ENTER)  
            .build()  
    transitions +=  
        ActivityTransition.Builder()  
            .setActivityType(DetectedActivity.IN_VEHICLE)  
            .setActivityTransition(  
                ActivityTransition.ACTIVITY_TRANSITION_EXIT)  
            .build()  
    return transitions
```

RETRIEVE RECOGNITIONS I

- The service will send an intent to the app. Therefore, a BroadcastReceiver is needed to retrieve the intent.
- The BroadcastReceiver has to implement the onReceive method:

```
class MyBroadcastReceiver() : BroadcastReceiver() {  
  
    override fun onReceive(context: Context, intent: Intent) {  
        if (intent != null &&  
            intent.getAction() == "RECOGNITION_ACTION") {  
            if (ActivityRecognitionResult.hasResult(intent)) {  
                val intentResult = ActivityRecognitionResult  
                    .extractResult(intent);  
  
                val detectedActivity = intentResult?.mostProbableActivity  
                //do something  
            } } }  
}
```

RETRIEVE RECOGNITIONS II

```
//finally, register the broadcast receiver
val intentFilter = IntentFilter("RECOGNITION_ACTION")
broadcastReceiver = MyBroadcastReceiver(logger)
context.registerReceiver(broadcastReceiver, intentFilter)
```

OUTLOOK

Jetpack Compose is becoming more and more popular for creating native Android Apps.

Check out the official GoogleIO talks on Jetpack Compose.

Mobile Computing Bluetooth Low Energy on Android Smartphones

CC BY-SA, T. Amberg, FHNW
Slides: tmb.gr/mc-cen



[Source on GDocs](#)

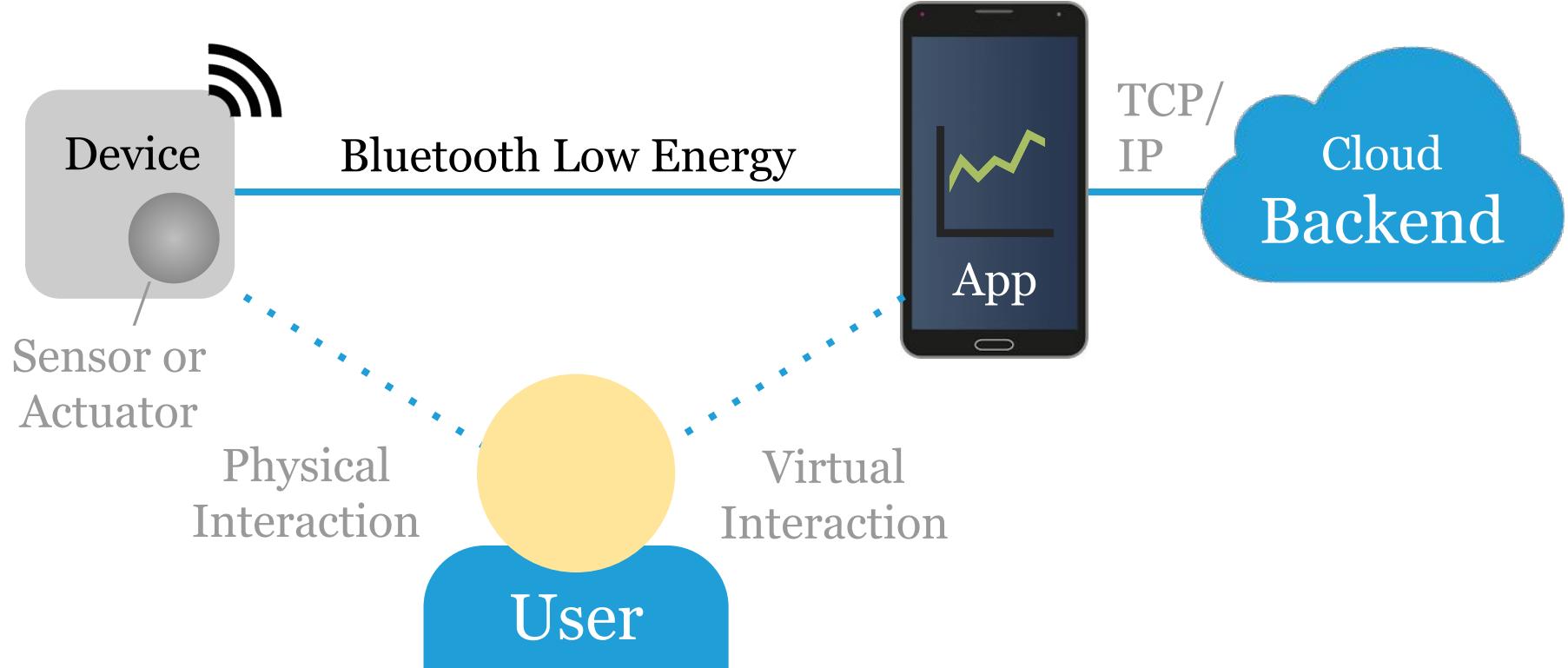
Overview

These slides introduce *BLE on Android smartphones*.

How to implement the central role, as a GATT client.

Scanning, reading, writing and getting notifications.

Reference model



BLE on Android

Android has built-in [BLE support*](#) since [API level 18](#).

A smartphone can take the central or peripheral role.

The Android SDKs BLE API is rather tricky to use.

Luckily there are some good 3rd party libraries.

*Also Bluetooth Classic for Headset, A2DP, HDP.

Adding permissions

Add these **permissions** to your app manifest file:

`android.permission.BLUETOOTH`

18+

`android.permission.BLUETOOTH_ADMIN`

18+

Also, as BLE discovery is privacy critical*, add:

~~`android.permission.ACCESS_COARSE_LOCATION`~~

23+

`android.permission.ACCESS_FINE_LOCATION`

29+

*A scan reveals nearby devices/beacons, often such a "fingerprint" is all it takes to derive a user's location. 5

Checking hardware capability

Indicate usage of BLE in your app manifest file:

```
<uses-feature  
    android:name="android.hardware.bluetooth_le"  
    android:required="true" />
```

And test if BLE is available on the current system:

```
if (getPackageManager().hasSystemFeature(  
    PackageManager.FEATURE_BLUETOOTH_LE)) { ... }
```

Setting up a BluetoothAdapter

Get the [BluetoothAdapter](#) via [BluetoothManager](#):

```
final BluetoothManager bluetoothManager =  
    (BluetoothManager) getSystemService(  
        Context.BLUETOOTH_SERVICE);  
BluetoothAdapter bluetoothAdapter =  
    bluetoothManager.getAdapter();
```

Ensuring Bluetooth is enabled

If Bluetooth is not enabled, ask the user to enable it:

```
if (bluetoothAdapter != null &&
    bluetoothAdapter.isEnabled()) { ...
} else {
    Intent enableBtIntent = new Intent(
        BluetoothAdapter.ACTION_REQUEST_ENABLE);
    startActivityForResult(
        enableBtIntent, REQUEST_ENABLE_BT);
}
```

Getting a scanner

Get a **BluetoothLeScanner** to scan for BLE devices:

```
BluetoothLeScanner scanner =  
    bluetoothAdapter.getBluetoothLeScanner();
```

The **Handler** runs a *Runnable** on another thread:

```
static final long SCAN_PERIOD_MS = 10000;  
Handler handler =  
    new Handler(Looper.getMainLooper());
```

*See slide *Starting a Scan*, p.11

Asking for location permission (again)

```
String[] permissions = new String[]{
    Manifest.permission.ACCESS_FINE_LOCATION
};
ActivityCompat.requestPermissions(
    MainActivity.this, permissions, 0);
```

```
@Override
public void onRequestPermissionsResult (...) {
    scan();
}
```

Starting a scan

```
void scan() {  
    handler.postDelayed(new Runnable() {  
        @Override  
        public void run() {  
            scanner.stopScan(scanCallback);  
        }  
    }, SCAN_PERIOD_MS);  
    scanner.startScan(scanCallback);  
}
```

Getting scan results

```
ScanCallback scanCallback = new ScanCallback() {  
    @Override  
    public void onScanResult(  
        int callbackType, ScanResult result) {  
        super.onScanResult(callbackType, result);  
        Log.d(TAG, result.getDevice());  
    }  
};
```

Hands-on, 10': Android BLE scanner

Build and run the [MyBleScannerApp](#) example code.

Check the Logcat output with filter "package:mine".

If it works, you should see BLE devices around you.

Test the app by disabling Bluetooth, location, etc.

Done? Add a button to the UI to start a scan.

Heart rate service

This service is intended for fitness heart rate sensors:

Heart Rate Service UUID (16-bit): **0x180D**

This service includes the following characteristics:

Heart Rate Measurement UUID: **0x2A37** [N]

Body Sensor Location UUID: **0x2A38** [R]

Heart Rate Control Point UUID: **0x2A39** [W]*

Standard service, defined by the Bluetooth SIG.

Base UUID for registered services

For a custom service UUID we define all 128-bits, e.g.

0x6E400001-B5A3-F393-E0A9-E50E24DCCA9E

For well known, **registered services** the *base UUID* is*

0x00000000-0000-1000-8000-00805F9B34FB

A 16-bit service "UUID", e.g. **0x180D** corresponds to:

0x000180D-0000-1000-8000-00805F9B34FB

*See **Bluetooth spec v5.3**, p.1181 ff.

Connecting to discover GATT services

```
final BluetoothGattCallback gattCallback =  
    new BluetoothGattCallback() {  
        @Override  
        public void onConnectionStateChange(...) { ... }  
        gatt.discoverServices();  
    } ...  
};  
BluetoothGatt gatt = device.connectGatt(  
    this, /* reconnect */ true, gattCallback);
```

Reading a GATT characteristic

```
final BluetoothGattCallback gattCallback =  
    new BluetoothGattCallback() { ...  
        @Override  
        public void onServicesDiscovered(...) { ...  
            BluetoothGattCharacteristic characteristic  
                = gattService.getCharacteristic(uuid);  
            gatt.readCharacteristic(characteristic);  
        } ...  
    };
```

Getting a GATT characteristic value

```
final BluetoothGattCallback gattCallback =  
    new BluetoothGattCallback() { ...  
        @Override  
        public void onCharacteristicRead(...) { ...  
            if (characteristic.getUuid().equals(...)) {  
                byte[] value = characteristic.getValue();  
            }  
        } ...  
    };
```

Writing a GATT characteristic

```
final BluetoothGattCallback gattCallback =  
    new BluetoothGattCallback() { ...  
        @Override  
        public void onServicesDiscovered(...) { ...  
            BluetoothGattCharacteristic characteristic  
                = gattService.getCharacteristic(uuid);  
            characteristic.setValue(value, format, 0);  
            gatt.writeCharacteristic(characteristic);  
        } ... };
```

Getting GATT characteristic write status

```
final BluetoothGattCallback gattCallback =  
    new BluetoothGattCallback() { ...  
        @Override  
        public void onCharacteristicWrite(...) { ...  
            if (characteristic.getUuid().equals(...)) {  
                Log.d(TAG, "write, status = " + status);  
            }  
        } ...  
    };
```

Enabling GATT char. notifications

```
@Override  
public void onServicesDiscovered(...) { ...  
    gatt.setCharacteristicNotification(..., true);  
    UUID configUuid = UUID.fromString("2902");  
    BluetoothGattDescriptor descriptor =  
        characteristic.getDescriptor(configUuid);  
    descriptor.setValue(BluetoothGattDescriptor.  
        ENABLE_NOTIFICATION_VALUE);  
    gatt.writeDescriptor(descriptor);  
}
```

Getting GATT characteristic notifications

```
final BluetoothGattCallback gattCallback =  
    new BluetoothGattCallback() { ...  
        @Override  
        public void onCharacteristicChanged(...) { ...  
            if (characteristic.getUuid().equals(...)) {  
                byte[] value = characteristic.getValue();  
            }  
        } ...  
    };
```

Hands-on, 10': Android BLE central

Build and run the [MyBleCentralApp](#) example code.

Use the nRF52840 with [HRM BLE peripheral code](#).

Check the Logcat output with filter "package:mine".

If it works, you should see heart rate measurements.

Done? Read the Android app source code in detail.

What's missing in the example code

BLE related code should be moved into a [Service](#).

Starting a scan should be triggered by the user*.

Discovered devices should be presented in a list.

Read, write, etc. operations should be queued.

*On API level 26+, try [companion device pairing](#).

Hands-on, 10': Android BLE issues

It looks easy, but BLE on Android has many issues.

Read some of the tips how to [make it actually work](#).

Also check this [list of issues](#) to get an impression.

iOS BLE seems more stable. Why could that be?

Done? Find some stats about the Android OS.

BLE libraries from 3rd parties

Writing robust apps based on the BLE API is hard.

There are a number of 3rd party libraries to fix this:

The [Nordic Android BLE library](#) is written in Java.

Nordic also has a [Kotlin BLE library](#) for Android*.

[RxAndroidBle library](#) uses the [RxJava](#) framework.

*Uses Kotlin coroutines.

Hands-on, 10': nRF Blinky app example

[Android nRF Blinky](#) is a complete BLE app example.

It shows how to use the [Nordic Android BLE library](#).

Read the library docs and study the app source code.

Which parts become easier by using this library?

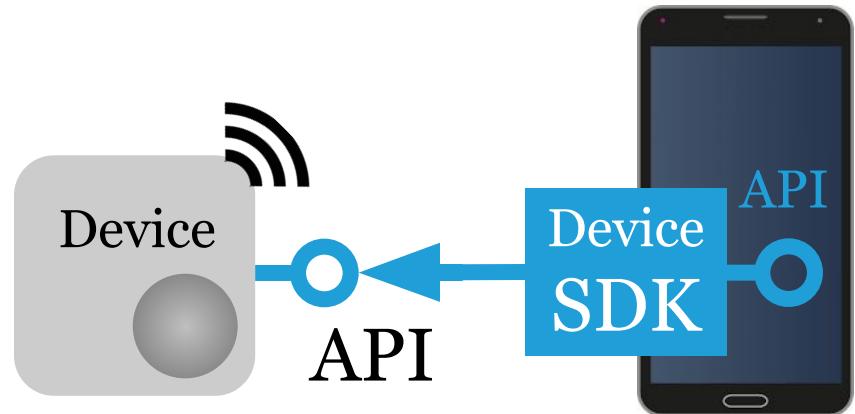
Done? Sketch the BLE API of the peripheral.

Device API vs. SDK

A *device API* specifies how to talk to the device, from any client (here via BLE).

A platform specific *device SDK* simplifies integration.

E.g. *iOS device SDK* to talk to a device API from iOS.



Battery Service
Battery Level [R]

vs.

```
p = ble.conn(addr);  
b = sdk.getBatt(p);  
x = b.getLevel();
```

Summary

The Android BLE API allows to implement a central.

Scan, read, write and notify operations use callbacks.

Service and characteristic UUID define the interface.

3rd party libraries can help to write robust BLE code.

Challenge: nRF Toolbox plugin

nRF Toolbox is a "container app" for BLE demos.

Build and run the [nRF Toolbox app source code](#).

Write a plugin for a [custom SHT30 BLE service](#).

Test the app with a [nRF52840 BLE peripheral](#).

Done? Create a custom icon for your service.

Feedback or questions?

Write me on Teams or email

thomas.amberg@fhnw.ch

Thanks for your time.

Mobile Computing Bluetooth Low Energy on Microcontrollers

CC BY-SA, T. Amberg, FHNW
Slides: tmb.gr/mc-per



[Source on GDocs](#)

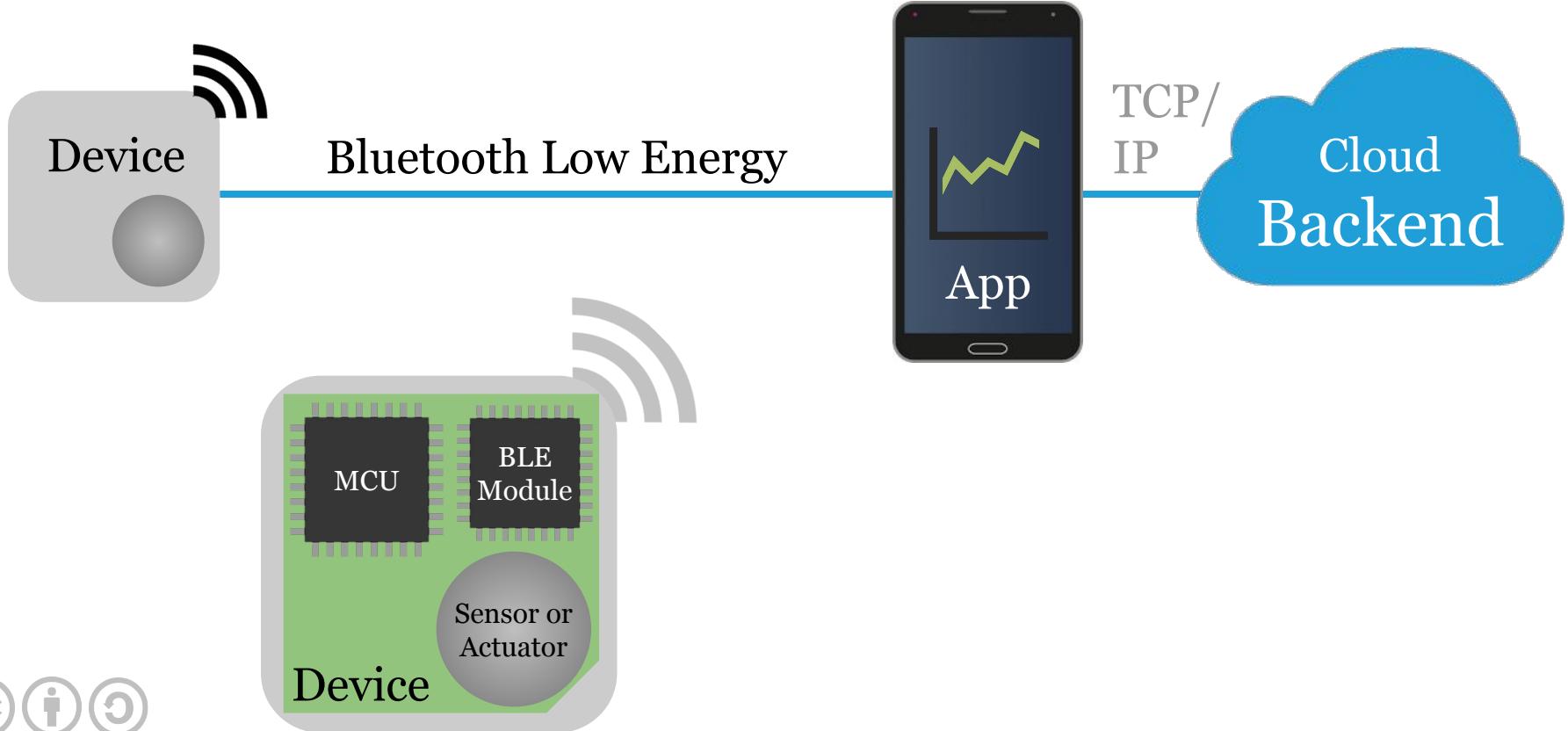
Overview

These slides introduce *BLE on microcontrollers*.

Examples for the peripheral and central roles.

Designing BLE services and characteristics.

Reference model



Prerequisites

Install the Arduino IDE and set up the nRF52840:

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

[Set up the Feather nRF52840 Sense with Arduino](#).

Setting up the board also installs this [BLE library](#).

For testing, a smartphone with BLE is required.

BLE on the nRF52840

The nRF52840 can take the peripheral or central role.

The [Adafruit BLE library source code](#) and [examples](#) provide some "documentation" and a starting point.

To implement or use a peripheral, we have to know its API, consisting of service and characteristic UUIDs.

Heart rate service

This service is intended for fitness heart rate sensors:

Heart Rate Service UUID (16-bit): **0x180D**

This service includes the following characteristics:

Heart Rate Measurement UUID: **0x2A37** [N]

Body Sensor Location UUID: **0x2A38** [R]

Heart Rate Control Point UUID: **0x2A39** [W]*

Standard service, defined by the Bluetooth SIG.

nRF52840 HRM BLE peripheral .ino

```
hrmSvc = BLEService(0x180D); // See HRM spec
hrmChr = BLECharacteristic(0x2A37); // See spec

hrmSvc.begin(); // to add characteristics
hrmChr.setProperties(CHR_PROPS_NOTIFY); ...
hrmChr.begin(); // adds characteristic

uint8_t hrmData[2] = { 0b00000110, value };
hrmChr.notify(hrmData, sizeof(hrmData));
```

Hands-on, 10': HRM BLE peripheral

Build and run the previous nRF52840 BLE example.

Use the *.ino* link on the page to get the example code.

Explore the HRM example using a smartphone app*.

Try to enable notifications to get value updates.

*Try nRF Connect for Android or iOS.

nRF52840 HRM BLE central .ino

