ionpy – Vollständige Implementierungs-Roadmap

Device Onboarding Wizard & offene Punkte

Erstellt: 2026-02-27
Status: In Planung
Projekt: ionpy Pro OS

Legende

- □ **Muss** – Blocker, ohne das geht es nicht weiter
- □ **Soll** – Wichtig, sollte in diesem Zyklus erledigt werden
- □ **Kann** – Nice-to-have, verschiebbar
- □ **Erledigt**
- □ **In Arbeit**

PHASE 0: Sofort-Fixes vor allem anderen

(~2-3 Tage – kleine Aufwände, große Wirkung)

Diese Tasks sind Einzeiler oder Kleinstaufwände die **vor jedem weiteren Schritt** erledigt werden müssen, weil sie sonst zu Crashes oder Memory-Leaks führen.

0.1 EventBus Queue maxsize

Datei: `core/event_bus.py`
Aufwand: 5 Minuten
Priorität: □ Muss

Problem:

```
# AKTUELL - kein maxsize = unbegrenztes RAM-Wachstum
def subscribe(self) -> asyncio.Queue:
    q = asyncio.Queue()  # ← KEIN LIMIT!
    self.subscribers.append(q)
    return q
```

Fix:

```
def subscribe(self, maxsize: int = 500) -> asyncio.Queue:
    q = asyncio.Queue(maxsize=maxsize)
    self.subscribers.append(q)
    return q
```

Hinweis für KI: Nur diese eine Zeile ändern. maxsize=500 ist ein guter Startwert – entspricht ca. 5 Sekunden Daten bei 100 Paketen/Sekunde.

0.2 ResizeObserver Memory Leak

Datei: `static/views/chart_view.js`
Aufwand: 15 Minuten
Priorität: Muss

Problem:

```
// onMounted - Observer wird erstellt...
const resizeObserver = new ResizeObserver(() => { ... });
resizeObserver.observe(chartWrapper.value);

// onUnmounted - Observer wird NIE getrennt!
onUnmounted(() => {
    if (metronomeWorker) { ... }
    uplotInstances.forEach(inst => inst.destroy());
    // ← resizeObserver.disconnect() FEHLT
});
```

Fix:

```
// resizeObserver als Variable außerhalb von onMounted deklarieren
let resizeObserver = null;

onMounted(() => {
    // ...bestehender Code...
    resizeObserver = new ResizeObserver(() => {
        // ...bestehender Code...
    });
    if (chartWrapper.value) resizeObserver.observe(chartWrapper.value);
});

onUnmounted(() => {
    if (resizeObserver) resizeObserver.disconnect(); // ← NEU
    if (metronomeWorker) {
        metronomeWorker.postMessage('stop');
        metronomeWorker.terminate();
    }
```

```
    if (workerUrl) URL.revokeObjectURL(workerUrl);
    uplotInstances.forEach(inst => inst.destroy());
});
```

0.3 header_framer.py Stub erstellen

Datei: `devices/framing/header_framer.py` (NEU)
Aufwand: 1 Stunde
Priorität: Muss

Problem: In `TransportFactory` registriert aber Datei existiert nicht → ImportError bei erstem Aufruf.

Fix – Minimaler Stub der nicht crasht:

```
"""
devices/framing/header_framer.py
Stub-Implementierung. Noch nicht produktionsreif.
Verhindert ImportError wenn header-Framing konfiguriert wird.
"""
import asyncio
from devices.framing.base import AbstractFraming


class HeaderFramer(AbstractFraming):
    """
    Längen-Header basiertes Framing.
    Format: [4 Byte Länge (Big Endian)] [Payload]
    TODO: Vollständig implementieren
    """

    def __init__(self, transport, header_size: int = 4,
                 byte_order: str = "big"):
        super().__init__(transport)
        self.header_size = header_size
        self.byte_order = byte_order
        self.logger.warning(
            "HeaderFramer ist noch nicht vollständig implementiert!"
        )

    async def get_next_frame(self, timeout: float = 5.0) -> tuple[bytes,
float]:
        import time
        start = asyncio.get_event_loop().time()

        # Header lesen
        while len(self.buffer) < self.header_size:
            if asyncio.get_event_loop().time() - start > timeout:
```

```
                raise asyncio.TimeoutError("Timeout beim Lesen des Headers")
            new_data = await self.transport.read(
                self.header_size, timeout=0.1
            )
            if new_data:
                self.buffer.extend(new_data)

        # Länge aus Header lesen
        length = int.from_bytes(
            self.buffer[:self.header_size],
            byteorder=self.byte_order
        )
        del self.buffer[:self.header_size]

        # Payload lesen
        while len(self.buffer) < length:
            if asyncio.get_event_loop().time() - start > timeout:
                raise asyncio.TimeoutError("Timeout beim Lesen des
Payloads")
            new_data = await self.transport.read(length, timeout=0.1)
            if new_data:
                self.buffer.extend(new_data)
            else:
                await asyncio.sleep(0.01)

        frame_time = time.time()
        frame = bytes(self.buffer[:length])
        del self.buffer[:length]
        return frame, frame_time

    async def put_frame(self, payload: bytes):
        length = len(payload).to_bytes(
            self.header_size, byteorder=self.byte_order
        )
        await self.transport.write(length + payload)
```

0.4 HID Transport aus Wizard ausblenden

Datei: `devices/transport_factory.py`
Aufwand: 30 Minuten
Priorität: Muss

Problem: `hid_transport.py` ist als UNFERTIG markiert. Sobald der Wizard HID als Option anzeigt und der User es wählt → Crash.

Kurzfristiger Fix – Flag hinzufügen:

```
# devices/transport_factory.py
```

```
TRANSPORT_REGISTRY = {
    "serial":  "devices.transport.serial.Serial",
    "tcp":     "devices.transport.tcp_client.TCPClient",
    "hid":     "devices.transport.hid_transport.HIDTransport",
    "virtual": "devices.transport.virtual.VirtualTransport"
}

# NEU: Welche Transports sind wizard-fähig (fertig implementiert)?
WIZARD_AVAILABLE_TRANSPORTS = {
    "serial":  True,
    "tcp":     True,
    "hid":     False,   # ← Noch nicht fertig
    "virtual": True
}
```

Der Wizard-Endpoint filtert dann:

```
# web/rest.py - in get_available_drivers() oder separatem Endpoint
from devices.transport_factory import TransportFactory,
WIZARD_AVAILABLE_TRANSPORTS

@router.get("/api/config/drivers/{name}/blueprint")
def get_driver_blueprint(name: str):
    driver_class = _find_driver_class(name)
    if not driver_class:
        raise HTTPException(404, "Treiber nicht gefunden")

    blueprints = getattr(driver_class, 'STACK_BLUEPRINTS', {})

    # Nur verfügbare Transports zurückgeben
    available = {
        k: v for k, v in blueprints.items()
        if WIZARD_AVAILABLE_TRANSPORTS.get(k, False)
    }
    return {"driver": name, "blueprints": available}
```

