Shoot A Pi!
with Eclipse Kura

David Woodard @ Eurotech
Luca Dazi @ Eurotech
Agenda

- Presentation of Kura architecture for Java and OSGi based multi service gateway platforms. (Dave, 20 mins)
- Presentation of the Shoot-A-Pi arcade game simulator, game logic explanation, MQTT topics and metrics (Luca, 15 mins)
- Hardware setup on the Raspberry Pi B+ (15 mins)
- Assisted creation of the Shoot-A-Pi bundle (90 mins)
- Tests (15 mins)
- Dashboard showcase and final game (15 mins)
- Q&A
Before starting...

- Power on the Raspberry Pi with the micro USB cable
- Connect the Raspberry Pi to an ethernet port on your PC
- Connect your PC to the WiFi network named ‘ShootAPi’
  Password: *KuraTutorial*
- Set the IP Address of your ethernet to 192.168.2.1
- Start VirtualBox and import the tutorial image (*shootapi.ova*)
- Start the newly imported EclipseCon VM
- Start the Terminal Emulator
- Access the Raspberry Pi with ssh at address 192.168.2.10
  `ssh pi@192.168.2.10`
  Password: *raspberry*
Share your WiFi with the Pi

**Linux / Mac users**
- Open the Kura Web Console on a browser (192.168.2.10)
- Navigate to the Network panel and set eth0 on DHCP
- Share your WiFi with the Ethernet interface
- Do an ifconfig on the terminal and take note of the IP address assigned to eth0 (defaults to 10.42.0.1 on Ubuntu)
- Scan for the IP of the Pi using nmap
  
  ```
  nmap 10.42.0.0-255
  ```

**Windows users**
- Share your WiFi with the Ethernet interface
- Set the IP address of the Ethernet interface to 192.168.2.1
- The Raspberry Pi will be available at address 192.168.2.10
Share your WiFi with the Pi

Windows

- Right-click on the Network icon in the taskbar
- Open Network and Sharing Center
- Click on Change adapter settings
- Right-click on WiFi, open the Properties
- Activate the Sharing tab
- Check the Allow other network users to connect checkbox
- Select Ethernet from the combo box below
- Apply and close
Share your WiFi with the Pi

**Ubuntu**

- Click on the network icon, then Edit Connections...
- Double click on the Wireless Connection
- Open the IPv4 Settings tab
- Select Shared with other computers on the Method combo box
- Apply and exit. Once the Pi is connected and set to DHCP click on the Wired Network so to renew the DHCP lease
IoT Gateways
Revolution: Towards Real-time Actionable Data

MQTT Broker

Business Application
KURA is the open source Java and OSGi-based Application Framework for M2M Service Gateways in the Eclipse IOT Working Group.

**Purpose**
Simplify the design, deployment and remote management of embedded applications.

**It provides**
- Cohesive and integrated app environment
- Modular software components
- HW abstraction layer
- Field protocol libraries
- Cloud connectivity
- Remote app and device management
- Local app and device management
- Built-in Security
- Development tools
Encapsulating complexity
Increase productivity and decrease cultural barriers

Kura helps customer focusing on their core business
## Kura Developers’ Experience

Designed from ground-up for developers

<table>
<thead>
<tr>
<th>Emulate on PC</th>
<th>Deploy on Target</th>
<th>Cloud Managed</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Emulate on PC Icon" /></td>
<td><img src="image" alt="Deploy on Target Icon" /></td>
<td><img src="image" alt="Cloud Managed Icon" /></td>
</tr>
<tr>
<td>Start developing your M2M application in the comfort of your PC.</td>
<td>When you are ready, deploy your application on the gateway.</td>
<td>Provision your application to field devices from the Cloud.</td>
</tr>
<tr>
<td>• Full Eclipse Integration</td>
<td>• One-click Deployment</td>
<td>Manage your application configuration and lifecycle from a Cloud infrastructure. No more field visits!</td>
</tr>
<tr>
<td>• Target Platform Definition</td>
<td>• Eclipse Plugin</td>
<td>• Web-based Console</td>
</tr>
<tr>
<td>• Emulated Services</td>
<td>• Remote Debugging</td>
<td>• REST API Integration</td>
</tr>
<tr>
<td>• Run/Debug from Eclipse</td>
<td></td>
<td>• Smart Alerts</td>
</tr>
<tr>
<td>• Support Mac/Linux Hosts</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Eclipse Open IoT Stack for Java

OSGi Application Container (Eclipse Equinox, Concierge)