```
BLEClientService hrmSvc(UUID16_SVC_HEART_RATE);
BLEClientCharacteristic hrmChr(UUID16_CHR_...);

// part of setup()
Bluefruit.begin(0, 1); // 1 central connection
hrmSvc.begin();
hrmChr.setNotifyCallback(notifyCbck);
hrmChr.begin(); // implicitly added to service
Bluefruit.Central.setConnectCallback(connCbck);
```

nRF52840 HRM BLE central (ff.) .ino

```
void connCbck(uint16_t connHandle) {  
    if (hrmSvc.discover(connHandle)) {  
        if (hrmChr.discover()) {  
            hrmChr.enableNotify();  
        } else { ... }  
    } else {  
        Bluefruit.disconnect(connHandle);  
    }  
}
```

nRF52840 HRM BLE central (ff.) .ino

```
Bluefruit.Scanner.setRxCallback(scanCbck);
Bluefruit.Scanner.filterUuid(hrmSvc.uuid);
Bluefruit.Scanner.restartOnDisconnect(true);
Bluefruit.Scanner.start(0); // non-stop

void scanCbck(ble_gap_..._report_t* report) {
    // optional: check for device address
    Bluefruit.Central.connect(report);
}
```

Hands-on, 10': HRM BLE central

Build and run the previous nRF52840 BLE example.

Use the *.ino* link on the page to get the example code.

Open the Arduino serial monitor to enter a message.

Use a second nRF52840* as a HRM peripheral.

*Or use your smartphone as a **peripheral simulator**.

Nordic UART service

This service provides a serial connection over BLE:

[Nordic UART Service](#) custom (128-bit) UUID:
0x6E400001-B5A3-F393-E0A9-E50E24DCCA9E

This service includes the following characteristics:

RX (device receives data) UUID: **0x0002 [W]**
TX (device transmits data) UUID: **0x0003 [N]**

This service is becoming a *de facto* standard.

nRF52840 UART BLE peripheral .ino

```
// UUID: 6E400001-B5A3-F393-E0A9-E50E24DCCA9E
uint8_t const uartSvcUuid[ ] = { 0x9E, 0xCA, ...,
0xB5, 0x01, 0x00, 0x40, 0x6E }; // lsb first

uartSvc = BLEService(uartSvcUuid); // 128-bit
rxChr = BLECharacteristic(rxChrUuid); // 128-b.
txChr = BLECharacteristic(txChrUuid); // 128-b.

txChar.setProperties(CHR_PROPS_NOTIFY);
rxChar.setProperties(CHR_PROPS_WRITE);
```

Hands-on, 10': UART BLE peripheral

Build and run the previous nRF52840 BLE example.

Use the *.ino* link on the page to get the example code.

Write bytes to *RX* with a generic BLE explorer app.

Check the serial monitor to see the received bytes*.

*Why do some bytes not show up?

nRF52840 UART BLE central .ino

```
Bluefruit.begin(0, 1); // 1 central connection
uartSvcClient.begin();
uartSvcClient.setRxCallback(rxCbck); // read
Bluefruit.Central.setConnectCallback(connCbck);

void connCbck(uint16_t connHandle) {
    if (uartSvcClient.discover(connHandle)) {
        uartSvcClient.enableTXD(); // enable notify
        uartServiceClient.print(...); // write data
    } }
```

nRF52840 UART BLE central (ff.) .ino

```
Bluefruit.Scanner.setRxCallback(found);  
  
void found(ble_gap_evt_adv_report_t* report) {  
    if (...Scanner.checkReportForService(  
        report, uartServiceClient)) {  
        Bluefruit.Central.connect(report);  
    } else {  
        Bluefruit.Scanner.resume();  
    }  
}
```

Hands-on, 10': UART BLE central

Build and run the previous nRF52840 BLE example.

Use the *.ino* link on the page to get the example code.

Open the Arduino serial monitor to enter a message.

Use a second nRF52840 as a UART peripheral.

nRF52840 beacon BLE observable .ino

```
BLEBeacon beacon(  
    beaconUuid, // AirLocate UUID  
    beaconMajorVersion,  
    beaconMinorVersion,  
    rssiAtOneMeter);  
  
beacon.setManufacturer(0x004C); // Apple  
startAdvertising();  
  
suspendLoop(); // save power
```

nRF52840 scanner BLE central .ino

```
Bluefruit.begin(0, 1); // Central
Bluefruit.Scanner.setRxCallback(found);
Bluefruit.Scanner.start(0);