0.5 Strukturierte Fehlerklassen

Datei: `core/exceptions.py` (NEU)
Aufwand: 1 Tag
Priorität: Muss (Wizard braucht strukturiertes Feedback)

```
"""
core/exceptions.py
Zentrale Fehlerklassen für ionpy.
Ermöglicht strukturiertes Fehler-Feedback vom Backend ans Frontend.
"""
```

```python
class IonpyBaseError(Exception):
    """Basis für alle ionpy Fehler."""
    def __init__(self, message: str, hint: str = None, code: str = None):
        super().__init__(message)
        self.message = message
        self.hint = hint      # Benutzerfreundlicher Tipp
        self.code = code      # Maschinenlesbarer Fehlercode

    def to_dict(self) -> dict:
        return {
            "error": self.__class__.__name__,
            "code": self.code,
            "message": self.message,
            "hint": self.hint
        }


class DeviceSpawnError(IonpyBaseError):
    """Fehler beim Erstellen eines Geräts."""
    def __init__(self, device_id: str, stage: str,
                 message: str, hint: str = None):
        super().__init__(message, hint, code=f"SPAWN_{stage.upper()}")
        self.device_id = device_id
        self.stage = stage
        # stage: "transport" | "framing" | "driver" | "connect" | "poll"

    def to_dict(self) -> dict:
        d = super().to_dict()
        d["device_id"] = self.device_id
        d["stage"] = self.stage
        return d


class DeviceNotFoundError(IonpyBaseError):
    def __init__(self, device_id: str):
        super().__init__(
            f"Gerät '{device_id}' nicht gefunden",
            hint="Prüfe ob das Gerät in der config.yaml eingetragen ist",
            code="DEVICE_NOT_FOUND"
        )
        self.device_id = device_id


class DeviceDuplicateError(IonpyBaseError):
    def __init__(self, device_id: str):
        super().__init__(
            f"Geräte-ID '{device_id}' existiert bereits",
            hint="Wähle eine andere eindeutige Geräte-ID",
            code="DEVICE_DUPLICATE"
        )
```

```
class TransportError(IonpyBaseError):
    def __init__(self, transport_type: str, message: str, hint: str = None):
        super().__init__(message, hint,
code=f"TRANSPORT_{transport_type.upper()}")
        self.transport_type = transport_type


class ConfigValidationError(IonpyBaseError):
    def __init__(self, field: str, message: str):
        super().__init__(
            f"Konfigurationsfehler in '{field}': {message}",
            code="CONFIG_INVALID"
        )
        self.field = field


# FastAPI Exception Handler registrieren
# In web/server.py einbinden:
#
# from core.exceptions import IonpyBaseError
# from fastapi.responses import JSONResponse
#
# @app.exception_handler(IonpyBaseError)
# async def ionpy_exception_handler(request, exc: IonpyBaseError):
#     return JSONResponse(status_code=400, content=exc.to_dict())
```

PHASE 1: Fundament – State & Config

(~7 Tage)

1.1 StateManager

Datei: `core/state_manager.py` (NEU)
Aufwand: 1.5 Tage
Priorität: Muss

Zweck: Trennt persistenten Laufzeit-State (poll_interval, Setpoints, User-Einstellungen) von der statischen YAML-Config. Verhindert dass Laufzeit-Änderungen nach Neustart verloren gehen.

Wichtige Implementierungshinweise für KI:

- Atomares Schreiben: immer `tmp → os.replace()` – nie direkt schreiben
- Dirty-Flag Pattern: nur schreiben wenn sich etwas geändert hat
- Korrupte Dateien umbenennen statt löschen (`→ .corrupt`)

- `save_now()` muss synchron aufrufbar sein (für `beforeunload`-Szenario)

```
# Dateistruktur die erzeugt wird:
# data/
# ├── device_state.json    ← StateManager verwaltet das
# ├── workspaces/          ← Bereits vorhanden
# └── logs/                ← Bereits vorhanden

# Beispiel device_state.json:
# {
#   "DPS5005_1": {
#     "u_set": 12.0,
#     "i_set": 2.5,
#     "poll_interval": 0.5,
#     "backlight": 3
#   }
# }
```

Schnittstelle:

```
class StateManager:
    def load(self): ...
    def get_device_state(self, device_id: str) -> dict: ...
    def set_value(self, device_id: str, entity_id: str, value: Any): ...
    def remove_device(self, device_id: str): ...
    async def start_autosave(self, interval: float = 30.0): ...
    async def save_now(self): ...
    async def _write_to_disk(self): ...  # atomar
```

1.2 Entity: persist Flag

Datei: `structures/entities/base.py`
Aufwand: 0.5 Tage
Priorität: Muss

Hinweis für KI: Nur ein neues Feld `persist: bool = False` in `__init__` hinzufügen. Bestehende Entities sind automatisch nicht-persistent (sicherer Default).

```
class BaseEntity:
    def __init__(
        self,
        name: str,
        mode: AccessMode = AccessMode.RO,
        ui: Optional[UIMeta] = None,
        channel: str = "Ch1",
        persist: bool = False,   # ← NEU
    ):
        # ... bestehender Code ...
```

```
        self.persist = persist
```

Faustregeln für persist-Flags in Treibern:

- Messwerte (u_out, i_out, temp, soc) → `persist=False`
- Einstellungen (u_set, i_set, poll_interval) → `persist=True`
- Verbindungsstatus (state, diag_msg) → `persist=False`
- **output_toggle → `persist=False`** (Sicherheit! Kein Auto-Einschalten)

1.3 Engine: StateManager einbinden

Datei: `core/engine.py`
Aufwand: 0.5 Tage
Priorität: Muss

Änderungen:

1. StateManager instanziieren
2. `start()`: State **vor** Config laden
3. `run_hardware()`: autosave Task starten
4. `stop()`: `save_now()` aufrufen
5. `execute_command()`: persistente Entities im StateManager speichern

```
async def execute_command(self, device_id: str, key: str, value: Any):
    dev = self.device_manager.get_device(device_id)
    if dev:
        await dev.execute_command(key, value)
        # NEU: Persistente Entities speichern
        entity = dev.entities.get(key)
        if entity and getattr(entity, 'persist', False):
            self.state_manager.set_value(device_id, key, value)
    else:
        logger.warning(f"Kommando verworfen: '{device_id}' nicht gefunden!")
```

1.4 ConfigManager

Datei: `core/config_manager.py` (NEU)
Aufwand: 1.5 Tage
Priorität: Muss

Zweck: Einzige Klasse die die YAML liest und schreibt. Trennt Config-Persistenz vom Rest der Engine.