Java SE 7 / 8 (OpenJDK)
Shoot-A-Pi Arcade Shooter Simulator

Architecture

- Eclipse Equinox OSGi
- Java VM
- Linux
- Hardware

Hardware

Device/Gateway (data collection)

MQTT Broker

- MQTT
- Websockets
- REST APIs
- Web Dashboards

GPIO

I2C

Laser

RF

Human Interface Device
Shoot-A-Pi Arcade Shooter Simulator

MQTT Topics and Metrics

**COMMAND/**
- reset
  - Timestamp
  - Metrics: Game ID
- new
- stop
  - Timestamp
  - Metrics: Game ID

**shootapi/**

**DATA/**
- shot
- score
- reload
- reloading
  - Timestamp
  - Metrics: Realoding status
  - Timestamp
  - Metrics: Realod flag
  - Timestamp
  - Metrics: Remaining rounds
  - Timestamp
  - Metrics: Current score
Shoot-A-Pi Arcade Shooter Simulator

**Game Logic**

- **HID Worker**
- **I2C Worker**
- **GPIO Actuator**
- **Game Logic**

The diagram illustrates the flow of information and actions within the Shoot-A-Pi system:

- **HID Worker** communicates with the system and triggers the start of an event.
- **I2C Worker** processes the initial input and sends the signal to the **GPIO Actuator**.
- **GPIO Actuator** responds to the signal and initiates a physical action, simulating a shot.
- The **Game Logic** module receives the shot event and updates the score accordingly, indicating the progress and outcome of the shoot.

The system operates with a 250 ms delay for each step, ensuring smooth and responsive gameplay.
Shoot-A-Pi Arcade Shooter Simulator

Hardware Setup

I2C

Raspberry Pi B+ J8 Header

GPIO

SDA1 - I2C

SCL1 - I2C

3.3v DC Power

5v DC Power

5v DC Power

Ground

Ground

GPIO17

GPIO18

Ground

RF Dongle
Setting up the Laser Tag

The LP-170 Laser Tag has two working modes.

**Mode A**

<table>
<thead>
<tr>
<th>Pen Function</th>
<th>Shoot A Pi Function</th>
<th>Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Mode Switch</td>
<td>-</td>
<td>NONE</td>
</tr>
<tr>
<td>B Page Down</td>
<td>None</td>
<td>0x00004e</td>
</tr>
<tr>
<td>C Page Up</td>
<td>None</td>
<td>0x00004b</td>
</tr>
</tbody>
</table>

**Mode B**

<table>
<thead>
<tr>
<th>Pen Function</th>
<th>Shoot A Pi Function</th>
<th>Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Mode Switch</td>
<td>-</td>
<td>NONE</td>
</tr>
<tr>
<td>B Start Slideshow</td>
<td>Reload</td>
<td>0x02003e</td>
</tr>
<tr>
<td>B Stop Slideshow</td>
<td>Reload</td>
<td>0x000029</td>
</tr>
<tr>
<td>C Hide Slideshow</td>
<td>Shoot</td>
<td>0x000005</td>
</tr>
</tbody>
</table>

A. Mode Switching  
B. Reload  
C. Fire  
D. Only laser  
E. RF USB Dongle
We will use a helper class to trace the commands received from the pen.

```java
public static enum PenCommand{
    CMD_LASER_NO_REPEAT("000005", "Shooting Laser!") ,
    CMD_LASER_REPEAT("00004b", "Scrolling down..."),
    CMD_RELOAD_REPEAT("00004e", "Scrolling up..."),
    CMD_RELOAD_NO_REPEAT_A("02003e", "Reload 1"),
    CMD_RELOAD_NO_REPEAT_B("000029", "Reload 2"),
    CMD_UNKNOWN_COMMAND("000000", "Unknown command");
}
```

Stub file: *PenCommandsEnum.stub.java*
Setting up the Laser Tag Worker

Reading from the pen will be handled by a Runnable class, which will constantly poll the pen for input using HID APIs provided by Kura. A listener is passed in the constructor, so that when the worker detects a command, it can wake up the caller.

```java
public class LaserPenWorker implements Runnable {

    private static final int PEN_VENDOR_ID = 4643;
    private static final int PEN_PRODUCT_ID = 16230;

    private static HIDDevice thePen = null;
    private static PenDetectListener callback;

    public LaserPenWorker(PenDetectListener callback) {
        LaserPenWorker.callback = callback;
    }

    public void run() {
        byte[] data = new byte[3];
        try {
            if (null == thePen) {
                thePen = HIDManager.getInstance().openById(
                        PEN_VENDOR_ID,
                        PEN_PRODUCT_ID, null);
            }
            while (true) {
                thePen.read(data);
                // Convert data to string and put in result
                if (!result.toString().isEmpty() && !result.toString().equals("000000") ) {
                    fireChangeEvent(result.toString());
                }
            }
        } catch (HIDDeviceNotFoundException ex) {
        } catch (IOException ex) {
        } finally {
    }
```
Deploying the Bundle

mToolkit

- Export the bundle using *Export -> Plug-in development -> Deployable plug-in and fragments*