void found(ble_gap_evt_adv_report_t* report) {
    Serial.printBufferReverse( // little endian
        report->peer_addr.addr, 6, ':' );
    if (Bluefruit.Scanner.checkReportForUuid(...)) ...
    Bluefruit.Scanner.resume();
}
```

Hands-on, 10': Scanner BLE central

Build and run the previous nRF52840 BLE examples.

Use the *.ino* link on the page to get the example code.

Test the scanner with a (simulated) HRM peripheral.

Adapt the scanner to scan for the beacon observable.

Bonus: Scan for Covid-19 apps as described [here](#).

Summary

The nRF52840 can take the peripheral or central role.

To build/use a service we need its 16-/128-bit UUID.

Peripherals set up services, update characteristics.

Centrals connect to read, write or get notifications.

The specific value format depends on the service.

Challenge

Design and implement an API for the [SHT30 sensor](#).

Create UUIDs for your service and its characteristics.

Chose a data format that fits the sensor value range.

Consider to allow reading, writing or notifications.

Test your peripheral with a generic BLE explorer.

Feedback or questions?

Write me on Teams or email

thomas.amberg@fhnw.ch

Thanks for your time.

Mobile Computing Bluetooth Low Energy Personal Area Networks

CC BY-SA, T. Amberg, FHNW
Slides: tmb.gr/mc-ble



[Source on GDocs](#)

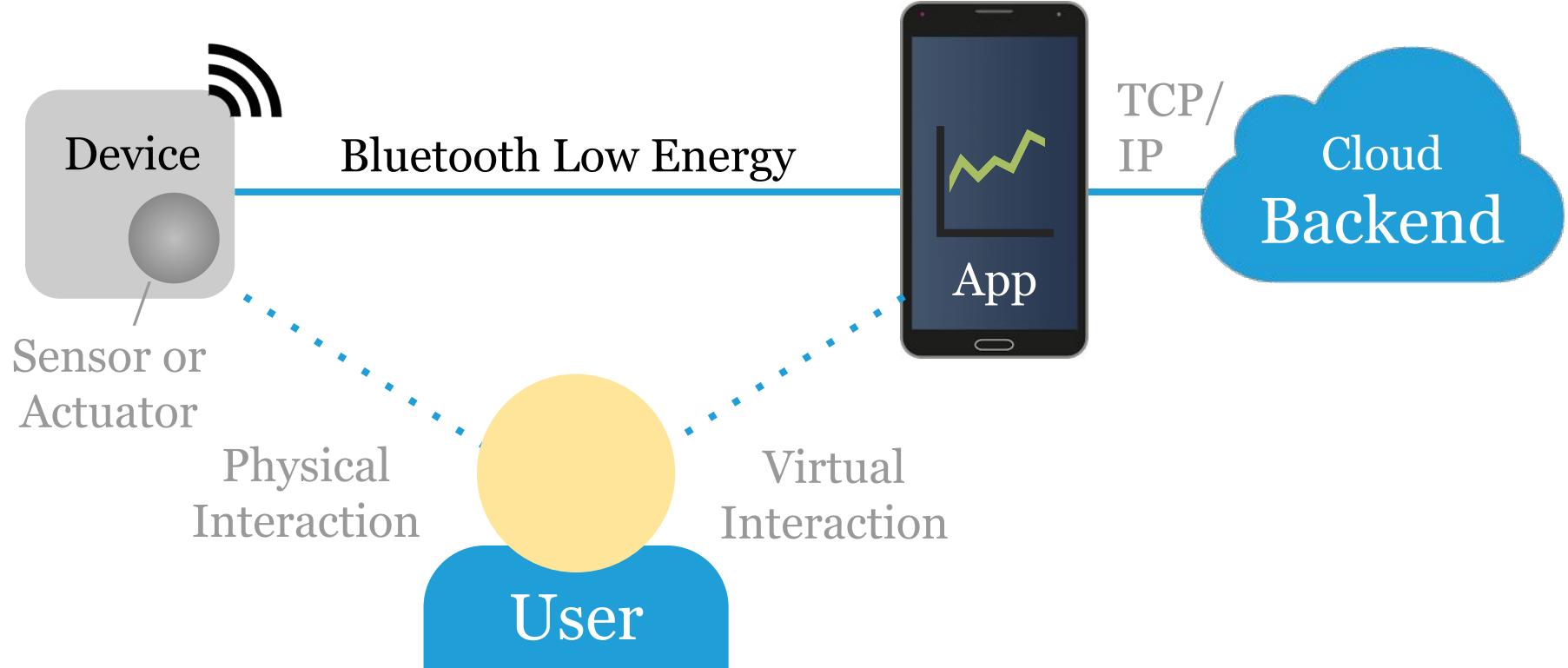
Overview

These slides introduce *Bluetooth Low Energy*.

Which are the roles of involved parties.

How a BLE service is structured.

Reference model



Bluetooth Low Energy (BLE)

BLE is a power-efficient Bluetooth variant (since 4.0).

BLE is well suited for small, battery powered devices.

It uses less energy than Wi-Fi and way less than 4/5G.

Range is ~30 m, data rate 1 Mbps, frequency 2.4 GHz.

The standard is maintained by the Bluetooth SIG.

How BLE works

Peripherals advertise the data they have, over the air.

Centrals scan for nearby peripherals to discover them.

The central connects to a peripheral and uses its data.

Data is structured into *services* and *characteristics*.

BLE protocol stack

Application – application specific code and formats

BLE library – thin, language-specific wrapper library

GATT – services & characteristics | GAP – discovery

ATT – attribute transport | SMP – security manager

L2CAP – logical link control and adaptation protocol

Link layer – exposed via the host controller interface

Physical layer – dealing with actual radio signals

Generic Access Profile (GAP)

GAP defines the following roles, communication types:

Broadcaster and *observer* (connectionless, one-way).

Peripheral and *central* (bidirectional connection).

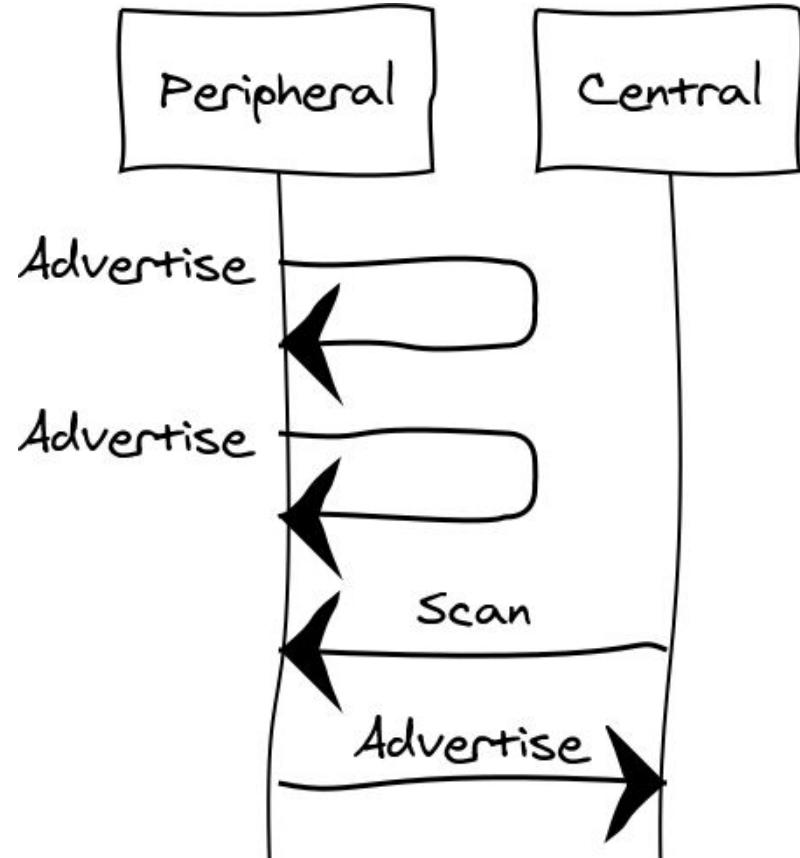
Each device supports one or more of these roles.

We start with peripheral and central roles.

Advertising

A peripheral *advertises* its services by broadcast, in a regular interval.

A central *scans* for all or a subset of services and gets device addresses and, if it's been sent, advertised data.



Attribute Transport (ATT)

ATT allows a *client* to access attributes on a *server*.

An *attribute* has a handle, a UUID and permissions.

An *attribute handle* is a server-assigned, 16-bit ID.

A *UUID* is a 16/128-bit universally unique identifier.

Permissions allow you to read, write or get notified.

See [Bluetooth spec v5.3](#), p.279 & [Assigned Numbers](#).

Generic Attribute Profile (GATT)

GATT is a simple application level protocol for BLE.

It's connection-based, with a *client* and a *server* role.

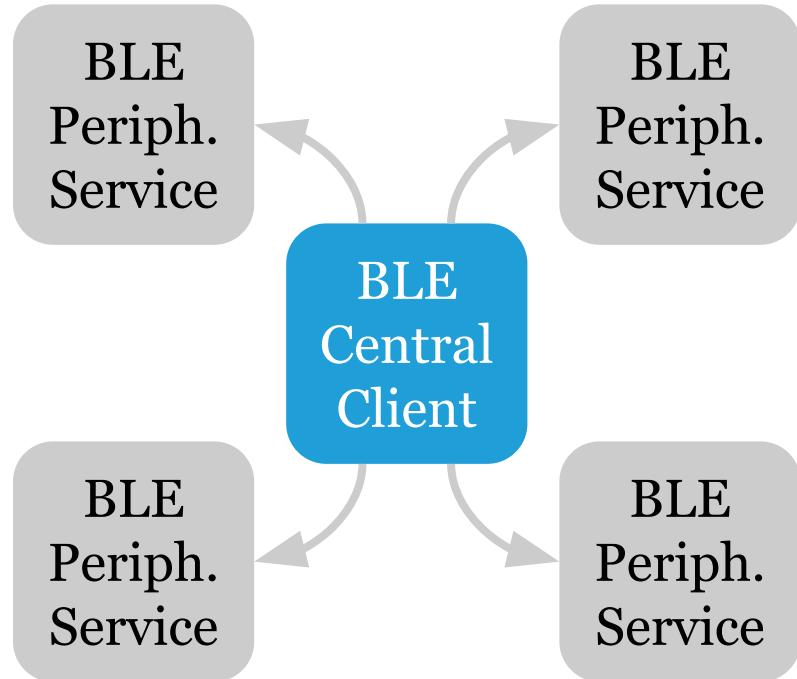
This enables a BLE device to provide a RESTful API.

A "GATT API", or *profile*, is a collection of *services*.

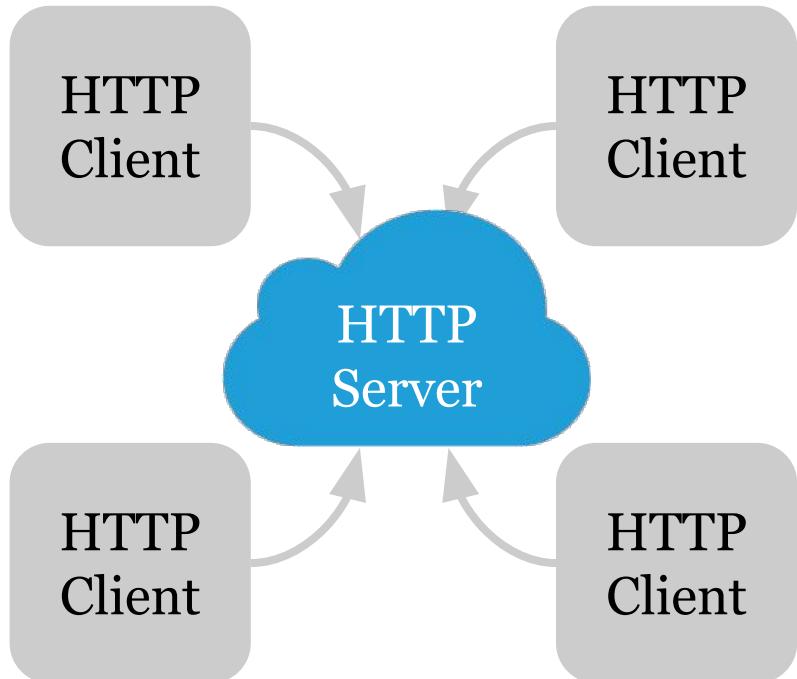
Usually, peripherals act as servers, central is client.

See [Bluetooth spec v5.3](#), p.280 & [List of Services](#).

BLE



HTTP



Services

A *GATT service* is a collection of characteristics.

Services encapsulate the behavior of part of a device.

In addition, such a service can refer to other services.

There are standard* and custom services and profiles.

*E.g. the **Battery Service** or the **Heart Rate Service**.

Characteristics

A *GATT characteristic** has a value and descriptors.

A *value* encodes data "bits" that form a logical unit.

Descriptors are defined attributes of a characteristic.

Supported procedures: read, write and notifications.

*E.g. *Battery Level* or *Heart Rate Measurement*.

Descriptors

A *GATT descriptor* describes a characteristic value.

E.g. *Presentation Format* or *Valid Range* descriptor.

Descriptors also allow to configure characteristics.

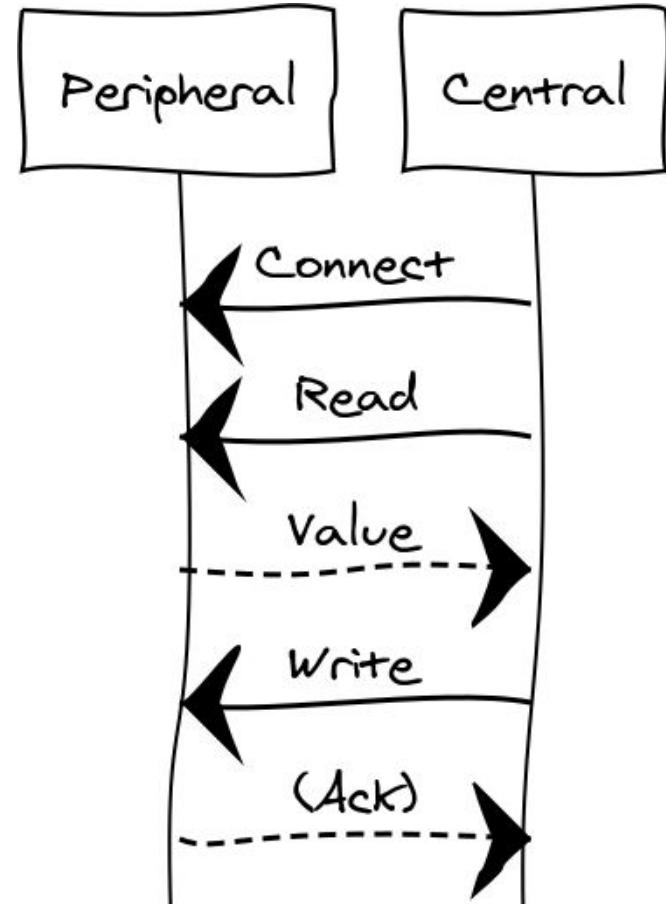
E.g. *Client Characteristic Configuration* descriptor allows a client to enable or disable notifications.

Read and write

Connect = the central connects to a peripherals BLE address.

Read = value of a characteristic or its descriptors is returned.

Write = characteristic value, or characteristic descriptor value is set, with/without response.

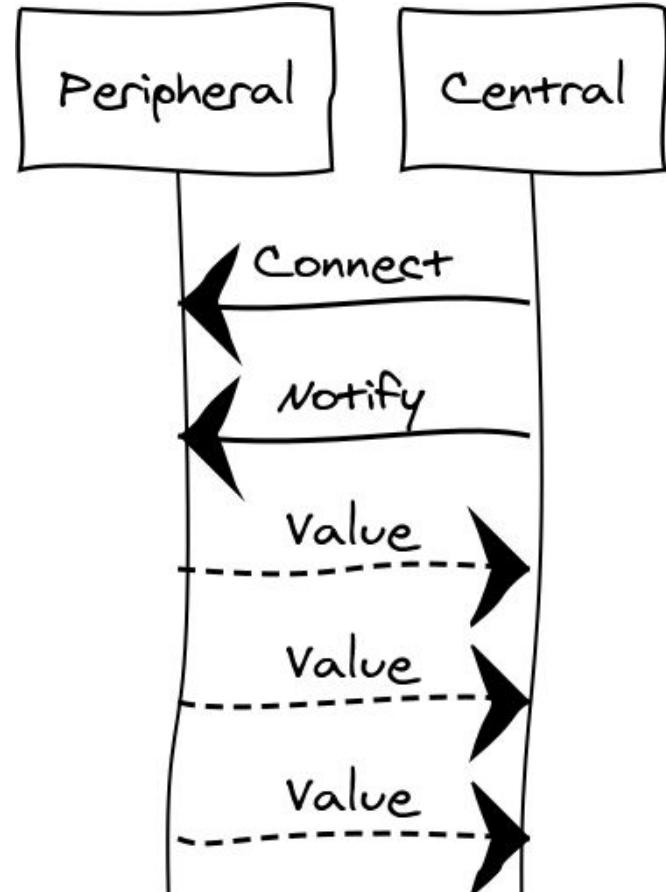


Notifications

Notify = *Client Characteristic Configuration* descriptor of a characteristic, UUID 0x2902, is set to 0x0001 using *Write*.

Value = A *Handle Value Notification* is sent if value changes.

See [Bluetooth spec v5.3](#), p.1489.



BLE explorer apps

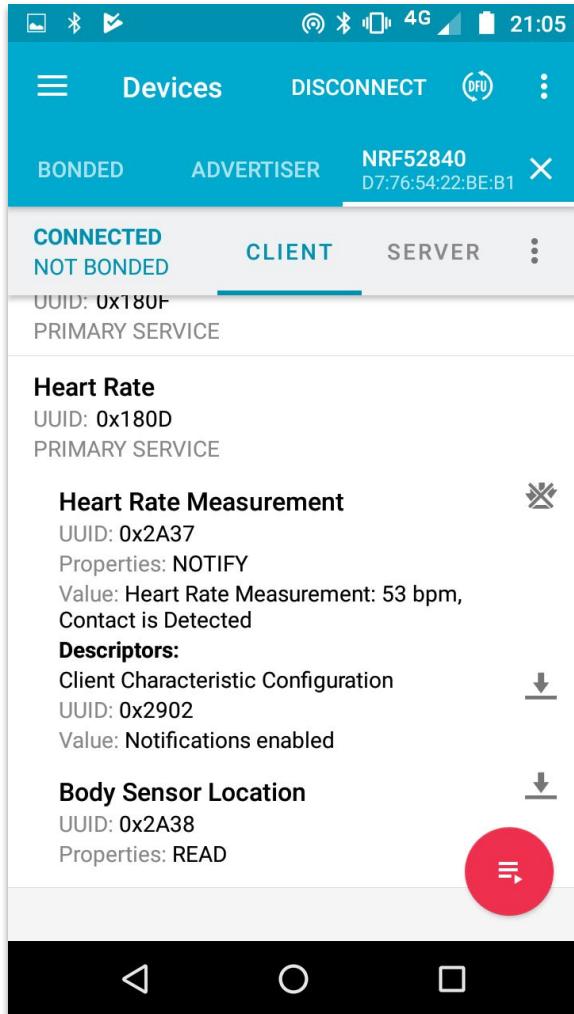
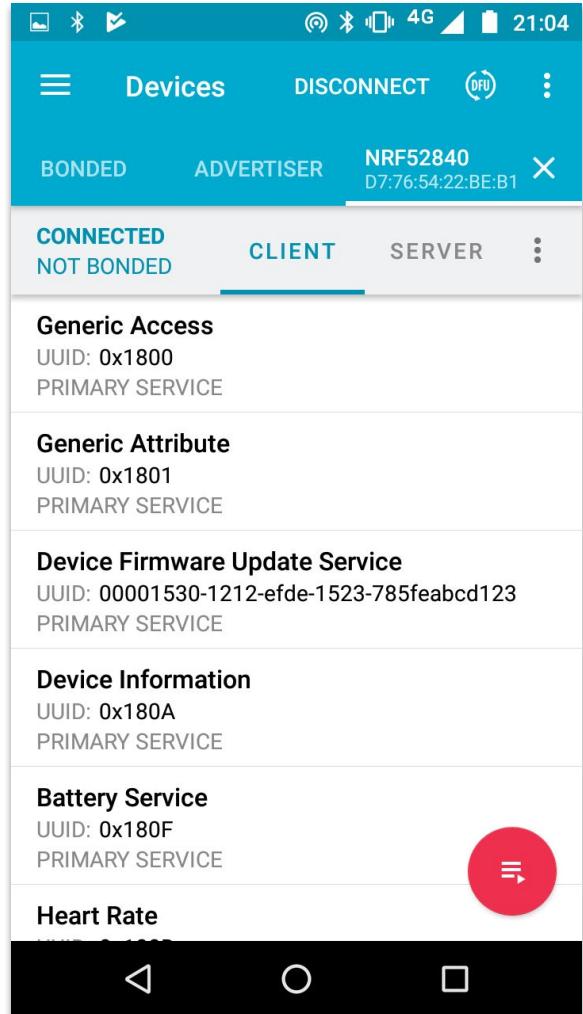
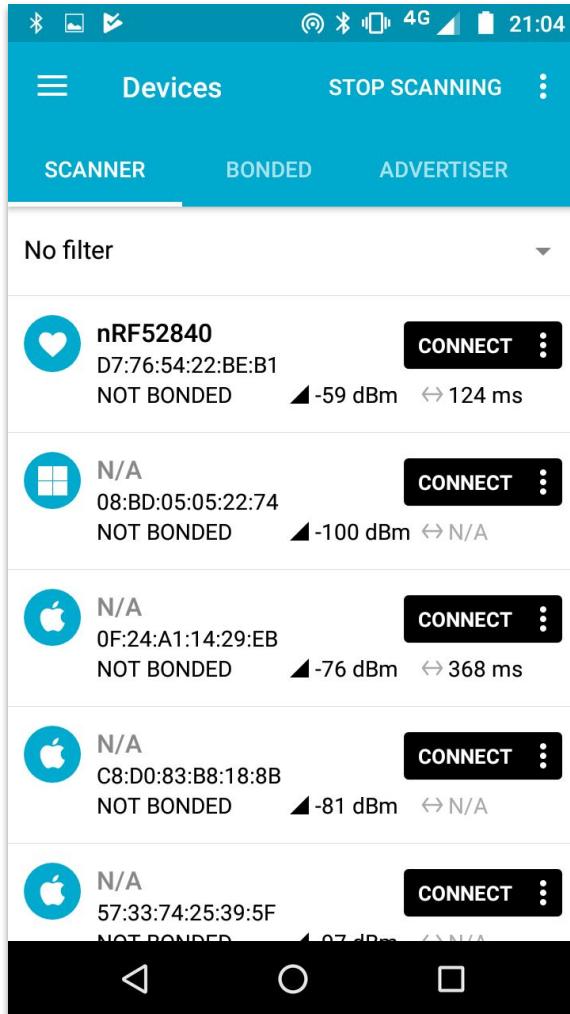
For debugging, use any generic BLE explorer app:

Find [BLE explorer apps](#) on the Google Play Store.

Search for "BLE explorer" in the iOS App Store.

Smartphones can act as central or peripheral.

Exploring is a great way to learn about BLE.



Heart rate service

This service is intended for fitness heart rate sensors:

Heart Rate Service UUID (16-bit): **0x180D**

This service includes the following characteristics:

Heart Rate Measurement UUID: **0x2A37** [N]

Body Sensor Location UUID: **0x2A38** [R]

Heart Rate Control Point UUID: **0x2A39** [W]*

Standard service, defined by the Bluetooth SIG.

Nordic UART service

This service provides a serial connection over BLE:

[Nordic UART Service](#) custom (128-bit) UUID:
0x6E400001-B5A3-F393-E0A9-E50E24DCCA9E

This service includes the following characteristics:

RX (device receives data) UUID: **0x0002 [W]**
TX (device transmits data) UUID: **0x0003 [N]**

This service is becoming a *de facto* standard.

Beacons

Beacons, e.g. [Apple iBeacon](#) are *broadcaster* devices.

Any *observer* can read the data which they advertise.

Lookup of "what a beacon means" requires an app.

Except for [Physical Web](#) / [Eddystone](#) beacons.

These contain an URL to be used right away.

Security

BLE has **security mechanisms** for pairing and more.

Pairing: exchanging identity and keys to set up trust.

Device chooses *Just Works*, *Passkey Entry* or **OOB**.

Or numeric comparison and **ECDH** for key exchange.

Some apps add encryption on the application layer.

Summary

BLE provides low power, personal area connectivity.

A BLE central scans for peripherals, who advertise.

Each BLE peripheral provides one or more services.

Services allow to read/write characteristic values.

Descriptors allow to configure notifications.

Feedback or questions?

Write me on Teams or email

thomas.amberg@fhnw.ch

Thanks for your time.

Mobile Computing Internet of Things & Connected Products

CC BY-SA, T. Amberg, FHNW

(Screenshots as "fair use")

Slides: tmb.gr/mc-iot



Source on GDocs

Overview

These slides introduce *IoT and connected products.*

Definitions, main use cases and how it works.

What becomes possible, good and bad.

Internet of Things (IoT)

Small, connected computers with sensors, actuators.

"Physical objects with a Web API." – [@hansamann](#)

IoT: "Global network of computers, sensors and actuators, connected through Internet protocols."

Web of Things: "RESTful Web services that measure or manipulate physical properties." – [@gsiot](#)

Ubiquitous computing

"The idea of integrating computers seamlessly into the world at large [...] *Ubiquitous computing*"

"How do technologies disappear into the background? The vanishing of electric motors may serve as an instructive example" e.g. in escalators or in a car.