Schnittstelle:

```
class ConfigManager:
    def __init__(self, config_path: str): ...
```

```
    def load(self) -> dict: ...
    def get_devices(self) -> list: ...
    def add_device(self, device_conf: dict): ...      # mit Duplikat-Check
    def remove_device(self, device_id: str): ...
    def update_device(self, device_id: str, conf: dict): ...
    def set_device_enabled(self, device_id: str, enabled: bool): ...
    def _write_atomic(self, data: dict): ...          # tmp → rename
    def _create_backup(self): ...                     # → config.yaml.bak
```

Wichtige Implementierungshinweise für KI:

- Vor jeder Schreiboperation ein Backup anlegen
- YAML-Kommentare gehen beim Schreiben verloren (ruamel.yaml

verwenden wenn Kommentare erhalten bleiben sollen,

```
  sonst einfaches yaml.dump ist ok)
* ID-Eindeutigkeit beim add_device prüfen
* ''DeviceDuplicateError'' aus ''core/exceptions.py'' werfen
```

1.5 DeviceManager: State restore

Datei: `core/device_manager.py`
Aufwand: 0.5 Tage
Priorität: Muss

Hinweis für KI: In `_spawn_device()` nach der Instanziierung des Devices `_restore_device_state()` aufrufen. Diese Methode setzt nur RAM-Werte – sendet **keine** Hardware-Befehle.

```
def _restore_device_state(self, device, saved_state: dict):
    for entity_id, value in saved_state.items():
        entity = device.entities.get(entity_id)
        if not entity:
            continue
        if not getattr(entity, 'persist', False):
            continue
        entity.value = value  # Nur RAM, kein Hardware-Befehl!
```

1.6 Config-Schema Validierung

Datei: `core/config_validator.py` (NEU)
Aufwand: 1 Tag
Priorität: Soll

Zweck: Tippfehler in config.yaml führen aktuell zu stillen Fehlern. Pydantic gibt klare Fehlermeldungen.

```
from pydantic import BaseModel, validator
from typing import Optional, Literal

class TransportConfig(BaseModel):
    type: Literal["serial", "tcp", "hid", "virtual", "visa"]
    params: dict = {}

class FramingConfig(BaseModel):
    type: Literal["delimiter", "header", "modbus_rtu",
                  "start_stop", "none"]
    params: dict = {}

class DeviceConfig(BaseModel):
    id: str
    driver: str
    enabled: bool = True
    auto_start: bool = True
    alias: Optional[str] = None
    icon: Optional[str] = "developer-board"
    description: Optional[str] = ""
    location: Optional[str] = ""
    transport: TransportConfig
    framing: FramingConfig = FramingConfig(type="none")

    @validator('id')
    def id_must_be_valid(cls, v):
        if not v.replace('_', '').replace('-', '').isalnum():
            raise ValueError('ID darf nur Buchstaben, Zahlen, _ und -
enthalten')
        return v

class RootConfig(BaseModel):
    devices: list[DeviceConfig] = []
    logging: dict = {}
```

PHASE 2: Transport Discovery APIs

(~10 Tage, kann parallel zu Phase 1 entwickelt werden)

Neue Dateistruktur:

```
web/
├── inventory/
│   ├── __init__.py       ← Router zusammenbauen
```

```
│   ├── serial.py          ← Serial Discovery
│   ├── usb.py             ← USB Discovery
│   ├── hid.py             ← HID Discovery
│   ├── network.py         ← TCP Probe + Scan
│   └── visa.py            ← VISA/SCPI Discovery
```

Neue requirements:

```
hidapi          # HID Devices
pyvisa          # VISA Instrumente
pyvisa-py       # Pure Python VISA Backend (kein NI nötig)
bleak           # Bluetooth BLE (optional)
```

2.1 Serial Discovery

Datei: `web/inventory/serial.py` (NEU)
Aufwand: 1 Tag
Priorität: Muss

Endpoints:

- GET `/api/inventory/serial` – Alle Ports mit VID/PID Hints
- POST `/api/inventory/serial/test` – Port öffnen und testen

Wichtige Implementierungshinweise für KI:

- `in_use` Flag: prüfen welche Ports bereits von Engine-Geräten

belegt sind (aus `engine.device_manager.devices` auslesen)

- VID/PID Lookup-Tabelle für bekannte Geräte einbauen
- Plattform-Exceptions sauber abfangen (Windows: Access Denied,

Linux: Permission Denied → mit Hilfetext zurückgeben)

Bekannte VID/PID Mapping (Grundstock):

```
KNOWN_SERIAL_DEVICES = {
    (0x1A86, 0x7523): {"name": "CH340 USB-Serial",
                       "hint": "DPS5005, günstige Arduinos"},
    (0x0403, 0xFA00): {"name": "FTDI FT232",
                       "hint": "Riden RD-Serie"},
    (0x10C4, 0xEA60): {"name": "CP2102 USB-Serial",
                       "hint": "Diverse Netzteile"},
    (0x067B, 0x2303): {"name": "Prolific PL2303",
                       "hint": "Ältere Geräte"},
    (0x0483, 0x5740): {"name": "STM32 Virtual COM",
                       "hint": "STM32-basierte Geräte"},
    (0x2341, 0x0043): {"name": "Arduino Uno",
                       "hint": "Arduino / Custom Firmware"},
```

```
}
```

2.2 USB Device Discovery

Datei: `web/inventory/usb.py` (NEU)
Aufwand: 1 Tag
Priorität: □ Soll

Endpoint: `GET /api/inventory/usb`

Hinweis für KI:

- System-Geräte und Hubs filtern (VID 0x1D6B = Linux Foundation)
- Produktname-Auslesen kann fehlschlagen ohne Admin-Rechte

→ Exception abfangen, nicht crashen

- Device-Klasse in lesbaren Text übersetzen

(0x03 = HID, 0x09 = Hub, etc.)

2.3 USB HID Discovery

Datei: `web/inventory/hid.py` (NEU)
Aufwand: 1 Tag
Priorität: □ Soll

Endpoint: `GET /api/inventory/hid`

Plattform-Hinweise die der Endpoint zurückgeben soll:

```
PLATFORM_HINTS = {
    "linux": "sudo usermod -aG plugdev $USER && udev-Rule anlegen",
    "windows": "Treiber korrekt installiert?",
    "darwin": "System Preferences → Security → Input Monitoring"
}
```

2.4 TCP Probe & Netzwerk-Scan

Datei: `web/inventory/network.py` (NEU)
Aufwand: 2 Tage
Priorität: □ Muss (TCP) / □ Soll (Scan)

Endpoints:

- `GET /api/inventory/tcp/probe?host=x&port=y` – Einzelner Test
- `POST /api/inventory/tcp/scan` – Scan starten (gibt scan_id zurück)
- `GET /api/inventory/tcp/scan/{scan_id}` – Ergebnis pollen

Wichtige Implementierungshinweise für KI:

- Scan läuft als `BackgroundTask` – nicht blockierend!
- Ergebnisse in `scan_results_cache` (dict) halten
- Semaphore(50) um nicht zu aggressiv zu scannen
- Subnetz auto-detektieren via `socket.getsockname()`
- Bekannte Instrument-Ports vorbelegen:

```
KNOWN_INSTRUMENT_PORTS = {
    5025: "SCPI (Riden, Rigol, Keysight Standard)",
    502:  "Modbus TCP",
    1000: "Custom TCP (CH9120 Ethernet-Module)",
    23:   "Telnet SCPI",
    3000: "Custom TCP",
}
```

- Scan-Cache nach X Minuten aufräumen (memory leak vermeiden)
- `GET /api/inventory/tcp/scan` ohne ID → Liste laufender Scans

2.5 VISA Discovery

Datei: `web/inventory/visa.py` (NEU)
Aufwand: 1.5 Tage
Priorität: □ Kann

Endpoint: `GET /api/inventory/visa`

Hinweis für KI:

- `pyvisa.ResourceManager('@py')` – '@py' erzwingt pure Python

Backend, kein NI-VISA nötig

- Timeout auf 2000ms setzen pro Instrument
- IDN-String parsen: “Rigol,DS1054Z,DS1ZA123,v4.04”

→ manufacturer, model, serial, firmware

2.6 Inventory Router

Datei: `web/inventory/init.py` (NEU)

Aufwand: 0.5 Tage
Priorität: Muss

```
# web/inventory/__init__.py
from fastapi import APIRouter
from web.inventory import serial, usb, hid, network, visa

router = APIRouter(prefix="/api/inventory", tags=["Inventory"])
router.include_router(serial.router)
router.include_router(usb.router)
router.include_router(hid.router)
router.include_router(network.router)
router.include_router(visa.router)

# In web/server.py einbinden:
# from web.inventory import router as inventory_router
# app.include_router(inventory_router)
```

PHASE 3: Device Lifecycle (Hot-Reload)

(~8.5 Tage)

3.1 SystemEvent Sample Typ

Datei: `structures/samples/system.py` (NEU)
Aufwand: 0.5 Tage
Priorität: Muss

```
from dataclasses import dataclass, field
from structures.samples.base import BaseSample

@dataclass(kw_only=True)
class SystemEvent(BaseSample):
    """
    System-Ereignisse die über den WebSocket ans Frontend gesendet werden.
    Ermöglicht Live-Updates ohne Page-Reload.
    """
    type: str = "system_event"
    event: str   # Siehe EVENT_TYPES unten
    payload: dict = field(default_factory=dict)

# Definierte Event-Typen:
EVENT_DEVICE_ADDED   = "device_added"
EVENT_DEVICE_REMOVED = "device_removed"
EVENT_DEVICE_UPDATED = "device_updated"
```

```
EVENT_DEVICE_ERROR   = "device_error"
EVENT_SERVER_RESTART = "server_restart"

# Payload-Struktur pro Event:
# device_added:
#   { "meta": {...}, "state": [...] }   ← komplettes Device-Objekt
# device_removed:
#   { "device_id": "DPS5005_1" }
# device_updated:
#   { "device_id": "DPS5005_1", "meta": {...} }
# device_error:
#   { "device_id": "...", "message": "...", "stage": "..." }
```

3.2 EventBus: SystemEvents publishen

Datei: `core/event_bus.py`
Aufwand: 0.5 Tage
Priorität: Muss

```
# In EventBus Klasse hinzufügen:
async def publish_system_event(self, event: str, payload: dict):
    from structures.samples.system import SystemEvent
    sample = SystemEvent(
        device_id="SYSTEM",
        entity_id="system",
        event=event,
        payload=payload
    )
    await self.publish(sample)
```

3.3 store.js: SystemEvents verarbeiten

Datei: `static/js/store.js`
Aufwand: 0.5 Tage
Priorität: Muss

```
// In connectWebSocket() → ws.onmessage erweitern:
ws.onmessage = (e) => {
    window.dispatchEvent(new CustomEvent('ws-message', { detail: e.data }));

    const sample = JSON.parse(e.data);

    // NEU: System-Events abfangen
    if (sample.type === 'system_event') {
```

```
        handleSystemEvent(sample);
        return;  // ← Nicht als normales Sample verarbeiten
    }

    // ... bestehende Messwert-Logik ...
};

function handleSystemEvent(sample) {
    const { event, payload } = sample;

    if (event === 'device_added') {
        globalStore.devices[payload.meta.device_id] = {
            meta: payload.meta,
            state: payload.state
        };
        console.log(`Gerät hinzugefügt: ${payload.meta.device_id}`);
    }
    else if (event === 'device_removed') {
        delete globalStore.devices[payload.device_id];
        console.log(`Gerät entfernt: ${payload.device_id}`);
    }
    else if (event === 'device_updated') {
        if (globalStore.devices[payload.device_id]) {
            globalStore.devices[payload.device_id].meta = payload.meta;
        }
    }
}
```

3.4 DeviceManager: add_device()

Datei: core/device_manager.py
Aufwand: 1.5 Tage
Priorität: Muss

```
async def add_device(self, config: dict) -> dict:
    """
    Fügt ein Gerät live zur Laufzeit hinzu.
    Speichert es in YAML und startet es sofort.
    Gibt das komplette Device-Manifest zurück.
    Wirft DeviceDuplicateError wenn ID bereits existiert.
    Wirft DeviceSpawnError bei Verbindungsproblemen.
    """
    dev_id = config.get('id')

    # Duplikat-Check
    if dev_id in self.devices:
        from core.exceptions import DeviceDuplicateError
        raise DeviceDuplicateError(dev_id)
```

```
    # Gerät spawnen (bestehende Methode)
    self._spawn_device(config)

    device = self.devices.get(dev_id)
    if not device:
        from core.exceptions import DeviceSpawnError
        raise DeviceSpawnError(dev_id, "driver",
                               "Gerät konnte nicht instanziiert werden")

    # Verbinden
    await self._connect_safe(device)

    # In YAML persistieren
    if self.config_manager:
        self.config_manager.add_device(config)

    # Frontend benachrichtigen
    manifest = device.get_manifest()
    state = [{"sensor": e.id, "value": e.value,
              "timestamp": e.timestamp, "alarm_state": "NORMAL"}
             for e in device.entities.values()]

    await self.bus.publish_system_event(
        'device_added',
        {"meta": asdict(manifest), "state": state}
    )

    return asdict(manifest)
```

3.5 DeviceManager: remove_device()

Datei: `core/device_manager.py`
Aufwand: 1.5 Tage
Priorität: Muss

Wichtige Implementierungshinweise für KI:

- Guard-Flag setzen um Race Conditions zu vermeiden
- Tasks in dieser Reihenfolge stoppen:

1. `_measure_task` cancellen und awaiten

```
2. ''transport.disconnect()'' aufrufen
3. Aus ''devices{}'' entfernen
4. StateManager aufräumen
5. ConfigManager aufräumen
6. SystemEvent publishen
```

```python
async def remove_device(self, device_id: str):
    device = self.devices.get(device_id)
    if not device:
        from core.exceptions import DeviceNotFoundError
        raise DeviceNotFoundError(device_id)

    # Guard gegen doppelten Aufruf
    if getattr(device, '_removing', False):
        return
    device._removing = True

    try:
        # 1. Loop stoppen
        if device._measure_task and not device._measure_task.done():
            device._measure_task.cancel()
            try:
                await asyncio.wait_for(
                    asyncio.shield(device._measure_task), timeout=3.0
                )
            except (asyncio.CancelledError, asyncio.TimeoutError):
                pass
            device._measure_task = None

        # 2. Hardware trennen
        try:
            await asyncio.wait_for(
                device.transport.disconnect(), timeout=3.0
            )
        except asyncio.TimeoutError:
            self.logger.warning(f"[{device_id}] Disconnect Timeout")

        # 3. Aus Registry entfernen
        del self.devices[device_id]

        # 4. State aufräumen
        if self.state_manager:
            self.state_manager.remove_device(device_id)

        # 5. Config aufräumen
        if self.config_manager:
            self.config_manager.remove_device(device_id)

        # 6. Frontend informieren
        await self.bus.publish_system_event(
            'device_removed', {"device_id": device_id}
        )

        self.logger.info(f"Gerät '{device_id}' erfolgreich entfernt")

    except Exception as e:
        self.logger.error(f"Fehler beim Entfernen von '{device_id}': {e}")
```

```
        raise
```

3.6 Probe Endpoint

Datei: `web/rest.py`
Aufwand: 2 Tage
Priorität: Muss

Endpoint: `POST /api/devices/probe`

Konzept: Baut einen temporären Stack, instanziiert den Treiber, führt **einen einzelnen** Poll-Zyklus aus, räumt auf und gibt ein Schritt-für-Schritt Protokoll zurück.