- *Open the mToolkit Frameworks view using Window -> Show View -> Others...*

- Activate the *Frameworks* tab and create a new Framework using the IP Address of the Pi

- Start the newly created framework

- Right-click on Bundles

- Click on «Install new...» and select the plug-in you exported before

- Connect to the Pi and see the Bundle in action!
After accessing the Pi through ssh you will be able to inspect the log files and control Kura using these commands:

- `tail -f /var/log/kura.log` will show the realtime kura log.
- `tail -f /var/log/kura-console.log` will show the System.err log.
- `telnet 127.0.0.1 5002` will open the OSGi telnet terminal.
- `sudo /etc/init.d/kura restart` will restart Kura. Bundles installed with mToolkit will be removed.
Setting up the Digital Light Sensor
Enable I2C on the Raspberry Pi

The Raspberry Pi ships with the I2C disabled.
In order to communicate with the Grove Digital Light Sensor we have to enable the Linux modules that will enable I2C communication on the Pi.

Enter the following commands in the Pi command line:

```
sudo nano /etc/modules
```

And add these two lines to the file:

```
i2c-bcm2708
i2c-dev
```

Then save the file and reboot the Pi
Setting up the Digital Light Sensor

Worker overview

- Detecting luminosity changes will be demanded to a separate Runnable
- The I2C Digital Light Sensor is acquired and managed using OpenJDK Device I/O APIs, provided by Kura
- The worker will be constantly polling the Light Sensor reading the luminosity and will trigger listeners when it needs to.
- The change in luminosity between polls is evaluated using several thresholds, programmable through the Kura Web UI.
- The LUX value is calculated using an helper method, provided in the stub.

Stub file: LuxCalculation.stub.java
Setting up the Digital Light Sensor

Managing I2C

I2C Devices are accessed using `jdk.dio.I2CDevice` and `jdk.dio.I2CDeviceConfig` classes.

Reads and writes on the sensor can be atomic or transacted.

```java
private static void initDevice() {
    try {
        I2CDeviceConfig config = new I2CDeviceConfig(
            1,
            LIGHT_SENSOR_ADDRESS,
            1,
            400000
        );
        s_light_sensor = (I2CDevice) DeviceManager.open(I2CDevice.class, config);
        // INIT
        s_light_sensor.begin();
        s_light_sensor.write(0x80); s_light_sensor.write(0x03);
        s_light_sensor.write(0x81); s_light_sensor.write(0x11);
        s_light_sensor.write(0x86); s_light_sensor.write(0x00);
        s_light_sensor.end();
    } catch (UnavailableDeviceException e) {
    } catch (DeviceNotFoundException e) {
    } catch (ClosedDeviceException e) {
    } catch (IOException ex) {
    }
}
```

Refer to OpenJDK Device I/O APIs for further info
Setting up the Digital Light Sensor Worker

Another Runnable is used to implement the Digital Light Sensor logic

```java
public class DigitalLightSensorWorker implements Runnable {
    public void run() {
        if (null == s_light_sensor || !s_light_sensor.isOpen()) { initDevice(); }
        try {
            while (true) {
                s_light_sensor.write(0x8C); Thread.sleep(5);
                L0 = s_light_sensor.read(); Thread.sleep(5);
                s_light_sensor.write(0x8D); Thread.sleep(5);
                H0 = s_light_sensor.read(); Thread.sleep(5);
                s_light_sensor.write(0x8E); Thread.sleep(5);
                L1 = s_light_sensor.read(); Thread.sleep(5);
                s_light_sensor.write(0x8F); Thread.sleep(5);
                H1 = s_light_sensor.read();

                int ch0 = (((H0 & 0xff) * 0x100) + L0) & 0xffff;
                int ch1 = (((H1 & 0xff) * 0x100) + L1) & 0xffff;
                int lux = Utilities.calculateLux(ch0, ch1);

                if (lux > PROP_THRESHOLD_LUX_MAX) {
                    fireChange(lux);
                }
                Thread.sleep(READ_RESOLUTION);
            }
        } catch (IOException ex) {
        } catch (InterruptedException ex) {
        } finally {
            closeDevice();
        }
    }
}
```

