Containers - where to start?!

Different approaches to run software on the edge of the cloud, respective use case, development resources and scalability

INSYS icom, Michael Kress

Ready-to-use products

Use containers like **apps on a smartphone**. Partners, open source communities or INSYS created fully functional containers for a specific pupose Users just download, configure and use them.

Example: icom Data Suite (iDS)

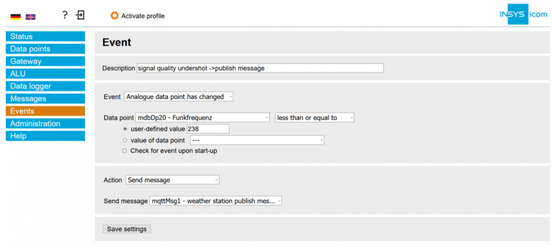

Figure 1:Configure instead of program

Example: Mirasoft AnyViz

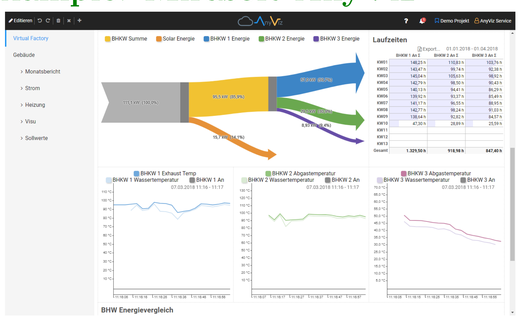

Figure 2:Similar to iDS, but completelly cloud based

Pros and cons

- Configuring instead of programming
- Support from vendor
- May cost license fees
- Maybe hard to extend (add features)

Recommendation

Ideal for **non-programmers.**

Well known Linux distributions

Use Linux distributions like on a **Raspberry Pi**. Minimalistic versions of standard Linux distributions are often well known to user. There are common packet managers included (like apt-get, apk), so additional software can be installed like on a normal Linux PC.

Example: Debian Linux

Figure 3:Debian is a base for a lot of other distributions.

Example: Alpine Linux

Figure 4:Alpine Linux tries to be as slim as possible.

Pros and cons

- Well known tools (like packet managers)
- Incredible wide range of available software
- Very good community support
- Dependent on distribution decisions
- Container size can quickly grow very large
- Updates after long time can be problematic

Recommendation

Ideal for **fast PoC** (Proof of Concept).

Basic Containers

Use **script languages** to program without need for a (cross) compiler.
There are containers with installed programming languages like Python or NodeJS. Users can log into such a container and immediatelly write their programs. Alternatively a program is written on a PC and simply copied into the container.

Example: Python scripts

This little script subscribes to an MQTT broker and displays received data on a simple HTTP server.

```
1  import paho.mqtt.client as mqtt
2  import http.server
3  import socketserver
4
5  def on_connect(client, userdata, flags, rc):
6      client.subscribe("machine/temp")
7
8  def on_message(client, userdata, msg):
9      with open("index.html", "w", encoding='utf-8') as file:
10         file.write(f"<html><body>Temperature:
             {msg.payload}</body></html>")
11
12 mqtt_client = mqtt.Client()
13 mqtt_client.on_connect = on_connect
14 mqtt_client.on_message = on_message
15 mqtt_client.connect_async("mqtt.broker.de", 8889)
16 mqtt_client.loop_start()
17
18 Handler = http.server.SimpleHTTPRequestHandler
19 with socketserver.TCPServer(("", 8080), Handler) as httpd:
20     httpd.serve_forever()
```

Pros and cons

- Minimal environment for low container size
- Programming experience like on PC
- Independent from architecture
- Sustainable because of little mainantance
- No separate SDK or PC software neccessary, programming within container
- Dependent on creator of base container
- Extending functionality can be problematic

Recommendation

Ideal for **scripters** to solve smaller tasks.

Develop for yourself

Use build scripts and SDK (Software Development Kit) to **create own containers**.
Intended for experts: Use the programming language of your choice, use available open or closed source as you please. The container might even consist of only a single binary! Collection of useful links: https://m3-container.net/#scripts

Basic container as template

There are build scripts on github that serve as templates for own containers. Use for all programming languages like C/C++, go or C#.

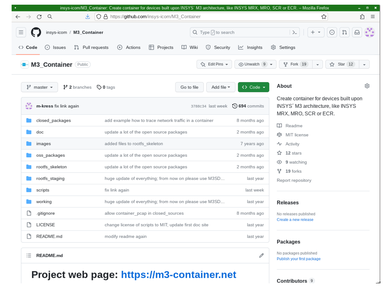

Figure 5:Scripts on github in combination with the SDK create containers

Pros and cons

- Control over everything
- Minimal container size
- Integrate into your existing CI/CD pipeline
- 100% traceable builds
- Minimum surface for hacking attempts
- Least external dependencies
- Requires deeper knowledge
- Initially requires more development time

Recommendation

Ideal for **professional developers**, optimal for mass roll out on huge scale.