Request Body:

```python
class ProbeRequest(BaseModel):
    driver: str            # "DPS5005"
    transport_type: str    # "serial"
    transport_params: dict # {"port": "COM3", "baudrate": 9600}
    framing_params: dict = {}

class ProbeStep(BaseModel):
    message: str
    ok: bool
    detail: str = None

class ProbeResult(BaseModel):
    success: bool
    steps: list[ProbeStep]
    first_values: dict     # {entity_id: {value, unit}}
    error: str = None
    duration_ms: float
```

Wichtige Implementierungshinweise für KI:

- Temporärer `ProbeBus` der Werte in ein dict sammelt

statt sie zu publishen

- **Immer** disconnect im finally-Block aufrufen
- Gesamt-Timeout: 10 Sekunden maximum
- `_find_driver_class()` als Hilfsfunktion auslagern

(wird auch von anderen Endpoints gebraucht)

- Framing aus `STACK_BLUEPRINTS` des Treibers holen

wenn nicht explizit angegeben

3.7 REST Endpoints Device Lifecycle

Datei: `web/rest.py`
Aufwand: 0.5 Tage
Priorität: □ Muss

```
POST   /api/devices              → add_device()
DELETE /api/devices/{id}         → remove_device()
PUT    /api/devices/{id}         → update_device() (remove + add)
PATCH  /api/devices/{id}/enabled → set_device_enabled()
POST   /api/devices/probe        → probe_device()
GET    /api/config/drivers       → get_available_drivers() (existiert)
GET    /api/config/drivers/{name}/blueprint → get_driver_blueprint() (NEU)
```

Fehlerhandling-Muster für alle Lifecycle-Endpoints:

```python
@router.delete("/devices/{device_id}")
async def delete_device(device_id: str, request: Request):
    engine = request.app.state.engine
    try:
        await engine.device_manager.remove_device(device_id)
        return {"status": "ok", "device_id": device_id}
    except DeviceNotFoundError as e:
        raise HTTPException(status_code=404, detail=e.to_dict())
    except Exception as e:
        logger.error(f"Fehler beim Löschen von {device_id}: {e}")
        raise HTTPException(status_code=500,
                            detail={"message": str(e)})
```

PHASE 4: Frontend Infrastruktur

(~3 Tage)

4.1 API Client Modul

Datei: `static/js/api.js` (NEU)
Aufwand: 1 Tag
Priorität: □ Soll (erspart viel Duplikation)

Zweck: Zentraler Wrapper für alle fetch()-Calls. Fehlerhandling an einer Stelle. Alle Endpoints dokumentiert.

```
// static/js/api.js
const BASE = '';  // Gleicher Origin

async function apiFetch(path, options = {}) {
    const res = await fetch(BASE + path, {
        headers: { 'Content-Type': 'application/json', ...options.headers },
        ...options
    });

    if (!res.ok) {
        const err = await res.json().catch(() => ({ message: res.statusText
}));
        throw Object.assign(new Error(err.message || 'API Fehler'), {
            status: res.status,
            code: err.code,
            hint: err.hint,
            detail: err
        });
    }

    return res.json();
}

export const api = {
    // Inventory
    inventory: {
        serial:       () => apiFetch('/api/inventory/serial'),
        serialTest:   (port, baud) => apiFetch('/api/inventory/serial/test',
                        { method: 'POST',
                          body: JSON.stringify({port, baudrate: baud}) }),
        usb:          () => apiFetch('/api/inventory/usb'),
        hid:          () => apiFetch('/api/inventory/hid'),
        tcpProbe:     (host, port) => apiFetch(
`/api/inventory/tcp/probe?host=${host}&port=${port}`),
        tcpScanStart: (subnet, ports) => apiFetch('/api/inventory/tcp/scan',
                        { method: 'POST',
                          body: JSON.stringify({subnet, ports}) }),
        tcpScanPoll:  (scanId) => apiFetch(
                        `/api/inventory/tcp/scan/${scanId}`),
        visa:         () => apiFetch('/api/inventory/visa'),
    },

    // Devices
    devices: {
        list:    () => apiFetch('/api/devices'),
        add:     (conf) => apiFetch('/api/devices',
                   { method: 'POST', body: JSON.stringify(conf) }),
        remove:  (id) => apiFetch(`/api/devices/${id}`,
                   { method: 'DELETE' }),
        update:  (id, conf) => apiFetch(`/api/devices/${id}`,
                   { method: 'PUT', body: JSON.stringify(conf) }),
```

```
        enable:  (id, enabled) => apiFetch(
                   `/api/devices/${id}/enabled`,
                   { method: 'PATCH',
                     body: JSON.stringify({enabled}) }),
        probe:   (conf) => apiFetch('/api/devices/probe',
                   { method: 'POST', body: JSON.stringify(conf) }),
    },

    // Config
    config: {
        drivers:      () => apiFetch('/api/config/drivers'),
        blueprint:    (name) => apiFetch(
                        `/api/config/drivers/${name}/blueprint`),
    },

    // Control (bereits vorhanden, hier zentralisiert)
    control: (device_id, key, value) => apiFetch('/api/control',
                { method: 'POST',
                  body: JSON.stringify({device_id, key, value}) }),
};
```

4.2 index.html: Wizard-Einstiegspunkt

Datei: static/index.html
Aufwand: 0.5 Tage
Priorität: Muss

Änderungen:

```
// Import hinzufügen
import DeviceWizard from '/static/components/device_wizard.js';

// In setup():
const deviceWizardOpen = ref(false);
const deviceWizardEditId = ref(null);  // null = Add-Modus

const openDeviceWizard = (deviceId = null) => {
    deviceWizardEditId.value = deviceId;
    deviceWizardOpen.value = true;
};
```

```
<!-- Im Geräte-Dropdown den toten Handler ersetzen: -->
<div class="dropdown-item" @click="openDeviceWizard()">
    <span><i class="mdi mdi-plus-box-outline"></i>
    Neues Gerät hinzufügen...</span>
</div>

<!-- Pro Gerät Edit-Option hinzufügen: -->
```

```
<div class="dropdown-item" @click.stop="openDeviceWizard(dev.id)">
    <span><i class="mdi mdi-pencil"></i> Bearbeiten...</span>
</div>

<!-- Wizard Modal einbinden: -->
<device-wizard
    v-if="deviceWizardOpen"
    :edit-device-id="deviceWizardEditId"
    @close="deviceWizardOpen = false"
    @saved="deviceWizardOpen = false">
</device-wizard>
```