- Mark Weiser in [The Computer for the 21st Century](#)

Connectivity

Transmit data at room, building or city scale.

Personal area network, e.g. BLE, Zigbee, ~30 m.

Local area networks, e.g. Ethernet, Wi-Fi, ~100 m.

Wide area networks, e.g. 5G, LoRaWAN, 2 to 20 km.

Use Cases

Efficiency, e.g. trash bins let you know they are full.

Convenience, e.g. remotely preheat a holiday home.

New insights, e.g. a crowdsourced air quality map.

Sectors include connected consumer products,
citizen sensing, industrial IoT and many more.

Connected products

Internet-connected consumer products.

[Philips Hue](#), connected lights with a Web API.

[Withings Scale](#), logs your weight to a dashboard.

[Good Night Lamp](#), linked lamps to share presence.



Smart lights Smarter controls

Philips Hue is not just a smart bulb, it's a smart lighting system. The smart lights, Hue Bridge, and smart controls will forever change the way you experience light.



Hue lights

These smart and energy-efficient LED lights come in a wide variety of shapes, sizes, and models to suit your space.



Hue Bridge

The heart of your Philips Hue system, the Bridge acts as a smart hub, connecting your devices to your smart lights. You can add up to 50 Philips Hue lights and accessories to one Bridge.



Hue app

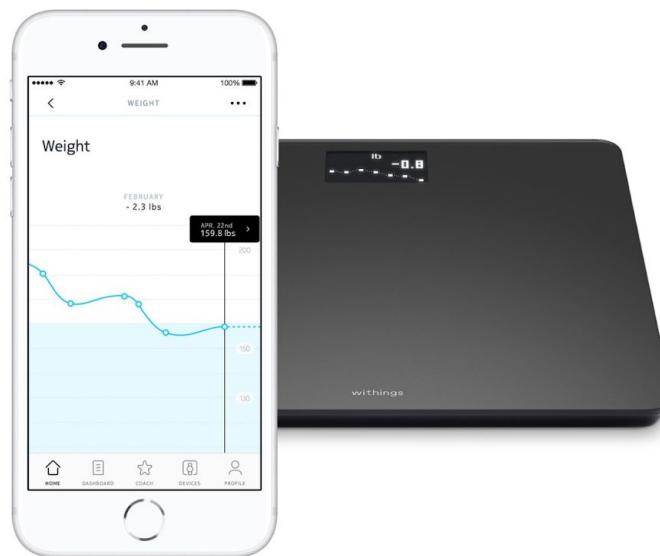
Control your smart lights quickly and conveniently with the Philips Hue app.

Weight & BMI Wi-Fi Scale - Body

https://www.withings.com/us/en/body

Meet your new accountability partner

Body offers a complete weight tracking experience tailored to individuals seeking easy, effective weight management. Weighing in is just the first step. Each session also provides instant feedback via weight trend and BMI screens, plus automatic sync to a free app on your smartphone, so you can track progress any time, anywhere.

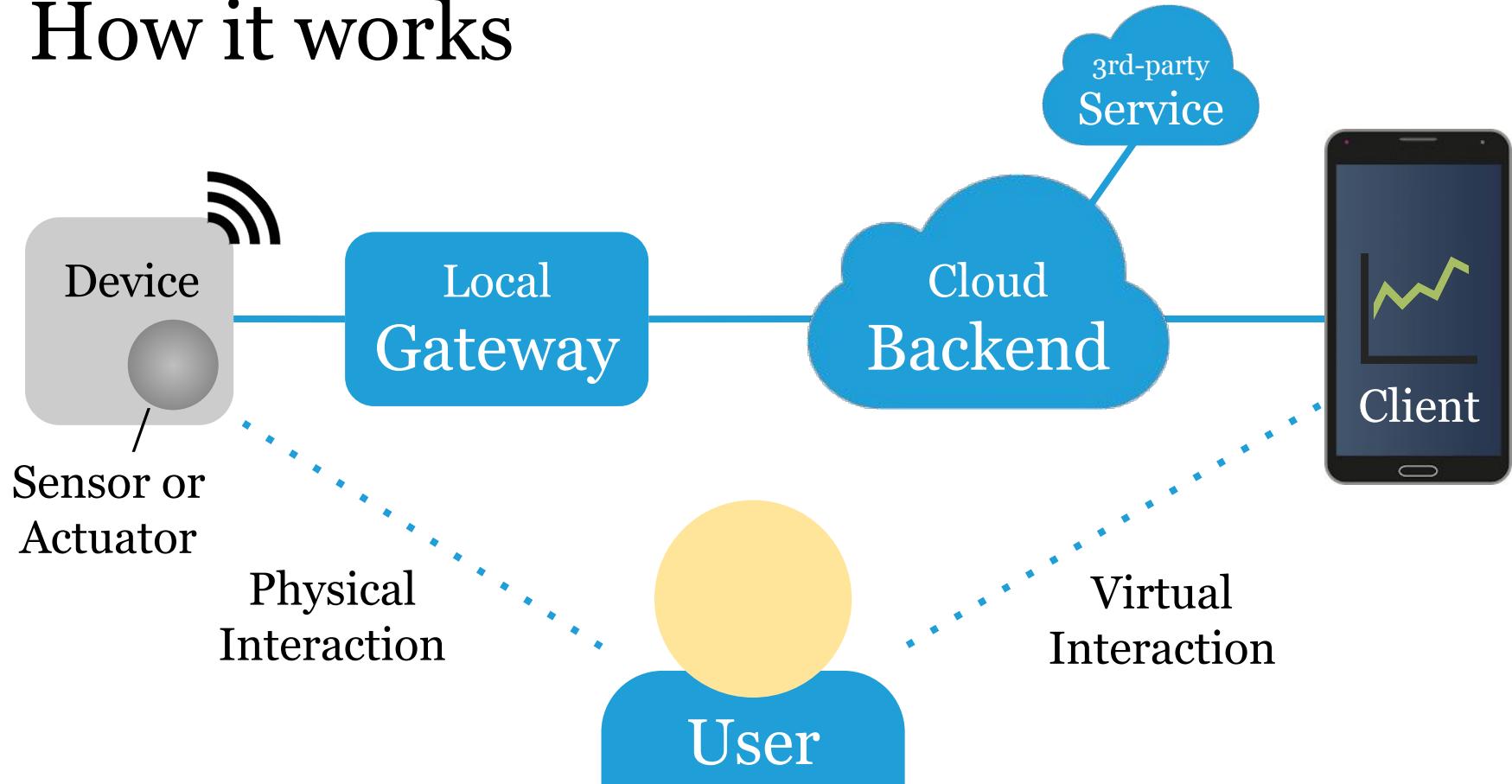


Turn a Big Lamp on and Little Lamps which you've given away turn on too. Anywhere in the world.

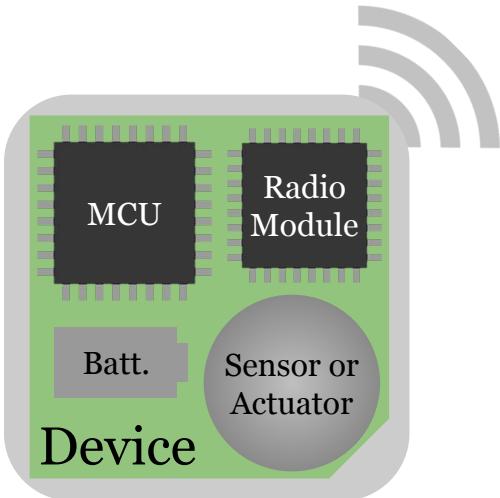
Use the Good Night Lamp to tell a loved one 'now's a good time for a chat', 'I'm thinking of you' or 'call me when you get home'. You decide. As your family grows or moves away, you can add as many Little Lamps as you want.



How it works



What's inside

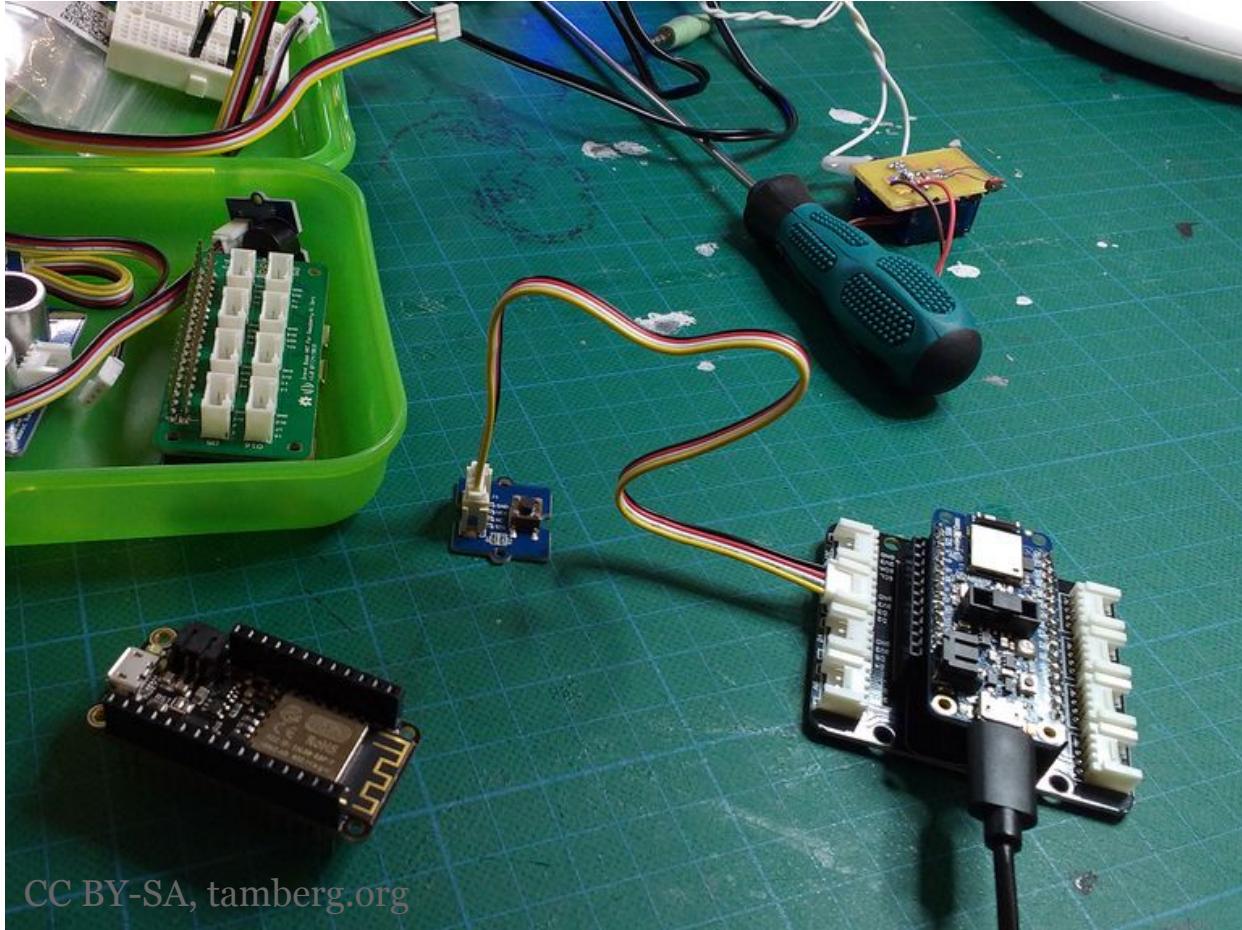


CC BY-SA, tamberg.org

Proto-typing

Use modular,
open source
hardware.

E.g. Feather
and Grove.



CC BY-SA, tamberg.org

From prototype to product

Hardware prototypes are easy, products are hard.

Engineering, to make electronics smaller, low power.

Design for manufacturing, to reduce cost per piece.

Scale up from 100 to 1000 to 10k per batch, learn.

From product to service

Each connected product represents a service.

Philips Hue lights for "smart lighting".

Kindle reader to "never be without a book".

Withings smart scale as "accountability partner".

mobility

Solutions

Vehicles & stations

How it works

Opening a new station



Good value

Unlike your own car, you only pay when you use our car sharing vehicles.



Transparent

Mobility is all inclusive: fuel, servicing and insurance.



Flexible

Rent for as little as one hour with round-the-clock self-service.



Convenient

Without a worry in the world: whether you are after somewhere to park, a



caru

Produkt
Über uns

Firmenkunden
Privatkunden

Login
Kontakt

D
E



IFTTT Guides For Hearing Aids | x +

oticon.com/support/how-to/ifttt/guides

Get a notification in your hearing aids when the doorbell rings.

[Click for full guide](#)



Receive a notification when your home alarm system goes off.

[Click for full guide](#)



Send a text message or email to a caregiver when the hearing aid battery is low.

[Click for full guide](#)



Have your hearing aids go into a special listening program when you make a voice command.

[Click for full guide](#)



Have your hearing aids switch to a preferred listening program based on your GPS location.

[Click for full guide](#)



[Click for full guide](#)

Better IoT

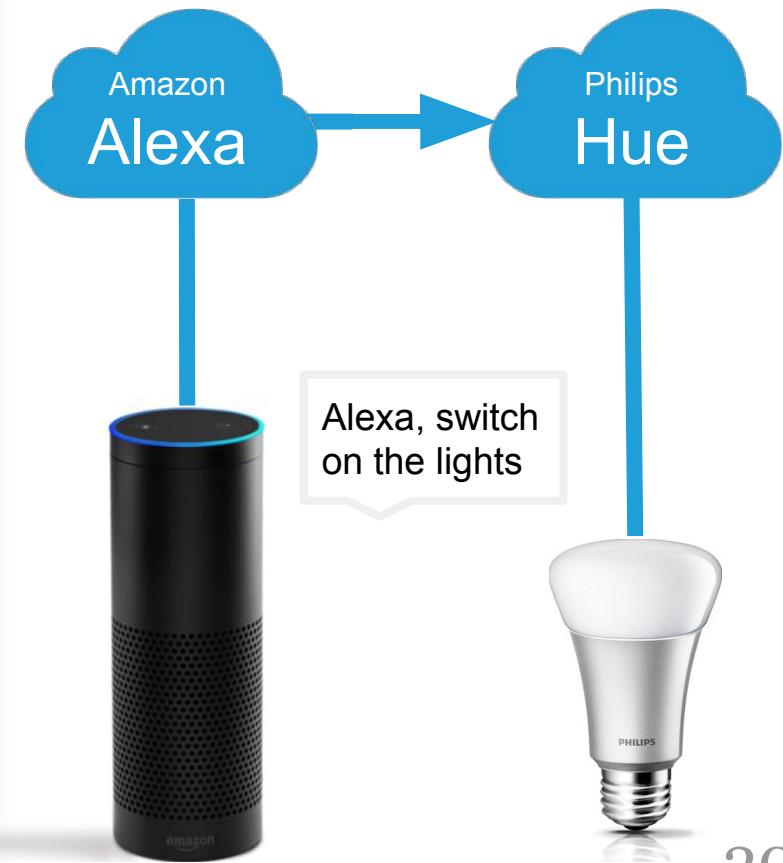
Security, to keep devices, network & backend secure.

Privacy, to keep people in control of their own data.

Interoperability, to become part of an ecosystem.

Openness, standards & open source build trust.

See, e.g. betteriot.org principles and [wiki](https://betteriot.org/wiki).





All Products

Protect Plans

Help

Log In

EN ▾

Cart (0)

New

Goes everywhere.
Sees everything.

Introducing the new Stick Up Cams.

Shop Now



Amazon's helping police build a surveillance network with Ring doorbells

Its popular Ring smart doorbells mean more cameras on more doorsteps, where surveillance footage used to be rare.

BY ALFRED NG | JUNE 5, 2019

ROBERT RODRIGUEZ

If you're walking in Bloomfield, New Jersey, there's a good chance you're being recorded. But it's not a corporate office or warehouse security camera capturing the footage -- it's likely a [Ring doorbell](#) made by [Amazon](#).

While residential neighborhoods aren't usually lined with [security cameras](#), the smart doorbell's popularity has essentially created private surveillance networks powered by Amazon and promoted by police departments.



OrCam MyEye.

Wearable technology for people who are blind or visually impaired.

Voice-activated device that provides increased independence by communicating visual information, audibly.

With OrCam MyEye, you can read text, recognize faces, identify products & more*.

*New: The Smart Reading feature is now available

Feedback or questions?

Write me on Teams or email

thomas.amberg@fhnw.ch

Thanks for your time.

Mobile Computing Microcontrollers, Sensors & Actuators

CC BY-SA 4.0, T. Amberg, FHNW
Slides: tmb.gr/mc-mcu



[Source on GDocs](#)

Overview

These slides introduce *microcontrollers*.

We learn how to run a program on one.

And how to use *sensors* and *actuators*.

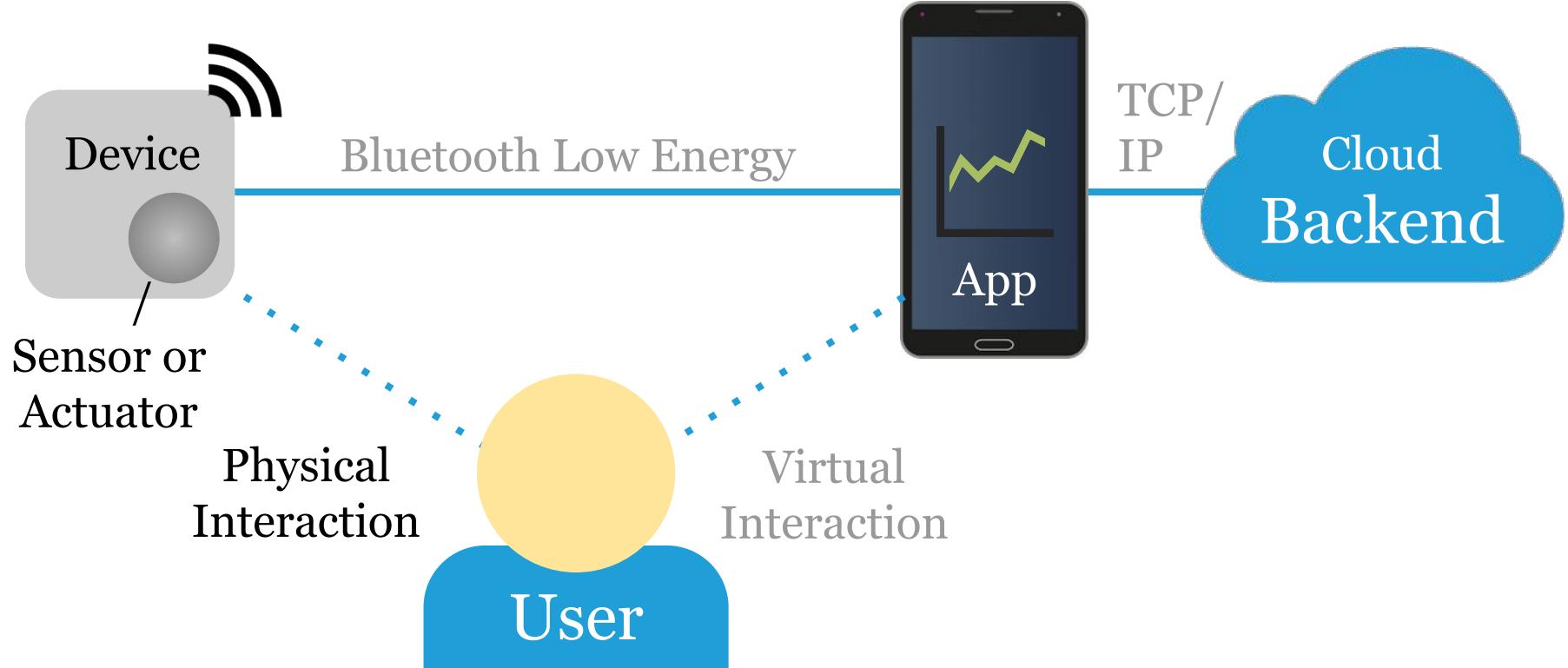
Prerequisites

Install the Arduino IDE and set up microcontrollers.

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

Set up the [Feather nRF52840 Sense](#) for Arduino.

Reference model



Let's look at physical computing

On device sensing/control, no connectivity.

Sensor → Device, e.g. logging temperature.

Device → Actuator, e.g. time-triggered buzzer.

Sensor → Device → Actuator, e.g. RFID door lock.

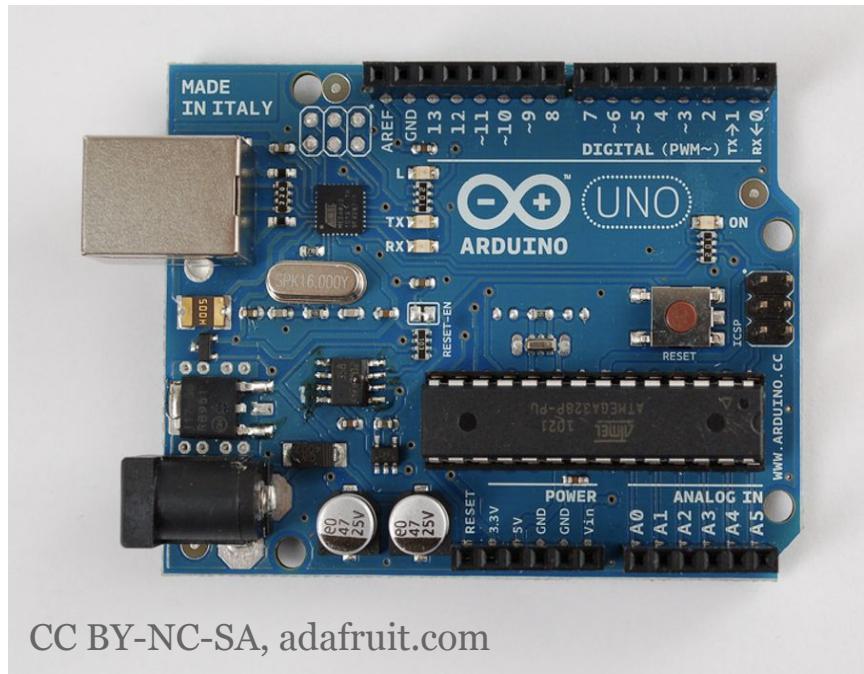
A → B: measurement or control data flow.

Arduino, a typical microcontroller

Microcontrollers (MCU)
are small computers that
run a single program.

Arduino is an MCU for
electronics prototyping.

Here's a [video](#) about it
with Massimo Banzi.

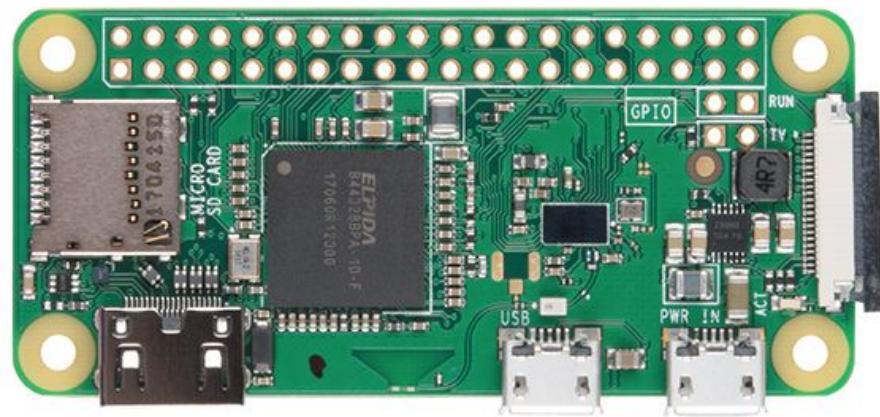


CC BY-NC-SA, adafruit.com

Raspberry Pi, a single-board computer

Single-board computers
like the *Raspberry Pi* are
not microcontrollers.

They run a full Linux OS,
have a lot of memory and
use way more power.



Here's a [video](#) on the Pi.

MCU vs. single-board computer

An MCU has limited memory and a slow processor.

But there's no operating system, i.e. less overhead.

This means an MCU can react faster, in real-time.

Use microcontrollers for simple, low latency tasks.

We'll use a microcontroller (and a smartphone).

Feather nRF52840 Sense

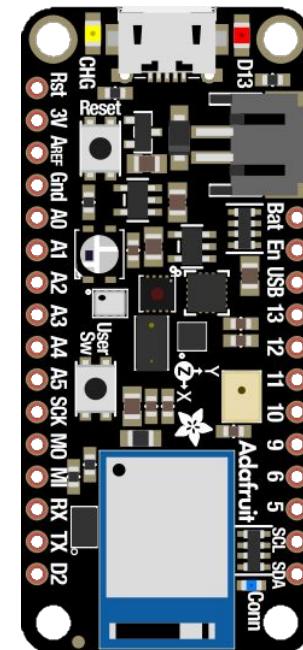
Microcontroller with **Bluetooth 5** (and more).

Nordic **nRF52840** System on Chip (SoC).

32-bit **ARM Cortex-M4** CPU with FPU.

1 MB **flash** memory, 265 kB RAM.

For details, check the **Wiki page**.



CC BY-SA, adafruit.com, fritzing.org

Programming a microcontroller

Microcontrollers are programmed via USB.

Code is (cross-) *compiled* on your computer.

The *binary* is *uploaded* to the microcontroller.

The uploaded program then runs "stand-alone".

Arduino IDE settings

Connect your board via USB and make sure that
Tools > Board is set to your microcontroller,
Tools > Port matches the current USB port.

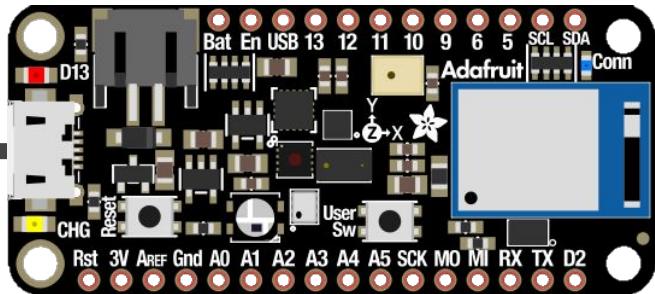
Some boards require additional settings.

Arduino IDE program upload

The *Upload* button compiles and uploads the code.



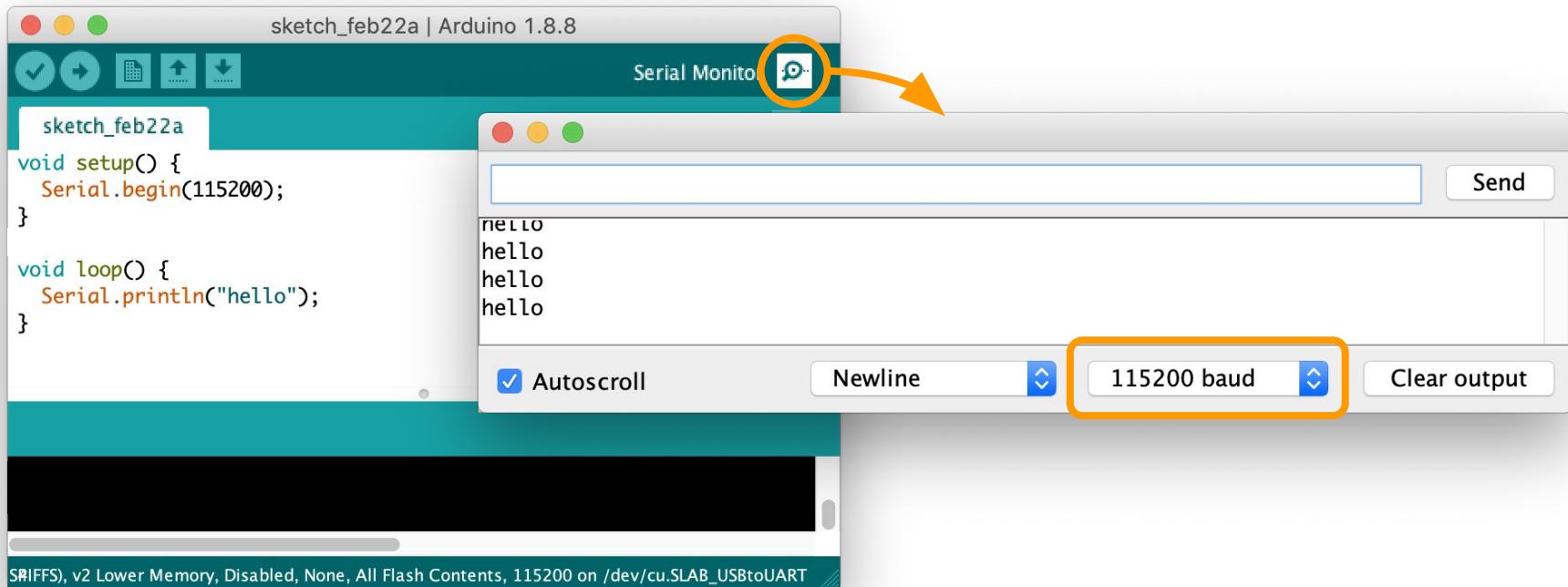
USB



CC BY-SA, adafruit.com, fritzing.org

Arduino IDE serial console

Make sure the baud rate matches *Serial.begin()*.



A typical program in Arduino C