PHASE 5: Device Wizard UI

(~8.5 Tage – das sichtbare Ergebnis)

Neue Dateistruktur:

```
static/
└── components/
    ├── device_wizard.js          ← Wizard-Container (NEU)
    └── wizard/
        ├── step_driver.js        ← Schritt 1: Treiber wählen
        ├── step_identity.js      ← Schritt 2: Basis-Infos
        ├── step_transport.js     ← Schritt 3: Verbindung
        ├── discovery_panel.js    ← Scan-Helper für Schritt 3
        ├── step_probe.js         ← Schritt 4: Verbindungstest
        └── step_confirm.js       ← Schritt 5: Speichern
```

5.1 device_wizard.js – Rahmen & Navigation

Aufwand: 1 Tag
Priorität: Muss

State der Wizard-Komponente:

```
// Hält die gesamte Konfiguration die durch alle Steps aufgebaut wird
const wizardState = reactive({
    // Step 1
    selectedDriver: null,      // { name, class, description, blueprints }

    // Step 2
    deviceId: '',
    alias: '',
    icon: 'developer-board',
```

```
    description: '',
    location: '',
    autoStart: true,

    // Step 3
    transportType: 'serial',   // aus availableTransports
    transportParams: {},       // port, baudrate ODER host, port
    framingParams: {},         // aus Blueprint übernommen

    // Step 4
    probeResult: null,         // Ergebnis des Verbindungstests

    // Meta
    isEditMode: false,
    originalDeviceId: null
});

const currentStep = ref(1);
const totalSteps = 5;

// Schritt-Validierung: Darf der User weiter?
const canGoNext = computed(() => {
    if (currentStep.value === 1) return !!wizardState.selectedDriver;
    if (currentStep.value === 2) return !!wizardState.deviceId;
    if (currentStep.value === 3) return isTransportConfigValid();
    if (currentStep.value === 4) return wizardState.probeResult?.success;
    return true;
});
```

Wichtige Implementierungshinweise für KI:

- Edit-Modus: Wenn `editDeviceId` prop gesetzt ist, bestehende

Config laden und Steps vorausfüllen

- Step 4 (Probe) automatisch starten wenn Step 3 verlassen wird
- “Überspringen” Option für den Probe-Step (falls Gerät offline

konfiguriert werden soll)

- Gleicher visueller Stil wie WidgetWizard (modal-overlay, etc.)

5.2 step_driver.js – Treiber wählen

Aufwand: 1 Tag
Priorität: Muss

Verhalten:

- Lädt GET `/api/config/drivers` beim Mount
- Zeigt Treiber als Kacheln mit Icon, Name, Beschreibung
- Filter-Tabs nach Kategorie (PSU, Last, BMS, Virtuell, ...)
- Suchfeld (filtert Name + Beschreibung)
- Ausgewählter Treiber wird blau hervorgehoben (wie WidgetWizard)

Hinweis für KI: Der Treiber-Katalog muss `category` und `icon` Felder zurückgeben. Diese kommen aus den Klassen-Attributen (`AbstractDevice` braucht optionale Klassenattribute dafür):

```
# devices/base.py - Klassenattribute ergänzen
class AbstractDevice:
    DRIVER_CATEGORY = "generic"    # "psu", "load", "bms", "virtual"
    DRIVER_ICON = "developer-board"
    # ...

# In konkreten Treibern überschreiben:
class DPS5005(AbstractDevice):
    DRIVER_CATEGORY = "psu"
    DRIVER_ICON = "flash"
```

5.3 step_identity.js – Basis-Infos

Aufwand: 0.5 Tage
Priorität: Muss

Felder:

- Geräte-ID (auto-generiert aus Treiber-Name + Zähler,

z.B. “DPS5005_1”, editierbar)

- ID-Eindeutigkeit live prüfen (gegen `globalStore.devices`)
- Alias (Anzeigename)
- Icon (einfaches Textfeld mit MDI-Vorschau, z.B. “flash”)
- Beschreibung (optional)
- Standort (optional)
- Auto-Start Toggle

ID Auto-Generator:

```
const generateDeviceId = (driverName) => {
    const base = driverName.toUpperCase().replace(/[^A-Z0-9]/g, '_');
    let counter = 1;
    while (globalStore.devices[`${base}_${counter}`]) {
        counter++;
    }
    return `${base}_${counter}`;
};
```

5.4 step_transport.js – Verbindung konfigurieren

Aufwand: 2 Tage
Priorität: □ Muss

Struktur:

- Tab-Leiste mit verfügbaren Transports (aus Blueprint)
- Pro Transport-Typ ein Sub-Formular

Serial Sub-Formular:

```
// Felder:
// - Port (Dropdown, befüllt via api.inventory.serial())
// - Refresh-Button für Port-Liste
// - Baudrate (Dropdown mit häufigen Werten + Custom)
// - Bekanntes Gerät als Hint anzeigen wenn VID/PID matcht
// - "Port testen" Button (öffnet nur, sendet nichts)

// Port-Eintrag Darstellung:
// COM3 - USB Serial (CH340) [DPS5005 vermutlich]  ← grün
// COM7 - Prolific USB-Serial [Belegt]              ← grau
```

TCP Sub-Formular:

```
// Felder:
// - Host (IP-Eingabe mit Validierung)
// - Port (Zahlenfeld, Dropdown mit bekannten Ports)
// - "Netzwerk scannen" Button → öffnet discovery_panel
// - Latenz-Test Button (ping)
```

Erweitert-Sektion (ausgeklappt):

```
// Zeigt framing-Parameter aus Blueprint
// Modbus: Slave ID
// Delimiter: Trennzeichen
// Timeouts
// Diese Werte aus STACK_BLUEPRINTS vorausfüllen!
```

5.5 discovery_panel.js – Scan Helper

Aufwand: 1.5 Tage
Priorität: □ Soll

Serial Discovery Panel:

```
// - Port-Liste mit VID/PID Hints
// - Spalten: Port | Beschreibung | Hersteller | Bekanntes Gerät | Status
// - Klick auf Zeile → übernimmt Port in step_transport
// - Refresh Button
// - Plattform-spezifische Permissions-Warnungen anzeigen
```

TCP Scan Panel:

```
// - Subnetz-Eingabe (auto-vorbelegt)
// - Port-Auswahl (Checkboxen für bekannte Ports)
// - "Scan starten" → POST /api/inventory/tcp/scan
// - Fortschrittsbalken (pollt GET /api/inventory/tcp/scan/{id})
// - Ergebnis-Liste: Host | Port | Service
// - Klick → übernimmt Host+Port in step_transport
// - Polling-Interval: 1 Sekunde
```

5.6 step_probe.js – Verbindungstest

Aufwand: 1.5 Tage
Priorität: Muss (das Herzstück des Wizards!)

Das ist der wertvollste Schritt – User sieht sofort ob es klappt.

Verhalten:

- Startet automatisch wenn der Step angezeigt wird
- Ruft `POST /api/devices/probe` auf
- Zeigt Schritte animiert ein (mit 300ms Verzögerung pro Schritt)
- grün / rot pro Schritt

Darstellung:

```
┌─────────────────────────────────────────────┐
│  Verbindungstest            2.3s     │
├─────────────────────────────────────────────┤
│  Treiber 'DPS5005' geladen              │
│  Stack erstellt: SERIAL via MODBUS      │
│  Port COM3 gefunden                     │
│  Verbindung aufgebaut (9600 Baud)       │
│  Modbus Request gesendet                │
│  Antwort empfangen (31 Bytes)           │
│  CRC korrekt                            │
│  13 Messwerte empfangen                 │
│                                         │
│  ┌─────────────────────────────────┐    │
│  │ Erste Messwerte:                │    │
│  │ u_out:   12.03 V                │    │
│  │ i_out:    2.51 A                │    │
```

```
│ │ p_out:   30.19 W                        │ │
│ └─────────────────────────────────────────┘ │
│                                             │
│ [ ↺ Erneut versuchen ]  [ Weiter → ]        │
└─────────────────────────────────────────────┘
```

Fehler-Darstellung:

```
│ ✓ Treiber geladen                           │
│ ✓ Stack erstellt                            │
│ ✗ Port COM3 nicht gefunden                  │
│    → Prüfe ob das USB-Kabel                 │
│      angeschlossen ist.                     │
│      Verfügbare Ports: COM1, COM7           │
```

Hinweis für KI: Fehlermeldungen vom Backend (`hint`-Feld aus `DeviceSpawnError`) direkt anzeigen. Der User soll nicht raten müssen was falsch ist.

5.7 step_confirm.js – Speichern

Aufwand: 1 Tag
Priorität: ★ Muss

Verhalten:

- Zeigt zusammenfassend alle Einstellungen
- "Gerät hinzufügen" → `POST /api/devices`
- Zeigt Ladeindikator während Request läuft
- Bei Erfolg: kurze Erfolgsanimation, dann Wizard schließen
- Bei Fehler: Fehlermeldung + "Zurück" Option
- Checkbox: "Weiteres Gerät hinzufügen" (schließt Wizard nicht)

Config-Objekt das gebaut wird:

```
const buildConfig = () => ({
    id:          wizardState.deviceId,
    driver:      wizardState.selectedDriver.name,
    enabled:     true,
    auto_start:  wizardState.autoStart,
    alias:       wizardState.alias || wizardState.deviceId,
    icon:        wizardState.icon,
    description: wizardState.description,
    location:    wizardState.location,
    transport: {
        type:   wizardState.transportType,
        params: wizardState.transportParams
    },
    framing: {
```

```
        type:   wizardState.framingType,
        params: wizardState.framingParams
    }
});
```

PHASE 6: Device Management UI

(~2.5 Tage – Nice-to-have nach dem Wizard)

6.1 Geräte-Verwaltungs-Panel

Datei: `static/views/device_manager_view.js` (NEU)
Aufwand: 1.5 Tage
Priorität: □ Soll

- Als Widget spawnbar (neuer Typ im Widget-Wizard)
- Liste aller Geräte mit Status-Indikator
- Edit-Button → Wizard im Edit-Modus
- Enable/Disable Toggle (kein Löschen!)
- Delete-Button mit Bestätigungs-Dialog
- Zeigt ob Gerät gerade verbindet/fehler hat

6.2 Bestätigungs-Dialog Komponente

Datei: `static/components/confirm_dialog.js` (NEU)
Aufwand: 0.5 Tage
Priorität: □ Soll

Generischer Dialog für Delete, Notaus, etc. Wird an mehreren Stellen gebraucht – lohnt sich als Komponente.

```
// Verwendung:
// <confirm-dialog
//   v-if="showConfirm"
//   title="Gerät löschen?"
//   message="DPS5005_1 wird dauerhaft entfernt."
//   confirm-label="Löschen"
//   confirm-color="#ef4444"
//   @confirm="doDelete"
//   @cancel="showConfirm = false">
// </confirm-dialog>
```

6.3 Widget-Wizard: Gerät Manager Eintrag

Datei: `static/components/widget_wizard.js`
Aufwand: 0.5 Tage
Priorität: □ Kann

```
// In widgetCatalogue Array hinzufügen:
{
    id: 'device_manager',
    type: 'device_manager',
    title: 'Geräte-Verwaltung',
    icon: 'devices',
    desc: 'Geräte hinzufügen, bearbeiten und entfernen.',
    needs: 'none'
}
```

Gesamtübersicht Zeitplan

Phase	Inhalt	Tage	Priorität
0	Sofort-Fixes	2-3	□ Muss zuerst
1	Fundament State & Config	7	□ Blocker
2	Transport Discovery APIs	10	□/□ Parallel
3	Device Lifecycle Hot-Reload	8.5	□ Blocker
4	Frontend Infrastruktur	3	□ Vor Wizard
5	Wizard UI	8.5	□ Ziel
6	Device Management UI	2.5	□ Danach
Gesamt		**~42 Tage**	

MVP (nur □ Muss-Tasks): ~28 Tage

Backlog: Wichtige offene Punkte (kein Wizard-Blocker)

Diese Punkte wurden in der Analyse identifiziert, blockieren den Wizard nicht, sollten aber mittelfristig angegangen werden.

B.1 Authentifizierung

Priorität: □ Soll (vor erstem Mehrbenutzerbetrieb)

Aktuell hat ionpy **kein Auth-System**. Für Einzelbetrieb im internen Netz ok. Für Laborumgebung mit mehreren Personen Pflicht.

Empfohlener Ansatz: Einfaches API-Key System

```
# config.yaml Erweiterung:
# auth:
#   enabled: false
#   api_key: "mein-geheimer-key-hier"

# web/auth.py (NEU):
from fastapi import Security, HTTPException
from fastapi.security.api_key import APIKeyHeader

API_KEY_NAME = "X-API-Key"
api_key_header = APIKeyHeader(name=API_KEY_NAME, auto_error=False)

async def verify_api_key(api_key: str = Security(api_key_header)):
    if not AUTH_ENABLED:
        return True  # Auth deaktiviert → alles durch
    if api_key == CONFIGURED_API_KEY:
        return True
    raise HTTPException(status_code=403, detail="Ungültiger API Key")

# Dann in Endpoints:
# @router.post("/control")
# async def send_control(..., _=Depends(verify_api_key)):
```

Frontend:

```
// api.js anpassen:
const API_KEY = localStorage.getItem('ionpy_api_key') || '';
// Header bei jedem Request mitschicken
headers: { 'X-API-Key': API_KEY, ... }
```

Mittelfristig: JWT-Token mit Login-Seite.