```
void setup() { // called once at startup
    Serial.begin(115200); // set baud rate
}
```

```
void loop() { // called in a loop
    Serial.println("Hello, World!");
}
```

Arduino language

The Arduino language uses a subset of C/C++.

The user exposed code looks a bit like Java.

There is a `string` type and a `String` class.

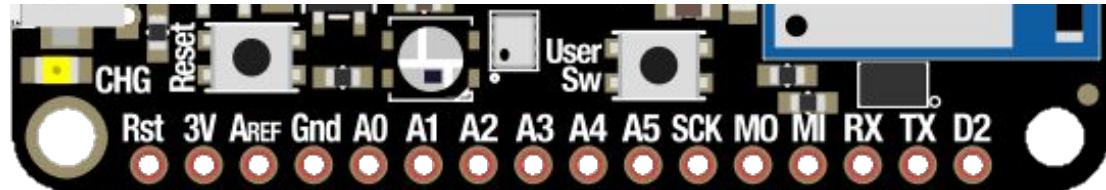
Libraries are programmed in C++.

For details, check the language reference.

General purpose input and output

Microcontrollers can "talk to" the physical world through general purpose input and output (GPIO).

GPIO *pins* allow a MCU to measure/control signals.



E.g. power, ground, analog pins, digital pin.

GPIO pin names

In Arduino, digital *pin names* are just numbers, e.g. pin 2, while analog pins start with an *A*, like pin *Ao*.

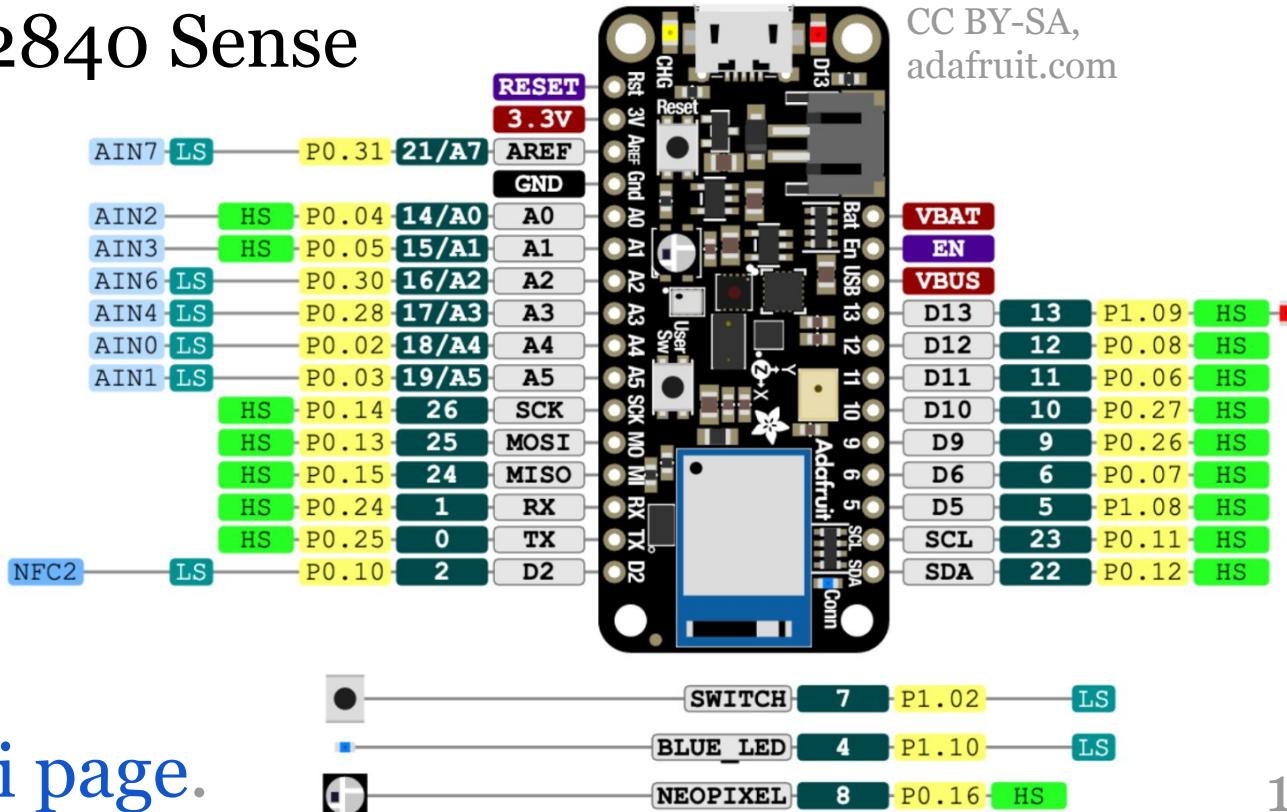
Which pins are available depends on the device.

The map of available pins is called *pinout*.

A pin can have multiple functions.

Pinout diagram

Feather nRF52840 Sense



More, see [Wiki page](#).

Sensors read the real world

Convert physical properties to electrical *input* signals.

E.g. temperature, humidity, brightness or orientation.

Input can be *digital* (0 or 1) or *analog* (e.g. 0 - 2^{10}).

Measuring = *reading* sensor values from input pins.

Actuators control the real world

Convert electrical *output* signals to physical properties.

E.g. light, current with a relay or motion with a motor.

Output can be *digital* (0 or 1) or *analog* (with PWM).

Controlling = *writing* actuator values to output pins.

Wiring sensors to the MCU

Sensors and actuators exchange signals with the MCU.

For prototyping, we use wires to achieve this, e.g.

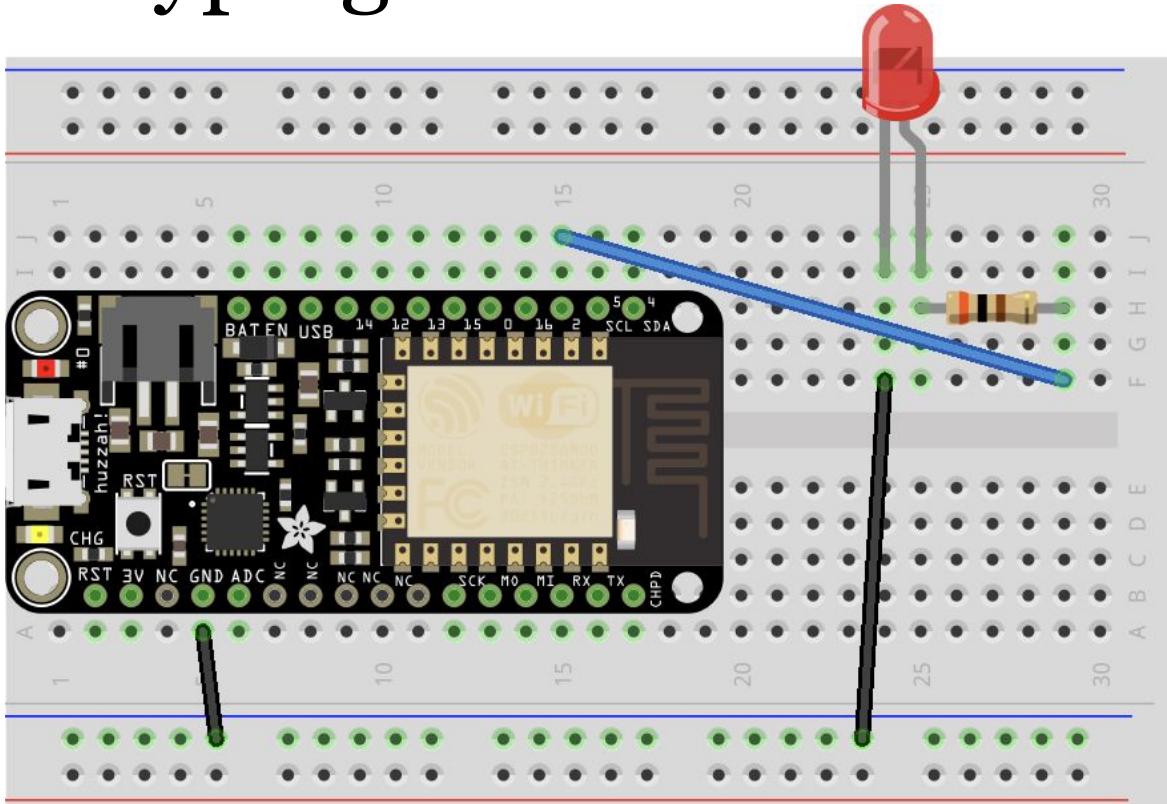
Breadboard and wires, or the Grove standard.

For products, custom PCBs are designed.

Breadboard prototyping

Wire electronic components, no soldering.

Under the hood, the columns are connected, also the power rails.

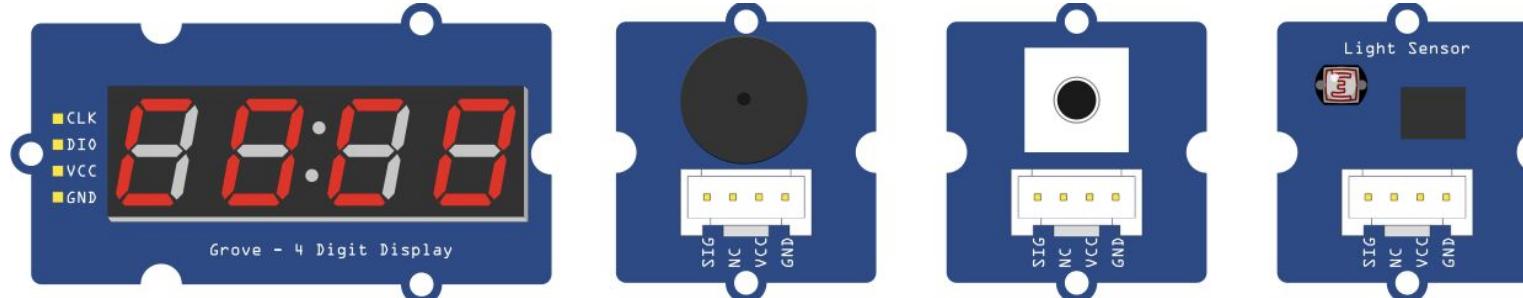


Grove wiring standard

Grove is a simple way to wire sensors and actuators.

It defines wires for power, ground and two signals.

Signals can be digital, analog, UART serial or I₂C.



Arduino example code

Each Arduino library comes with example code.

And there are a number of basic examples.

See *Arduino IDE > File > Examples*

GPIO pin numbers may vary.

Use the nRF52480 Sense pinout or adapter pins.

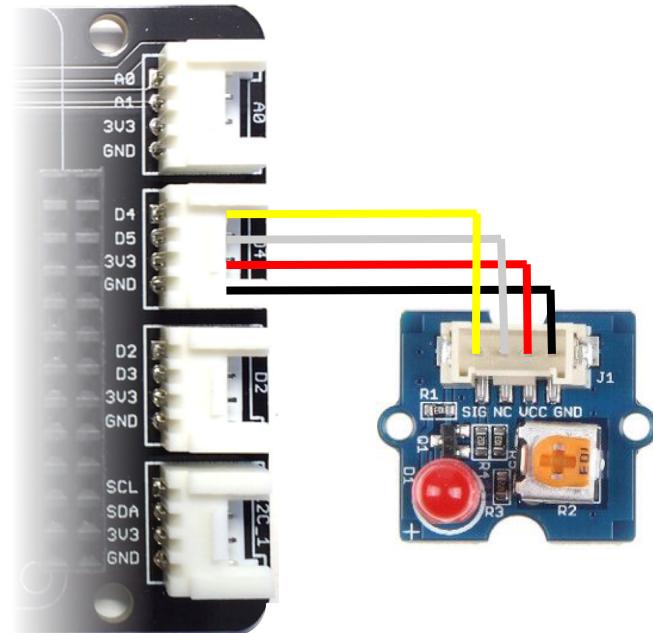
Blinking a LED (digital output)

Try *Examples > Basics > Blink*

Use *LED_BUILTIN*, i.e. pin 13.

Or wire a LED to Grove port *D4*.

D4 maps to nRF52840 pin 9.



The same code works with the buzzer.

Blinking a LED (digital output)

```
pin = 13; // or 9 for Grove D4

void setup() { // called once
    pinMode(pin, OUTPUT); // configure pin
}

void loop() { // called in a loop
    digitalWrite(pin, HIGH); // switch pin on
    delay(500); // ms
    digitalWrite(pin, LOW); // switch pin off
    delay(500); // ms
}
```

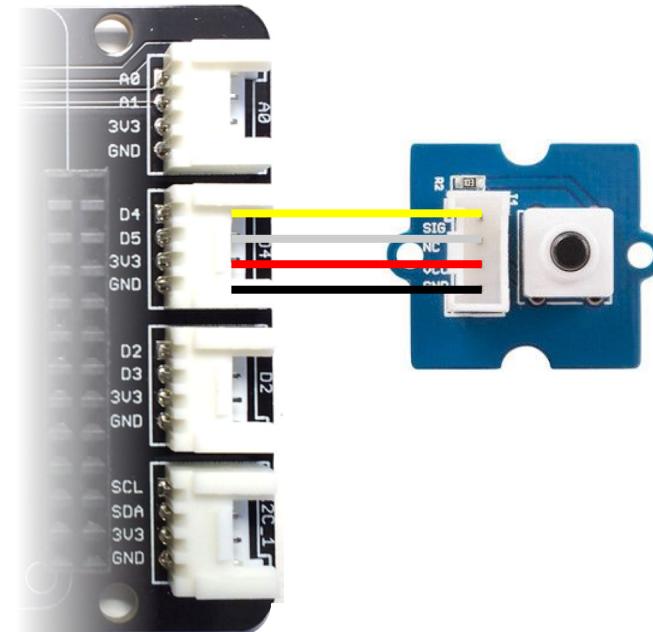
Reading a button (digital input)

Try *Basics > DigitalReadSerial*

Use the onboard button, pin 7.

Or wire a button to Grove D4.

D4 maps to nRF52840 pin 9.



Use the serial console to see output.

Reading a button (digital input)

```
pin = 7; // or 9 for Grove D4

void setup() { // called once
    pinMode(pin, INPUT_PULLUP); // or INPUT
    Serial.begin(9600);
}

void loop() { // called in a loop
    int value = digitalRead(pin);
    Serial.println(value);
    delay(500); // ms
}
```

Hands-on, 15': Button-triggered LED

Use blue onboard LED, pin 4, and the button, pin 7.

Combine the previous examples to switch the LED.

Or wire a LED to Grove port *D2* and a button to *D4*.

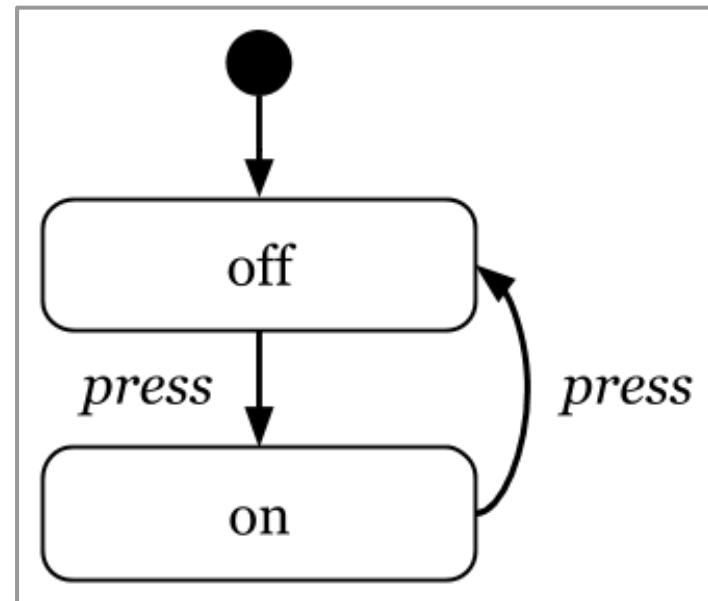
Use the **pin mapping** to adapt the pin numbers.

State machine

A (finite-) **state machine** is a simple way to manage state in embedded programs.

System is in one state at a time,
inputs trigger state transitions.

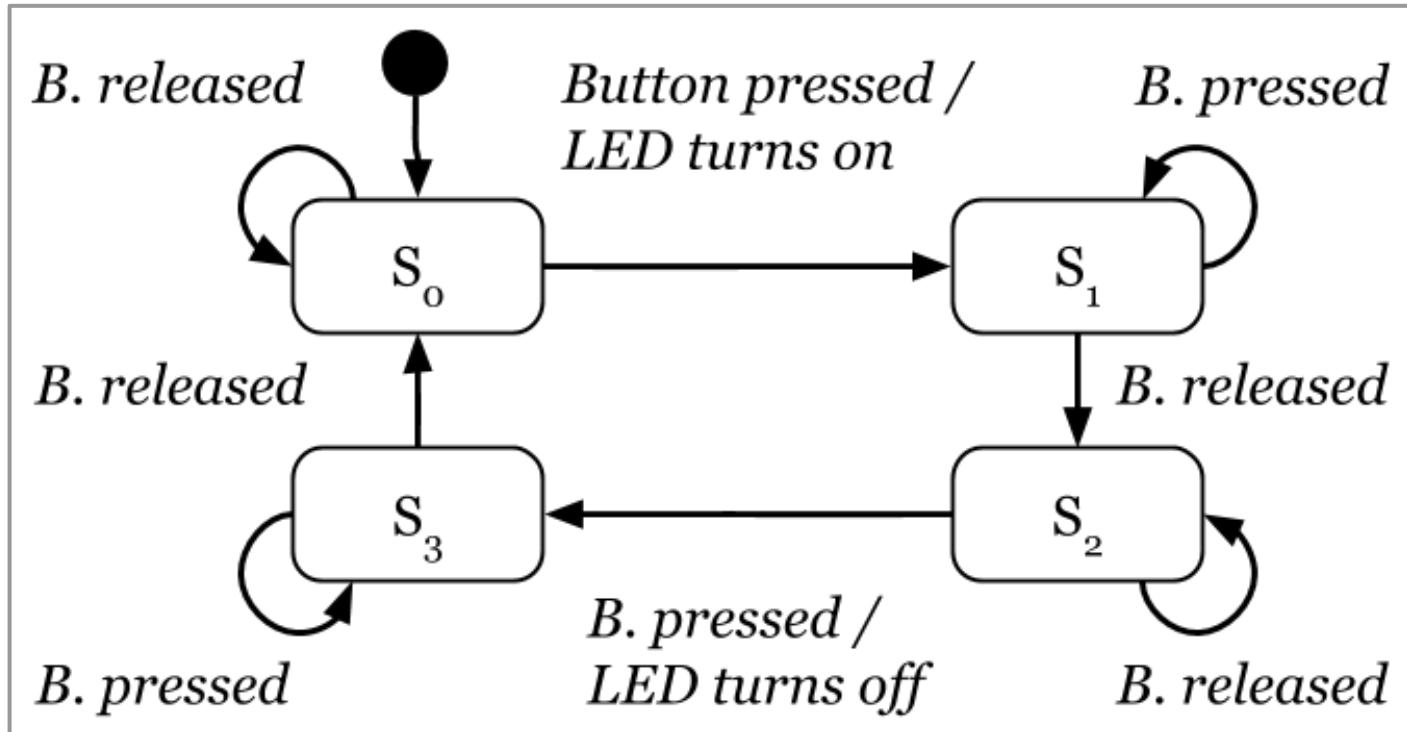
E.g. 1st button *press* => light *on*,
2nd button *press* => light *off*,
3rd => *on*, 4th => *off*, etc.



State machine (refined)

Button is
pressed or
released.

LED can
be turned
on or *off*.



State machine (code snippet)

```
int b = digitalRead(buttonPin);
if (s == 0 && pressed(b)) { // s is state
    s = 1; digitalWrite(ledPin, HIGH); // on
} else if (s == 1 && !pressed(b)) {
    s = 2;
} else if (s == 2 && pressed(b)) {
    s = 3; digitalWrite(ledPin, LOW); // off
} else if (s == 3 && !pressed(b)) {
    s = 0;
}
```

Hands-on, 15': State machine

Copy and complete the code of the state machine.

Make sure it works, with a button and LED setup.

Change it to switch off only, if the 2nd press is *long*.

Let's define long as > 1s, measure time with [millis\(\)](#).

Commit the resulting code to the hands-on repo.

Reading a light sensor (analog input)

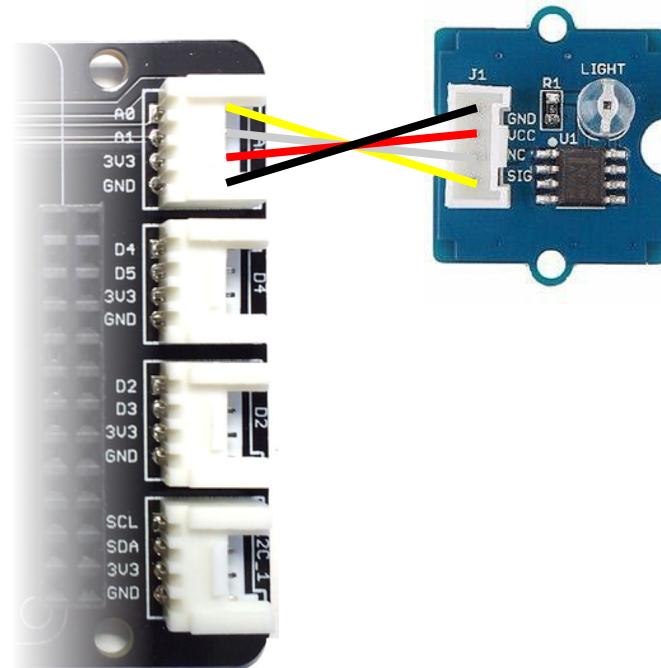
Use *Basics > AnalogReadSerial*

Wire the sensor to port/pin *Ao*.

The analog value is, e.g. 0-1024*

```
int value = analogRead(pin);
```

Use **serial plotter** to see output.



*Range depends on **ADC resolution**.

Mapping input to value range

Sometimes mapping sensor value ranges helps, e.g.

0 - 1024 analog input => 0 - 9 brightness levels.

Arduino has a simple map() function for this:

```
int map(value, // measured input value  
       fromLow, fromHigh, // from range  
       toLow, toHigh); // to range
```

e.g. result = map(value, 0, 1024, 0, 9); 35

Measuring humidity (SHT30)

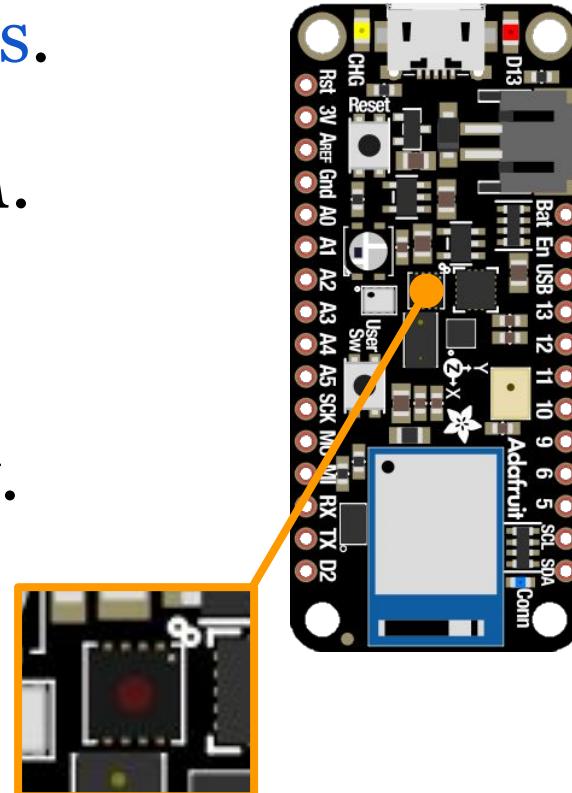
Onboard sensor, wired via **I₂C bus**.

I₂C uses *3V3*, *GND*, *SCL* and *SDA*.

I₂C address of the sensor is *0x44*.

Requires *Adafruit_SHT31* library.

This, more sensors in the Wiki.



CC BY-SA, adafruit.com, fritzing.org

Hands-on, 15': Humidity Alert

Design a state machine with this specification:

Button press sets humidity $\pm 10\%$ as threshold.

Red LED turns on, as long as monitoring is active.

Once threshold has been crossed, blue LED turns on.

Button confirms alert, red led turns on for 1 s, then off.

Summary

We programmed a microcontroller in (Arduino) C.

We used digital and analog sensors and actuators.

We learned to design and code a state machine.

These are the basics of physical computing.

Next: Bluetooth Low Energy.

Challenge

Implement the humidity alert you designed before.

Commit the code and docs to the hands-on repo.

Test your device in a humid environment*.

*Never submerge or rinse electronics.

Feedback or questions?

Write me on Teams or email

thomas.amberg@fhnw.ch

Thanks for your time.

Mobile Computing

From Prototype to Connected Product

CC BY-SA, T. Amberg, FHNW

(unless noted otherwise)

Slides: tmb.gr/mc-pro



[Source](#) on GDocs

Overview

These slides show *how prototypes become products.*

How to get from an idea to a physical prototype.

Scaling up from small batch to mass product*.

*Or at least some insight on what it takes.

Prerequisites

We'll use [OpenSCAD](#) software to create a 3D design.

And [Cura](#), a tool to prepare a 3D design for printing*.

*No hardware is required for this lesson.

Building connected products

So far we focused on firmware and backend software.

To create a real product, there are additional steps*:

Ideation → Prototyping → Development → Testing →

Production → QA → Logistics → Marketing → Sales →

Operations → Support → Maintenance → Sunsetting.

*Vendor perspective, there might be iterations.

Ideation

There are methodologies to find product ideas.

E.g. [Know Cards](#) by Alex Deschamps-Sonsino.

Or the [IoT Design Kit](#) by Studio Dott in Belgium.

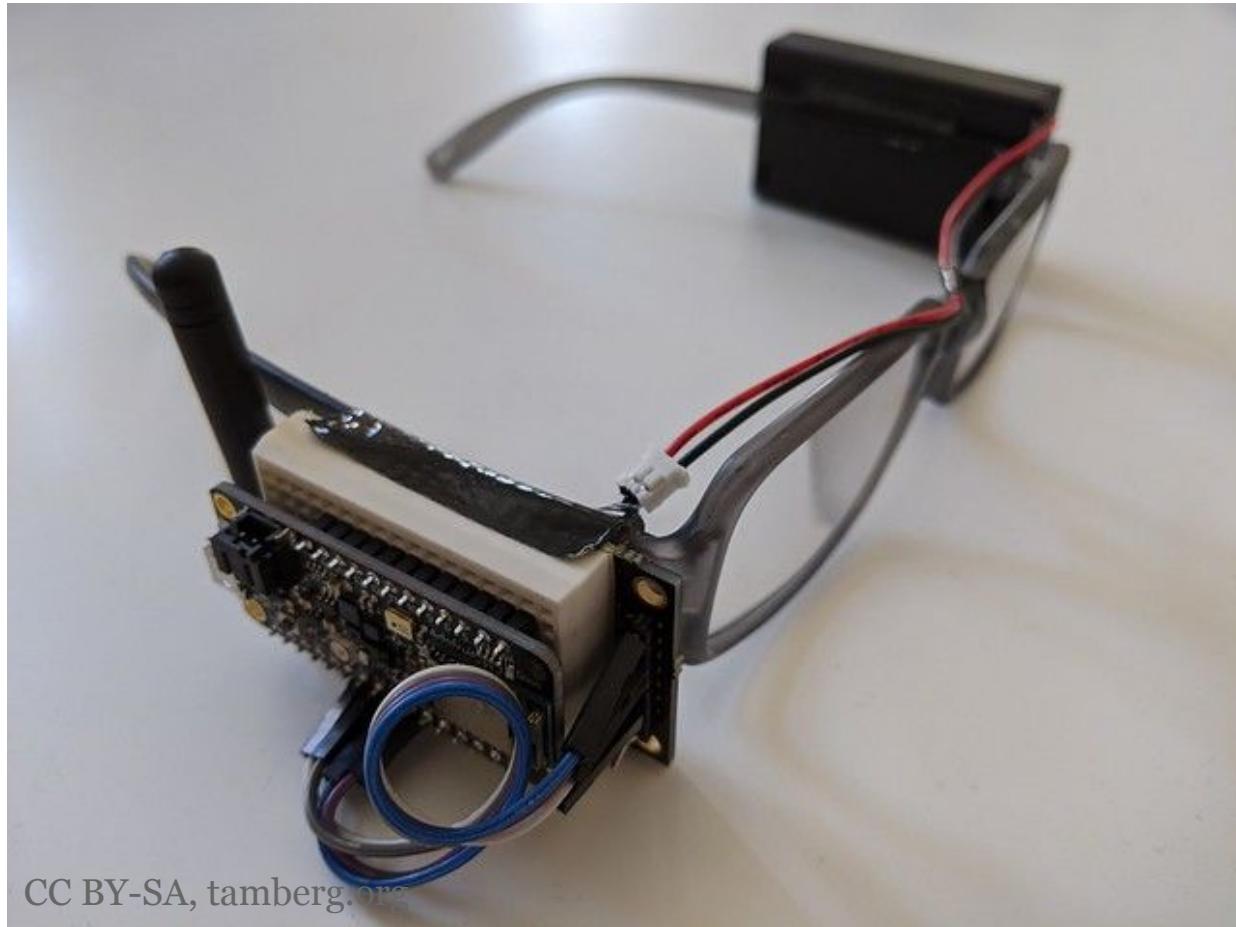
Or [Loaded Dice](#) by Albrecht Kurze, TU Chemnitz.

Attending hackdays is another way to get new ideas.

Hackdays

Assemble a few people w/ complementary skill sets and roles.

Let them build whatever they can imagine.



CC BY-SA, tamberg.org

Prototyping

There are roughly these three variants of prototypes:

Design prototype, how the device will look and feel.

Functional prototype, device/service user experience.

Technical prototype, real hardware, often still too big.

These prototypes can be developed in parallel.

Prototypes

Left: Functional prototype, most electronics are in the base.

Right: Design prototype, no electronics.



CC BY-SA, tamberg.org

Product development

Software / hardware are often developed in parallel.

Technical prototypes are miniaturised, integrated.

Functional prototypes are re-implemented.

Design prototypes are productised.

Testing and certification

Software and hardware, unit and integration testing.

Field studies, with representative (beta) user groups.

Certification, for each new HW version of a product.

QA at the factory for every instance of a device type.

Monitoring backend services to guarantee SLA.

Important IoT system qualities

Security, to keep devices, network & backend secure.

Privacy, to keep people in control of their own data.

Interoperability, to become part of an ecosystem.

Openness, standards & open source build trust.

See, e.g. [betteriot.org principles](http://betteriot.org/principles) for guidance.

Prototyping at a Fab Lab

A shared workshop for personal digital fabrication.

Computer-controlled tools to make almost anything:

3D printers, laser cutters, CNC mills, electronics lab.

Coming from MIT, **Fab Labs** are a global movement.

Made in a Fab Lab => can be replicated in others.

3D printing

Additive manufacturing, builds objects layer by layer.

Fused deposition modeling (FDM) is commonly used.

Filament materials include PLA and ABS (like Lego).

Stereolithography (SLA) is great for very small parts.

Selective laser sintering (SLS) is rather expensive.

Many schools have printers, e.g. **FHNW, ZHAW**.

3D printing process

Create a 3D model with CAD software, export as STL.

Printer-specific slicer* cuts the STL model into layers.

Resulting GCODE file is transferred to the 3D printer.

Printing takes 10-s of minutes up to multiple hours.

E.g. [Cura](#), [Slic3r](#) or [Prusa Slicer](#).

CAD software

CAD (computer aided design) tools for 3D modelling:

Commercial tools, e.g. [Onshape](#), [Rhino](#), [Solidworks](#).

Some are free to use for students, e.g. [Fusion 360](#).

Open source tools, e.g. [Blender](#), [OpenSCAD](#).

All of these can export STL files.

Hands-on, 15': Parametric design

Download [OpenSCAD](#) and create a simple 3D design.

OpenSCAD is a domain specific [language for CAD](#).

Objects can be built by [subtracting](#) simple shapes.

Design a box that fits a [Raspberry Pi Zero](#).

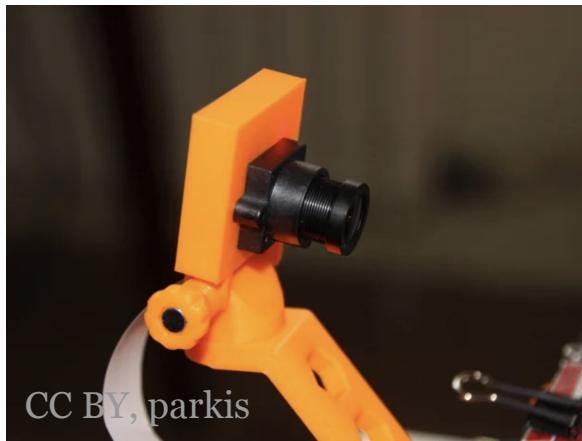
Export as STL and slice it e.g. in [Cura](#).

Done? Take a look at [this model](#).

Design repository

E.g. [Thingiverse](#), for openly licensed 2/3D models.

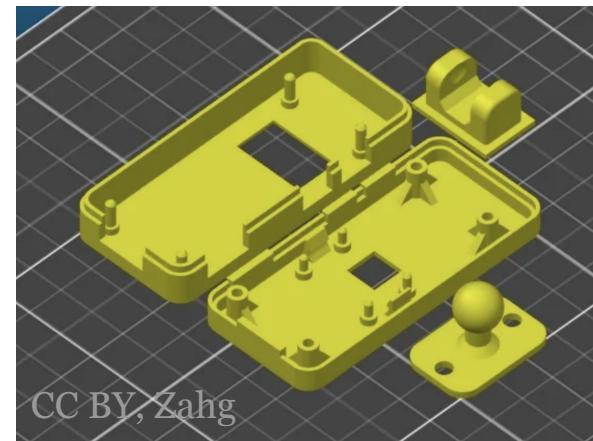
Here's a selection of Raspberry Pi camera adapters.



CC BY, parkis



CC BY-SA, Douglas K



CC BY, Zahg

Laser cutting

Create a 2D design with vector based CAD software*.

Lasers know vector file formats like AI, DXF or PDF.

Materials include wood and acrylic sheets, < 10 mm.

Power/speed settings depend on material/thickness.

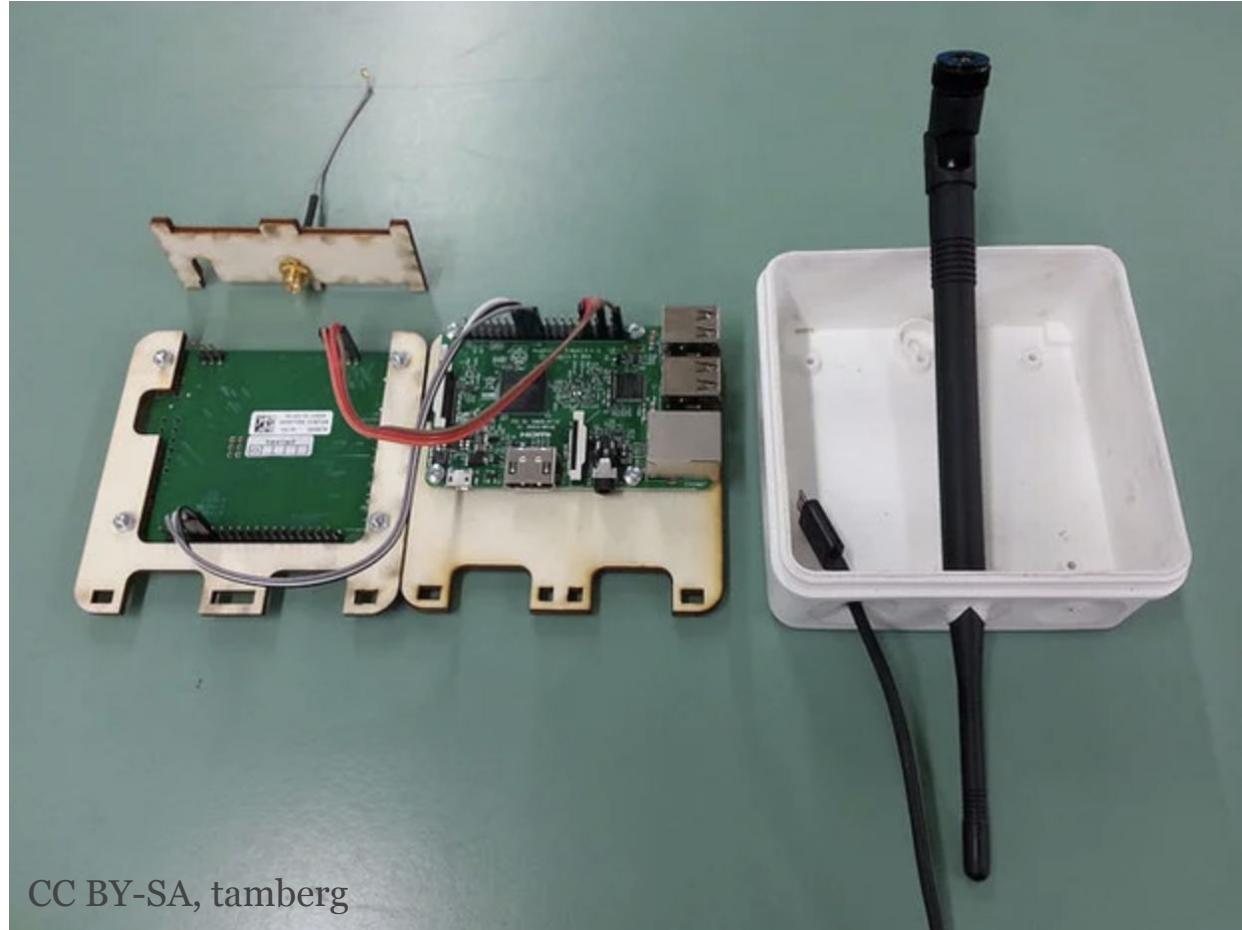
Laser cutting is fast, usually just takes a few minutes.

*E.g. [Inkscape](#), a free and open source design tool.

Laser-cut adapters

A quick way to fit electronics into an existing enclosure.

This is a simple LoRa gateway.



CC BY-SA, tamberg

CNC milling

Subtractive manufacturing, cuts object out of a block.

3 or more axes, working area from 10-s of cm up to m.

2D or 3D CAD model is prepared with CAM software.

Materials include PCB, foam and wood, metal is hard.

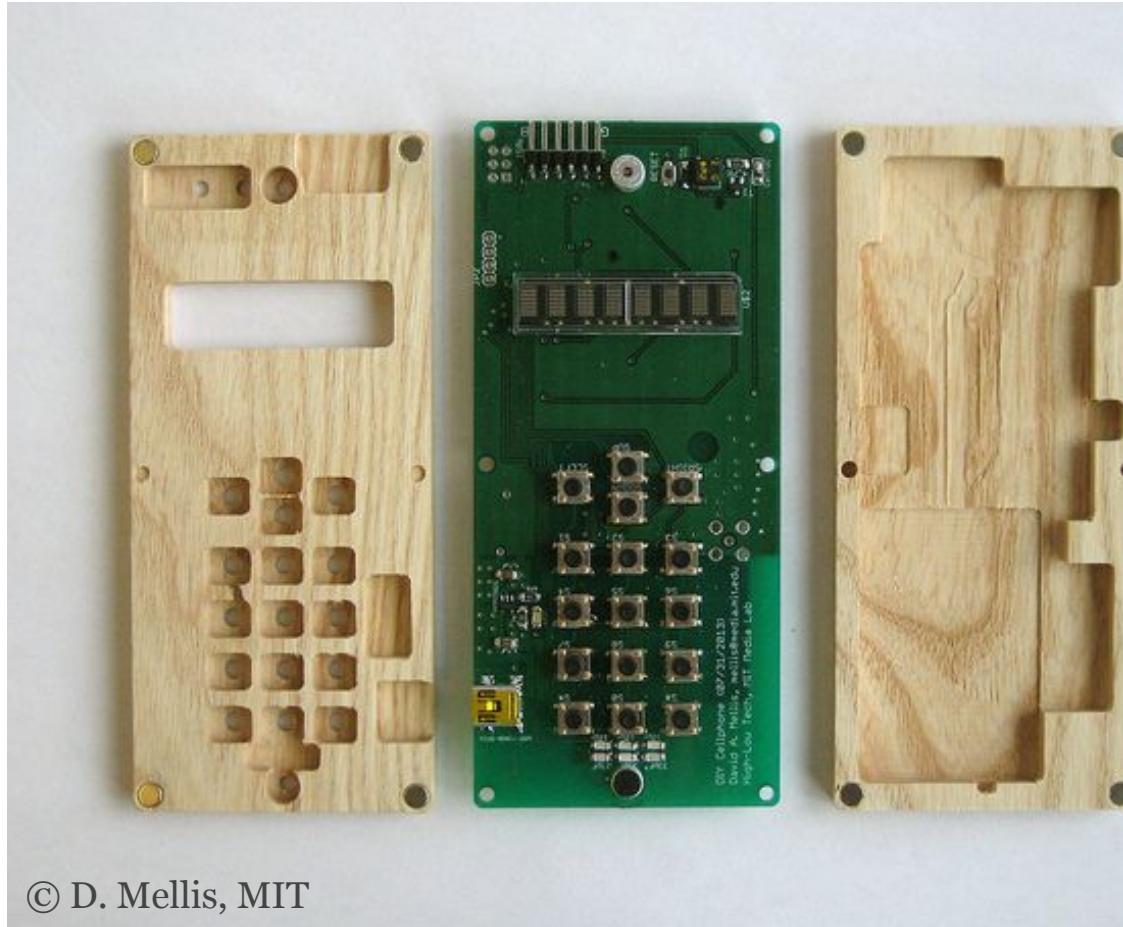
Tool head type and size must be taken into account.

Basic CNC milling can be learned in about a day.

CNC milled enclosure

Nice materials
for prototypes,
too expensive in
production.

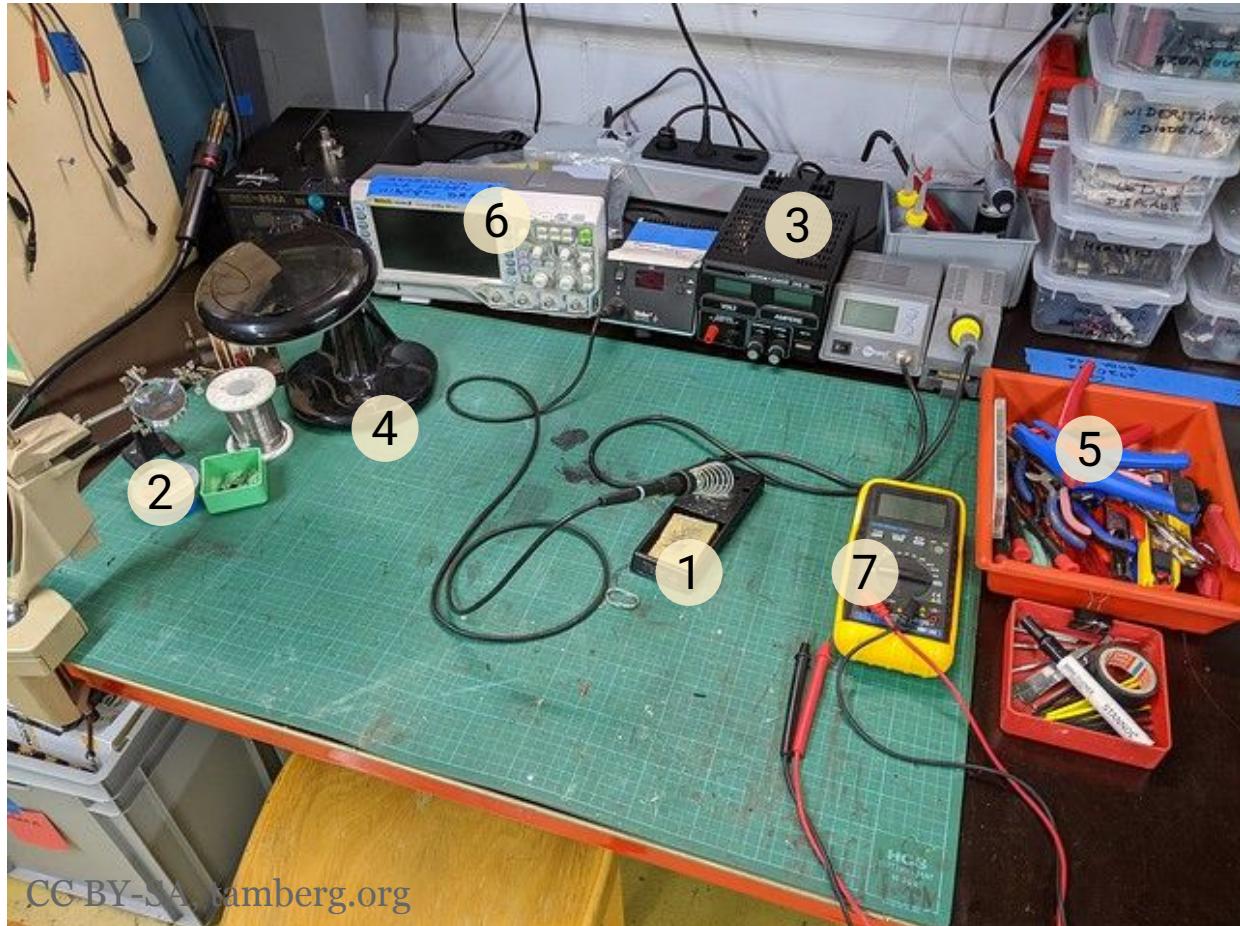
This is a simple
DIY cell phone.



© D. Mellis, MIT

Electronics laboratory

Soldering iron¹,
helping hands²,
power supply³,
lamp⁴, pliers⁵,
oscilloscope⁶,
multimeter⁷.



Designing a custom PCB

A *printed circuit board* (PCB) helps to integrate parts.

E.g. a controller, sensors, a radio & a battery holder.

The traces on a PCB are designed with a layout tool.

Electronic parts come with ready to use footprints.

Layout software includes [Fritzing](#), [Eagle](#) or [Kicad](#).

Producing a custom PCB

Single or double sided PCBs can be etched "by hand".

But it's easier and quite fast to get your PCBs made.

E.g. [Aisler](#), [OSHPark](#), [PCBWay](#) make small batches.

Electronic components can then be hand soldered.

Sourcing parts

The list of all parts is called *bill of materials* (BOM).

Shops for makers include [Adafruit](#), [Sparkfun](#) & [Seeed](#).

Electronic parts suppliers are, e.g. [Digikey](#) & [Mouser](#).

Or platforms like [AliExpress](#) and [Taobao](#) in China.

And for large batches, talk to manufacturers.

Prototype product

Focused on quick development, OK for beta users.

Off-the-shelf components, easy to use modules.

Off-the-shelf box, or 3D printed enclosure.

Custom PCB, manual assembly/soldering.

Batch size 1 to 100.

Small batch product

Focused on getting a certified product to real users.

Off-the-shelf, certified modules to save one-off cost.

Off-the-shelf or custom injection molded enclosure.

Custom PCB, automated assembly. Manual QA.

Batch size 100 to a few 1000.

Mass product

Optimised for cost, ease of use to reduce support.

Custom, certified PCB with integrated modules.

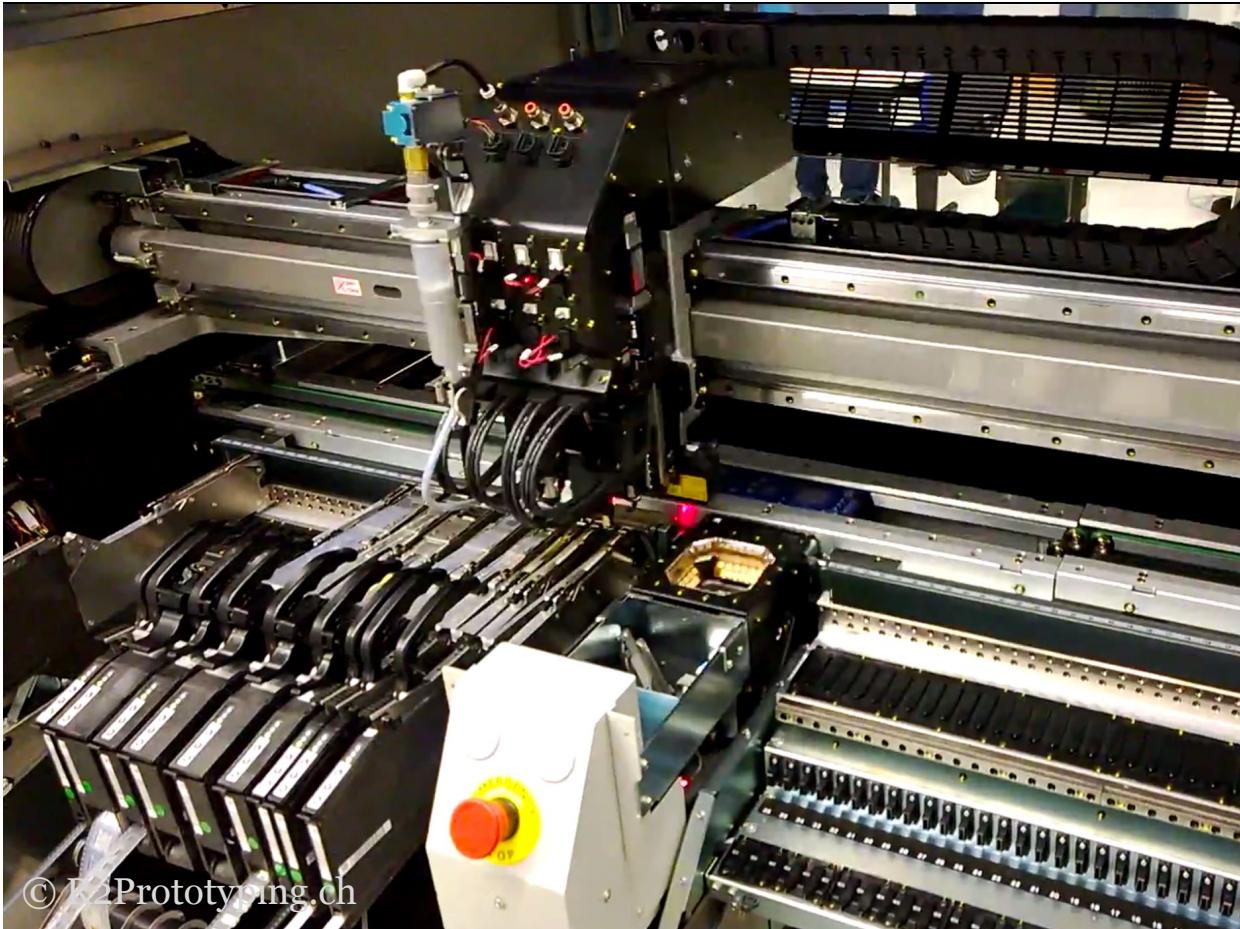
Custom injection molded parts, ~\$10k per mold.

Automated assembly and more automated QA.

Batch size over 10'000.

Automated assembly

A pick & place machine puts parts on a PCB which are then soldered in an oven.



© K2Prototyping.ch

Product examples

Opening products is a great way to learn about them:

[Avelon Wisely LoRaWAN temp. and humidity sensor.](#)

[Gardena smart gateway](#) from [Lemonbeat](#) to Internet.

A [TTN indoor gateway](#) and the [Belkin WeMo switch](#).

[Nest learning thermostat](#) and [Amazon Echo](#) device.

Wisely

MCU, radio
module and
sensors are
integrated
on the PCB.

Off-the-shelf
enclosure.



CC BY-SA, tamberg.org

Gardena gateway

Custom PCB
with existing
Linux SoC¹ and
radio module².

Custom plastic
enclosure.

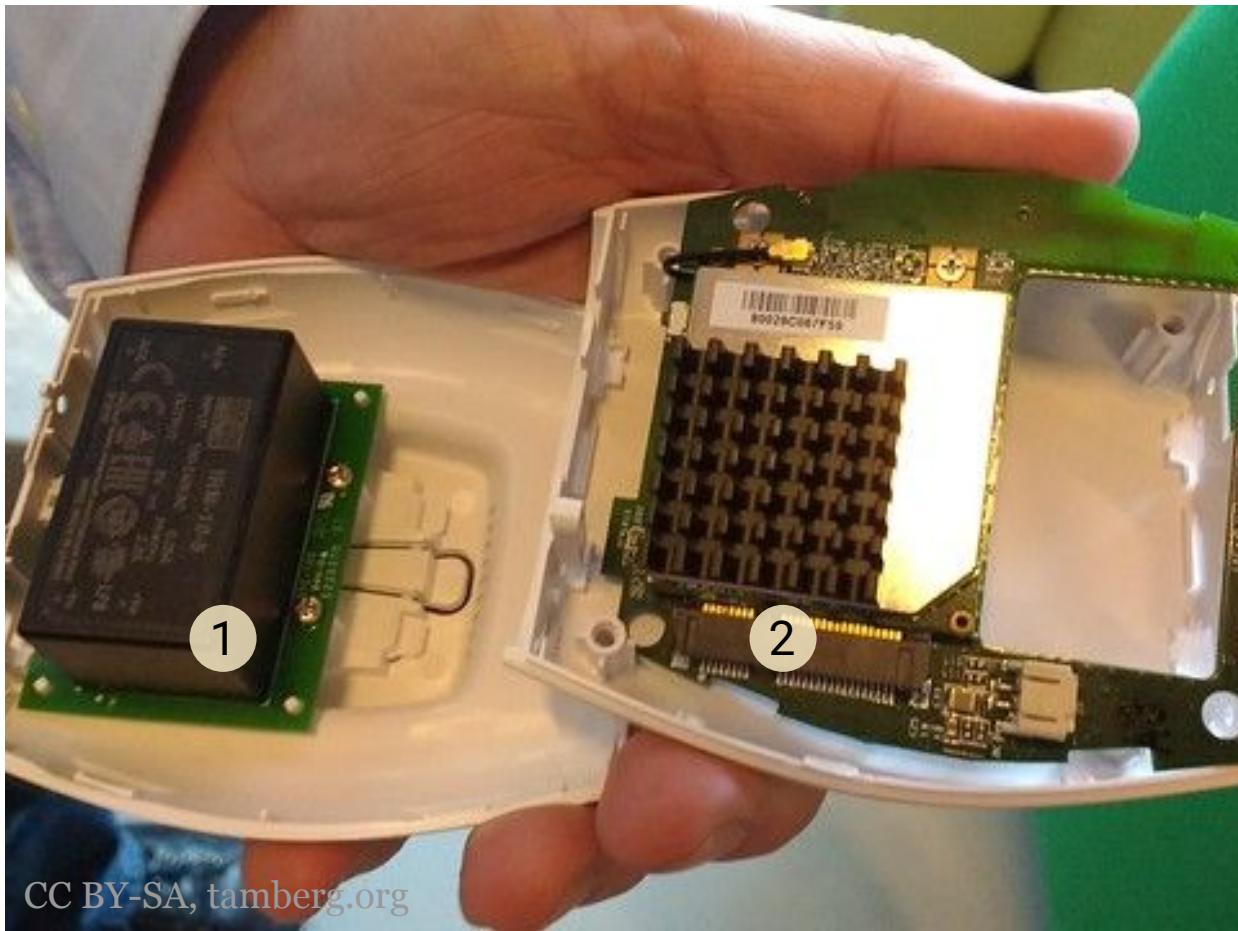


CC BY-SA, tamberg.org

TTN indoor gateway

Encapsulated power supply¹.

Connector² for Wi-Fi module, probably not yet cost optimised.



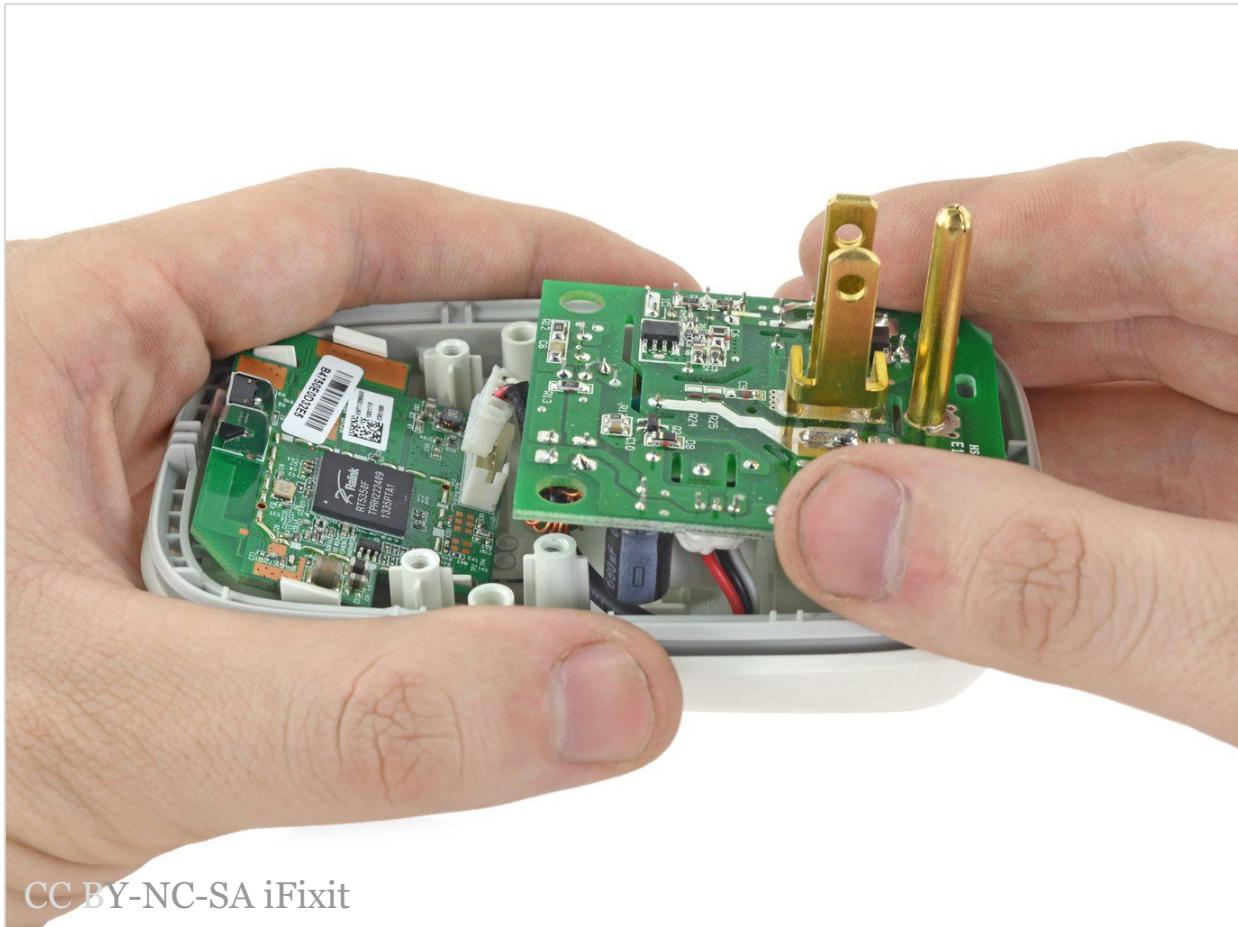
CC BY-SA, tamberg.org

Belkin smart plug

Separate PCB
for high voltage
and Wi-Fi part.

Cost optimised.

See [tear-down](#).



CC BY-NC-SA iFixit

Nest

Optimised for installation by customer.

Built-in level,
spring-loaded
wire terminals.

See [tear-down](#).



CC BY-NC-SA iFixit

Echo

Optimised for
audio quality.

Many, complex
plastic parts.

See [tear-down](#).



CC BY-NC-SA iFixit

Hands-on, 5': Second gen products

Compare the [Echo Dot \(2nd gen\)](#) to the original Echo.

What was changed, and what could be the rationale?

If an injection mold is \$10k, how much was saved?

Which other parts could bring down the price?

Lean startup methodology

Described by Eric Ries, in his book [The Lean Startup](#):

Build, measure, learn, to discover product/market fit.

Minimum viable product, to learn about customers.

Pivot, to change course, test a new hypothesis.

This process iterates towards a working business.

MVP

A minimum
viable product,
made from off
the shelf parts.

VeloTracker.ch
connected bike
tracker / light.

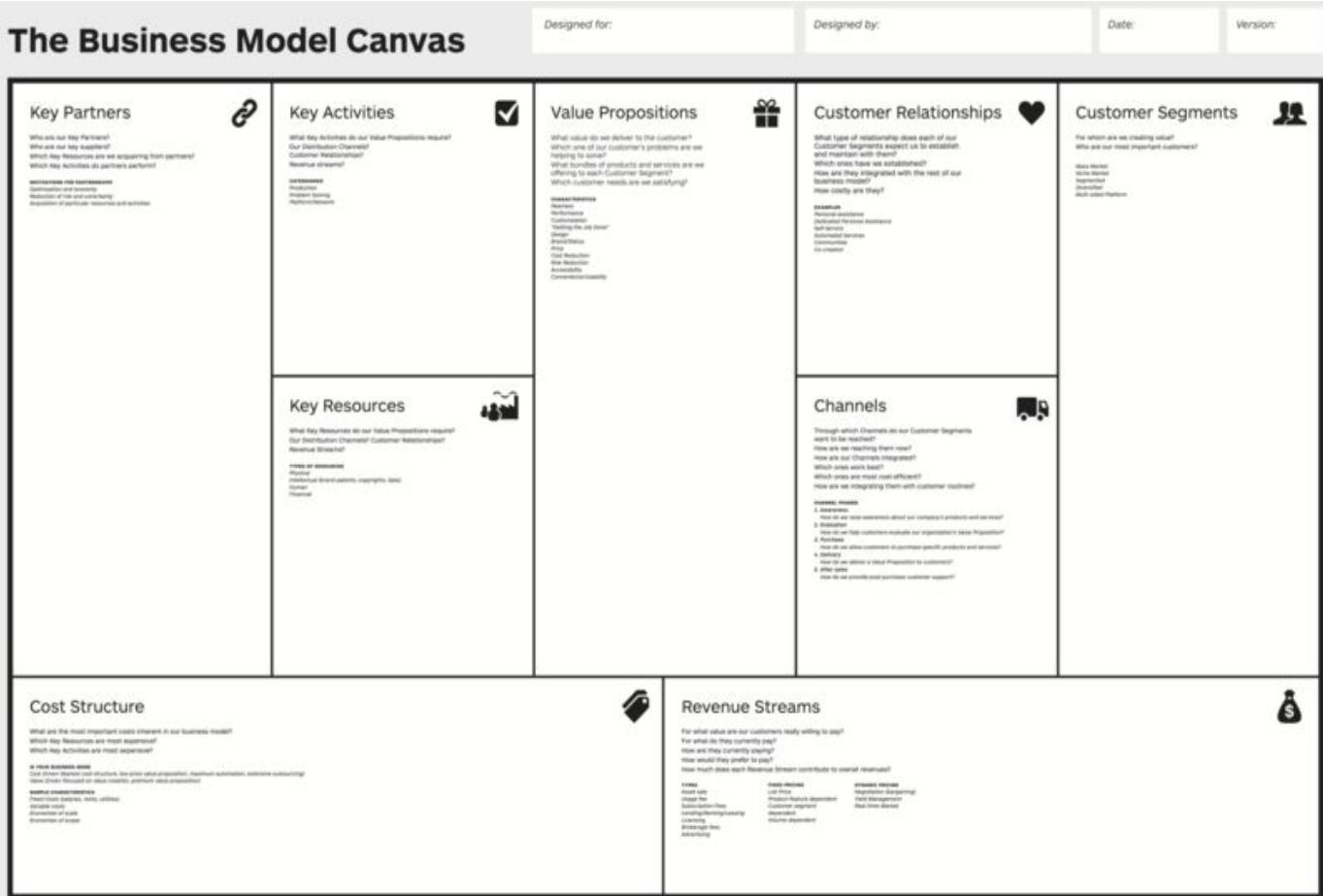


CC BY-SA, tamberg.org

Business model canvas

A tool to
prototype
business
models.

Get it [here](#).



Summary

We saw the steps involved from prototype to product.

We looked at digital fabrication as a prototyping tool.

We learned about PCBs, layout software, pick & place.

We got some insight into production at various scales.

This was the last lesson before the project kick-off.

Feedback or questions?

Write me on Teams or email

thomas.amberg@fhnw.ch

Thanks for your time.

Mobile Computing Sending Sensor Data to IoT Platforms

CC BY-SA, T. Amberg, FHNW

(Screenshots as "fair use")

Slides: tmb.gr/mc-eco



Source on GDocs

Overview

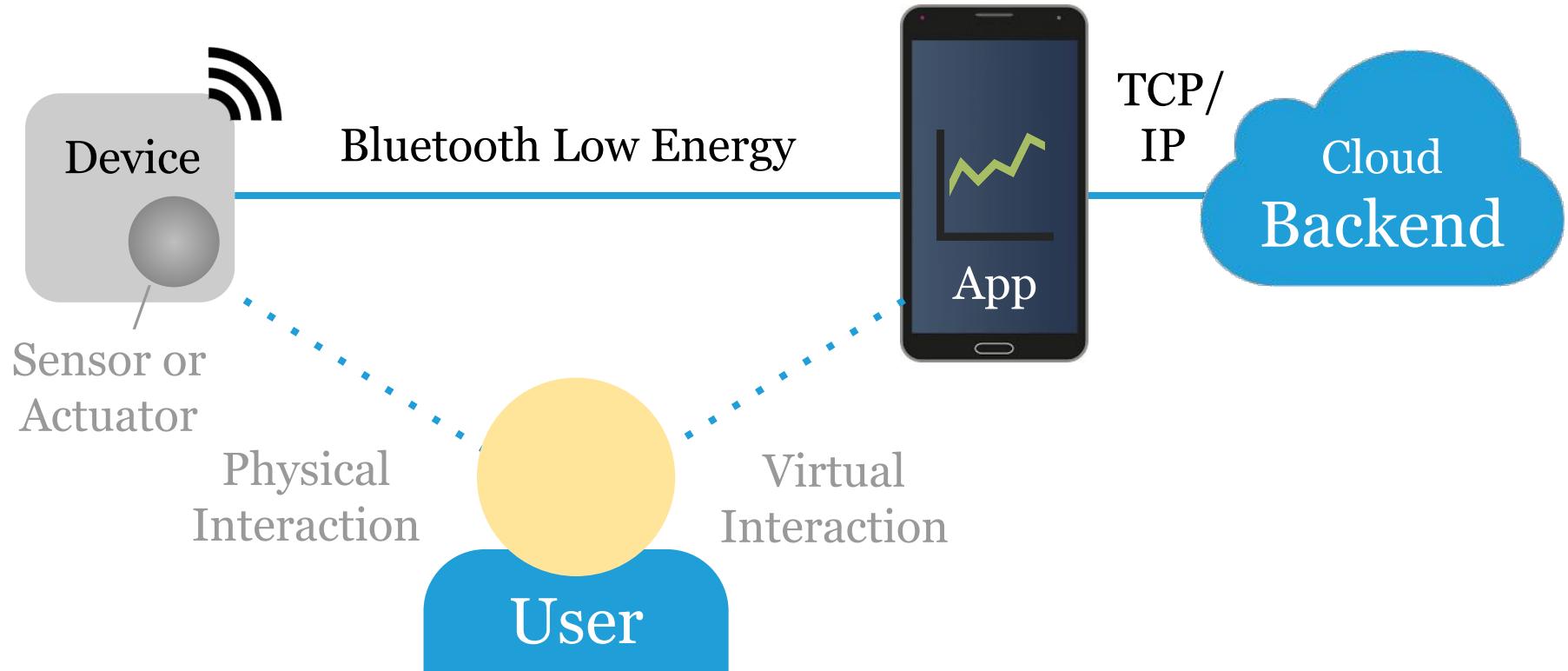
We will send data to an IoT platform or dashboard.

Using HTTP and MQTT, two major IoT protocols.

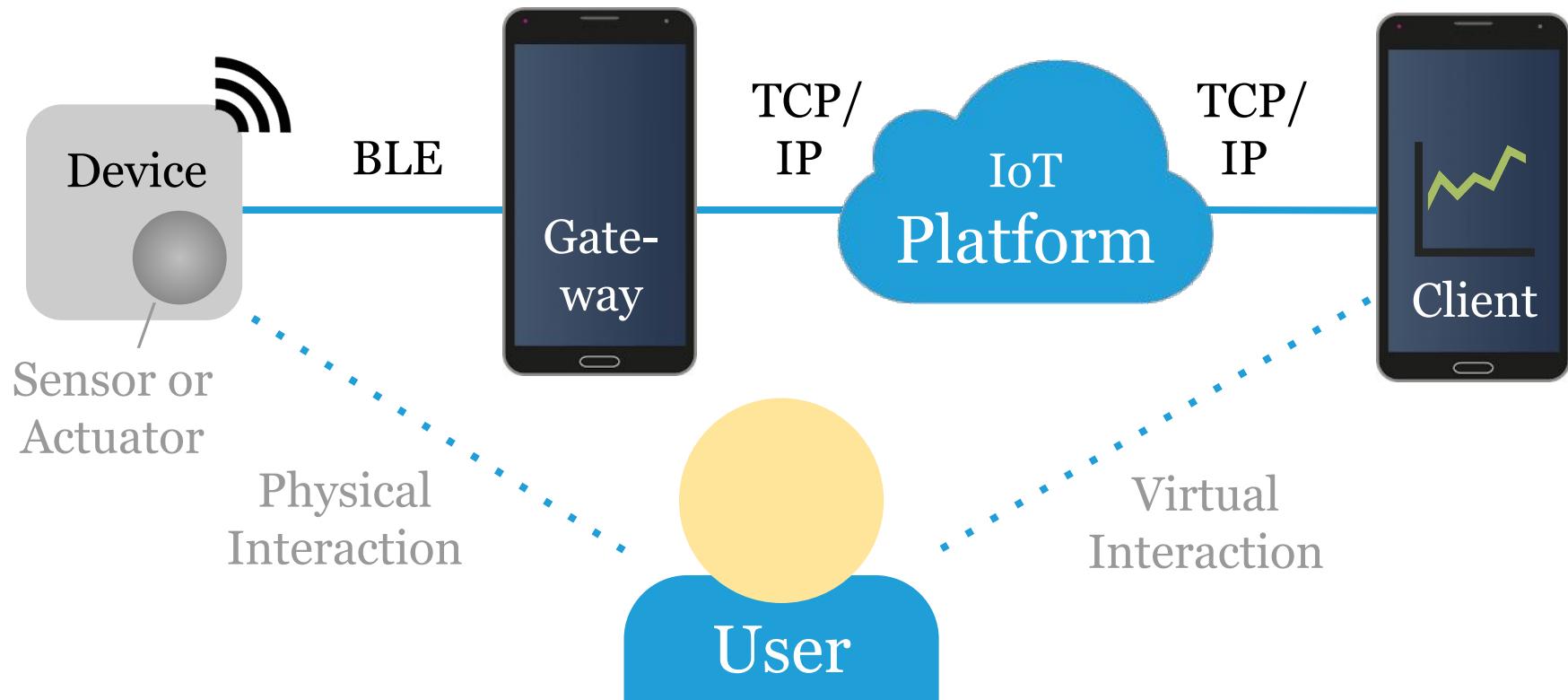
Debug API calls with simple command line tools.

Define API, data format and information model.

Reference model



App has two roles



IoT platforms

IoT platforms make data available at a central point.

There are many platforms, incl. [AWS](#) and [Azure IoT](#).

We will look at a simple IoT platform as an example:

[ThingSpeak](#) provides data storage and visualisation.

It receives and provides data via HTTP or MQTT.

Internet protocol suite layers

RFC 1122 layers are loosely based on the OSI model:

Application layer, process to process, HTTP, MQTT, ...

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.

Hypertext Transfer Protocol (HTTP)

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

It uses TCP/IP as its transport, on port 80 and 443.

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

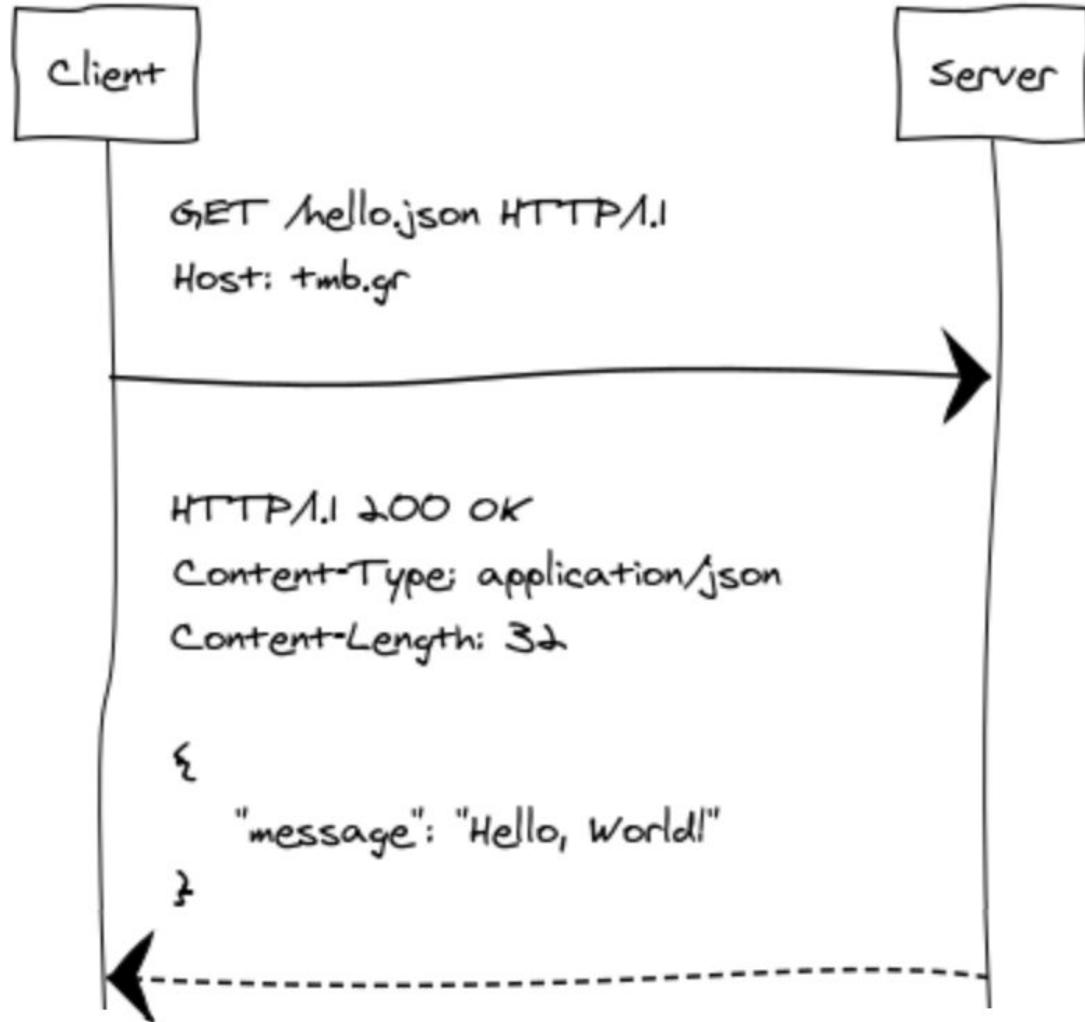
Request and response *headers* are encoded in [ASCII](#).

The content type and length are declared in headers.

HTTP

Web request w/
host header.

Web response
with headers
and content.



Debugging HTTP with Curl and PostBin

Curl (<https://curl.se/>) is a command line 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 "temp:23" https://postb.in/...
```