B.2 Path-Traversal Fix Workspaces

Priorität: Muss (Sicherheitslücke!)

```
# AKTUELL - unsicher:
@router.get("/workspaces/{name}")
def api_load_workspace(name: str):
    filepath = os.path.join(WORKSPACE_DIR, f"{name}.json")
    # name = "../../config" würde config.yaml lesen!

# FIX:
```

```python
import pathlib

@router.get("/workspaces/{name}")
def api_load_workspace(name: str):
    # Nur alphanumerisch + Bindestriche erlauben
    if not name.replace('-', '').replace('_', '').isalnum():
        raise HTTPException(400, "Ungültiger Workspace-Name")

    base = pathlib.Path(WORKSPACE_DIR).resolve()
    target = (base / f"{name}.json").resolve()

    # Sicherstellen dass target wirklich in base liegt
    if not str(target).startswith(str(base)):
        raise HTTPException(400, "Ungültiger Pfad")

    if not target.exists():
        raise HTTPException(404, "Workspace nicht gefunden")

    return json.loads(target.read_text(encoding='utf-8'))
```

B.3 Alarm-System Ende-zu-Ende

Priorität: □ Soll

Backend erkennt Alarme bereits (HIHI/HI/LO/LOLO in `NumericEntity._evaluate_alarms()`), aber:

- Keine Alarm-Übersicht im UI
- Kein visuelles/akustisches Signal bei neuem Alarm
- Keine Alarm-Quittierung
- Keine Alarm-Historie

Vorgeschlagenes Vorgehen:

1. Alarm-Event über SystemEvent-Mechanismus publishen
2. Alarm-Badge im Header (Zahl aktiver Alarme)
3. Alarm-Panel Widget (Liste aller aktiven + historischen Alarme)
4. Optional: Browser Notification API für aktive Tabs

B.4 Graceful Shutdown

Priorität: □ Soll

```python
# core/engine.py - stop() erweitern:
async def stop(self):
```

```
    self._running = False
    logger.info("Engine fährt herunter...")

    # 1. State sichern
    await self.state_manager.save_now()

    # 2. Alle Geräte sauber trennen
    disconnect_tasks = []
    for dev in self.device_manager.devices.values():
        disconnect_tasks.append(self._disconnect_safe(dev))

    if disconnect_tasks:
        await asyncio.gather(*disconnect_tasks, return_exceptions=True)

    logger.info("Alle Geräte getrennt. Engine gestoppt.")

async def _disconnect_safe(self, device):
    try:
        await asyncio.wait_for(device._do_disconnect(), timeout=5.0)
    except Exception as e:
        logger.warning(f"[{device.device_id}] Disconnect Fehler: {e}")
```

B.5 Audit-Trail für Steuerbefehle

Priorität: Kann

Wer hat wann welchen Befehl gesendet? Wichtig sobald mehrere Personen das System bedienen.

```
# core/audit_logger.py (NEU) - einfaches CSV/JSON Log:
# Timestamp | User/IP | Device | Entity | OldValue | NewValue

# In web/rest.py bei /control:
# audit_logger.log(request.client.host, cmd.device_id,
#                  cmd.key, cmd.value)
```

B.6 Unit Tests

Priorität: Soll

```
# tests/conftest.py
import pytest
from core.engine import SystemEngine

@pytest.fixture
async def engine():
```

```
    eng = SystemEngine(config_path="tests/test_config.yaml")
    eng.start()
    await eng.run_hardware()
    yield eng
    await eng.stop()

# tests/test_config.yaml - nur virtuelle Geräte:
# devices:
#   - id: V01_Test
#     driver: V01_Battery
#     transport: {type: virtual}
#     framing: {type: none}

# tests/test_device_lifecycle.py
async def test_add_remove_device(engine):
    conf = {"id": "TEST_DEVICE", "driver": "V01_Battery", ...}
    await engine.device_manager.add_device(conf)
    assert "TEST_DEVICE" in engine.device_manager.devices

    await engine.device_manager.remove_device("TEST_DEVICE")
    assert "TEST_DEVICE" not in engine.device_manager.devices
```

B.7 Mobile App (PWA)

Priorität: □ Kann

Separate Einstiegsseite `/mobile` mit:

- Bottom-Tab-Navigation
- Geräteübersicht mit Key-Metrics
- Live-Werte pro Gerät
- NOTAUS Button immer sichtbar
- Alarm-Anzeige
- PWA Manifest für Homescreen-Installation

Wiederverwendet: `store.js`, WebSocket, `uPlot`, alle Vue Utilities.
Neue Datei: `static/mobile.html` + `static/mobile/` Verzeichnis.

B.8 Datenbank-Anbindung für Historien-Charts

Priorität: □ Soll (Chart-Historie ist als “bald” markiert)

Optionen:

- **SQLite** – kein Setup, gut für Einzelbetrieb

- **InfluxDB** – optimiert für Zeitreihen, skalierbar
- **TimescaleDB** – PostgreSQL-Extension, guter Mittelweg

Empfehlung für ionpy: SQLite als erstes, InfluxDB als Option.

```
# Minimale SQLite Integration:
# - Neuer Subscriber auf EventBus
# - Schreibt NumericSamples in SQLite
# - REST Endpoint: GET /api/history/{device}/{entity}?from=&to=
# - Chart-View schaltet von "Live-Puffer" auf "DB-Abfrage" um
```

B.9 Config YAML Kommentare erhalten

Priorität: Kann

Standard `yaml.dump()` löscht alle Kommentare. Falls der User seine config.yaml manuell kommentiert hat, gehen diese beim ersten Wizard-Speichern verloren.

Fix: `ruamel.yaml` statt `pyyaml` für das Schreiben verwenden.

install ruamel.yaml

```

```

```
from ruamel.yaml import YAML
yaml = YAML()
yaml.preserve_quotes = True
# Dann yaml.dump(data, file) statt yaml.dump(data, file)
```

B.10 Mehrsprachigkeit (i18n)

Priorität: Kann

Aktuell komplett Deutsch hardcoded. Für internationale Nutzung oder gemischte Teams relevant.

Vue 3 i18n Plugin + JSON-Sprachdateien.
Aufwand: ~5 Tage initial, danach Übersetzungen hinzufügen.

B.11 Error Boundaries im Frontend

Priorität: Soll

Wenn eine Vue-Komponente wirft, reißt sie aktuell das gesamte Widget mit. `errorCaptured`-Hook fehlt.

```
// In spawn() nach dem createApp():
vueApp.config.errorHandler = (err, instance, info) => {
    console.error(`Widget Fehler [${wid}]:`, err, info);
    // Widget mit Fehlermeldung ersetzen statt alles zu crashen
    const body = document.getElementById(wid);
    if (body) body.innerHTML = `
        <div style="padding: 20px; color: #ef4444; font-size: 12px;">
            <i class="mdi mdi-alert-circle"></i> Widget Fehler<br>
            <small style="color: #71717a;">${err.message}</small>
        </div>
    `;
};
```

B.12 Dokumentation

Priorität: ☐ Soll

- Treiber-Entwicklerdoku: Wie schreibt man einen neuen Treiber?
- API-Doku: FastAPI `'/docs'` ist bereits vorhanden (Swagger UI)
- Deployment-Guide: Systemd-Service, Autostart
- config.yaml Referenz mit allen Optionen

Ende der Roadmap
Erstellt im Kontext der ionpy Pro OS Implementierungsdiskussion
Alle Code-Beispiele sind Implementierungshinweise, keine fertigen Lösungen