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

ThingSpeak HTTP API

ThingSpeak has a device- and client-side HTTP API.

Host: api.thingspeak.com

Port: 80 or 443

POST /update?api_key=WRITE_API_KEY&field1=3

GET /channels/CHANNEL_ID/feed.json?

api_key=READ_API_KEY

See Wiki for ThingSpeak cURL examples.

Uniontown Weather Data - Thin X +

https://thingspeak.com/channels/3

ThingSpeak™ Channels Apps Community Support How to Buy Sign In Sign Up

Uniontown Weather Data

Channel ID: 3

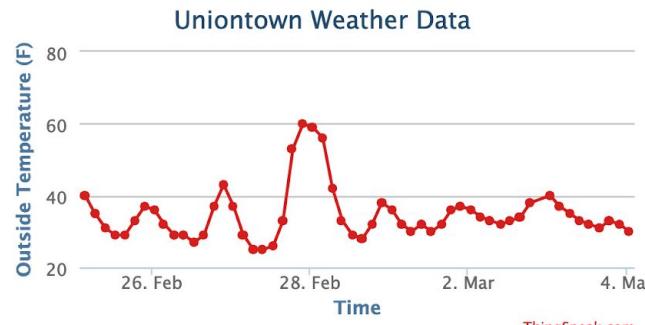
Author: iothans

Access: Public

Weather data from Uniontown, PA

temperature, humidity, weather station, dew point, channel_3

Field 1 Chart



Field 2 Chart



MQTT

MQTT is a standard messaging protocol (v3.1.1, v5.0).

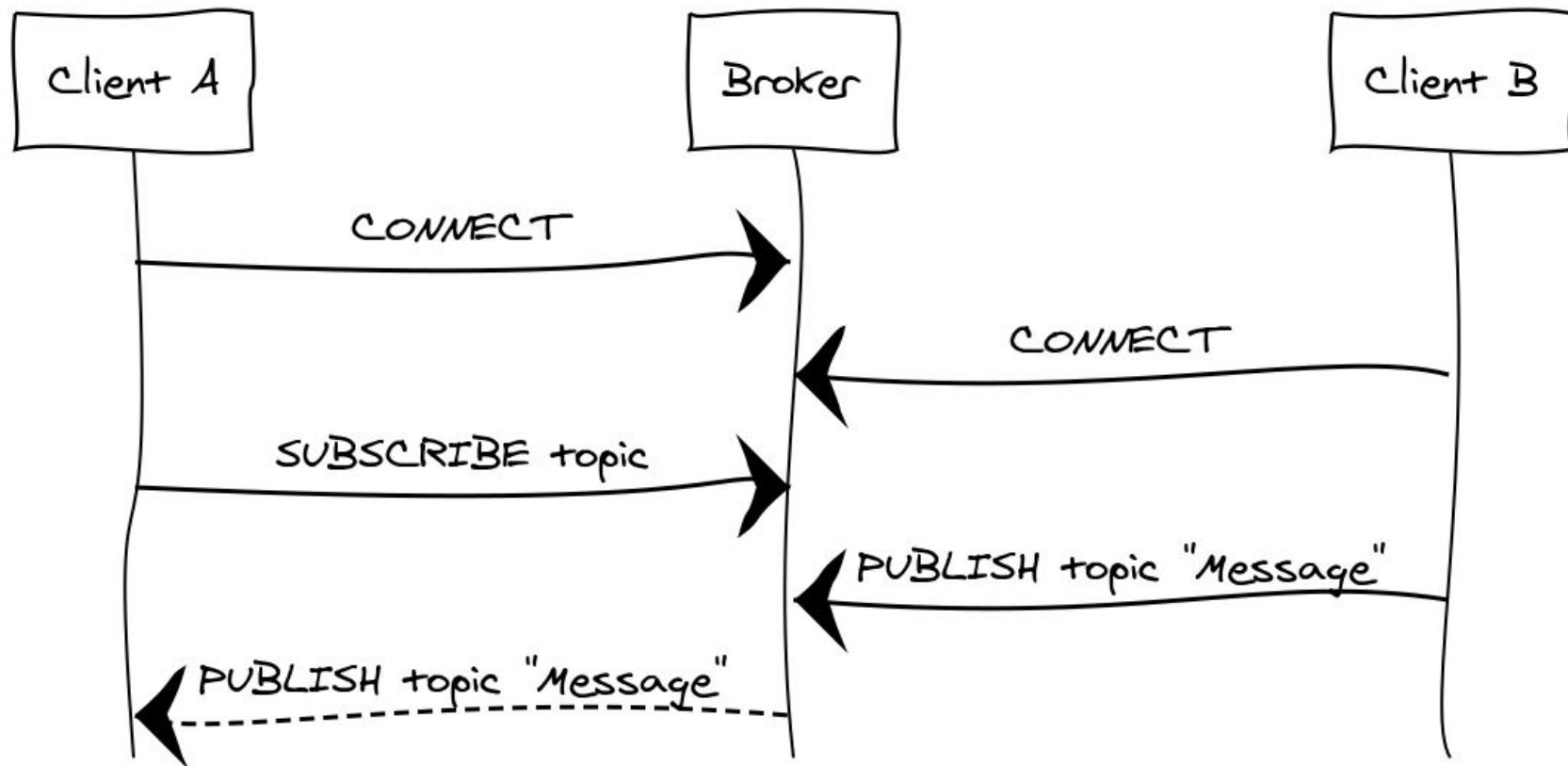
It uses TCP/IP as a transport, on port 1883 and 8883.

In MQTT, *clients* exchange *messages* via a *broker*.

Clients can be *publishers*, *subscribers* or both.

Brokers offer multiple channels, or *topics*.

MQTT



Debugging MQTT with Node.js *mqtt* CLI

Install the Node.js [mqtt](#) command line tool (CLI):

```
$ sudo npm install mqtt -g # adds tool to path
```

To publish/subscribe with the command line tool, try:

```
$ mqtt sub -t 'mytopic' -h 'test.mosquitto.org'  
$ mqtt pub -t 'mytopic' \  
-h 'test.mosquitto.org' \  
-m 'Hello, world!'
```

ThingSpeak MQTT API

ThingSpeak has a **device-** and **client-side** MQTT API.

Host: `mqtt.thingSpeak.com`

Port: 1883 or 8883 (or Websocket: 80, 443)

```
PUB -t 'channels/CHANNEL_ID/publish/\  
WRITE_API_KEY' -m 'field1=42&field2=23'
```

```
SUB -t 'channels/CHANNEL_ID/subscribe/\  
FORMAT/READ_API_KEY'
```

MQTT on Android

Eclipse Paho is an open source MQTT client library.

There are Android specific Java [docs](#) and [examples](#).

An alternative is the [HiveMQ](#) MQTT client library.

Consider MQTT if there is no HTTP endpoint.

General definition of 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.

Data formats

Two parties need to agree on what is valid content.

Parsing means reading individual "content tokens".

Record-based formats, e.g. CSV, are good for tables.

Text-based formats, e.g. JSON are easily readable.

Binary formats, e.g. Protobuf, are more compact.

CSV

Comma Separated Values (CSV), defined in [RFC4180](#)*.

```
file = record *(CRLF record) [CRLF];
record = field *(COMMA field);
field = *TEXTDATA;
CRLF = CR LF;
COMMA = %x2C; CR = %x0D; LF = %x0A;
TEXTDATA = %x20-21 / %x23-2B / %x2D-7E;
```

*Specified in [EBNF](#), simplified for shortness.

JSON

JSON is a simple data format based on Unicode text:

```
{"temp": 23} // try ddg.co/?q=json+validator
```

On Arduino, use e.g. the **Arduino_JSON** library:

```
JSONVar obj = JSON.parse(" {\\"temp\\": 23} " );  
String data = JSON.stringify(obj);
```

On Android, use e.g. **json.org**, **Gson** or **Moshi**.

```
JSONObject obj = new JSONObject(data);  
String data = obj.toString();
```

Protobuf

Protocol Buffers (Protobuf) is a binary data format:

```
message Measurement {  
    required int32 temp = 1;  
    optional int32 humi = 2;  
}
```

Message schemas are compiled to a target language,
i.e. a parser is generated, to be called from your code.

Try this [JSON to Protobuf](#) schema converter.

Information model

The information model defines how data is structured.

It's the "common denominator" of all involved parties.

Data formats define how data is transported or stored.

An information model is more about data semantics*.

*E.g. ThingSpeak knows channels and fields.

Summary

IoT platforms receive and provide data via an API.

E.g. through HTTP requests or MQTT messages.

Data formats allow to write and read payloads.

The information model defines semantics.

Feedback or questions?

Write me on Teams or email

thomas.amberg@fhnw.ch

Thanks for your time.