Stub file: `LightSensorWorker.stub.java`
Setting up the Digital Light Sensor

Wake-up / Sleep logic

A simple wake-up / sleep logic is implemented in the worker in order to have it fire lux change events only when needed.

```java
public class DigitalLightSensorWorker implements Runnable {

private static LightSensorChangeListener callback;
private static boolean s_listen = false;

public DigitalLightSensorWorker(LightSensorChangeListener callback) {
    DigitalLightSensorWorker.callback = callback;
}

public static void startListeningForLaser() { s_listen = true; }
public static void stopListeningForLaser() { s_listen = false; }
public static boolean isAcquiring() { return s_listen; }

private void fireChange(int lux) {
    if (s_listen) {
        callback.lightSensorChangeDetected(lux);
        stopListeningForLaser();
    }
}
}
```

The `startListeningForLaser()` method is called when the Laser Tag Worker detects a Shot command.

Stub file: `LightSensorWorker.stub.java`
The GPIO Actuator is yet another runnalbe. This time it is a simple class delegated to work on the GPIOs using jdk.dio.GPIOPin. In this class Device I/O features are loaded using the default configuration.

```java
public class GPIOActuator implements Runnable {
    private static final int ledPinGPIO = 17;
    private static GPIOPin led;

    public GPIOActuator() {
        try {
            Device<?> d = DeviceManager.open(ledPinGPIO);
            led = (GPIOPin) d;
            led.setValue(false);
        } catch (IOException e) {
        }
        try {
        }
        public static void closeGPIOs() {
            try{
                led.close();
            } catch(IOException ex) {
            }
        }
    }

    public void run() {
        try {
            led.setValue(true); Thread.sleep(1000); led.setValue(false);
        } catch (IOException e) {
        } catch (InterruptedException e) {
        } catch (IOException e) {
        }
    }
}
```

Stub file: `GPIOActuator.stub.java`
Game Logic

Overview

- When the game starts, player must be set to 0 scored points and must have a programmable amount of rounds (default 12)

- When the player fires a round (C button) the game starts listening for lux change on the DLS for a programmable time window (default 200ms)
  - Available rounds are decreased by 1. If the lux change is detected in the time frame, 1 point is scored, led and buzzer get activated
  - Lux variance threshold is programmable. Defaults to 300lux

- Once the clip is empty the player should reload the gun (B button). Reload will take a programmable amount of time (default 5s) during which no point can be scored.

- Game should subscribe to a Commands topic, listening for «NewGame» and «StopGame» commands.
  - When receiving a «NewGame» it should reset score and available rounds
  - When receiving a «StopGame» it should stop scoring points until a «NewGame» is received
Game Logic
Implementation

public class GameLogic {

private static int s_score;
private static int s_clip;
private static boolean s_reloading = false;
private static boolean s_game_stopped = false;

public static void startGame() {
    s_game_stopped = false;
    s_score = 0;
    s_clip = PROP_CLIP_SIZE;
}

public static void stopGame() {
    s_game_stopped = true;
}

public static void shoot() {
    if (s_game_stopped) { return; }
    if (isReloading()) { return; }
    if (s_clip == 0) {
        ShootApi.doPublish("NeedsReload", true);
    } else {
        s_clip--;
        ShootApi.doPublish("Shot!", s_clip);
    }
}

public static boolean isReloading() {
    return s_reloading;
}

public static void scorePoint() {
    if (s_game_stopped) { return; }
    if (s_clip > 0) {
        s_score++;
        ShootApi.doPublish("Score", s_score);
    }
}

public static void reload() {
    if (s_game_stopped) { return; }
    s_clip = PROP_CLIP_SIZE;
    Thread reloadThread = new Thread(
            new Runnable() {
                public void run() {
                    try {
                        ShootApi.doPublish(
                                "Reloading", true);
                        s_reloading = true;
                        Thread.sleep(PROP_RELOAD_DELAY);
                        ShootApi.doPublish(
                                "Reloading", false);
                    } catch (InterruptedException ex) {
                    } finally {
                        s_reloading = false;
                    }
                }
            });
    if (!s_reloading) {
        reloadThread.start();
    }
}

Stub file: GameLogic.stub.java
Shoot A Pi

Main class overview

The ShootAPi class is responsible for managing the whole application

- Implements ConfigurableComponent
  - It exposes a component in the Kura Web UI, letting the user change configuration parameters from any browser

- Acquires the CloudService
  - Publishes data to the MQTT Broker using the MQTTDataTransport

- Implements CloudClientListener
  - Listens for requests on the Commands MQTT topic

- Manages the Executors
  - It starts, stops and cancels the runnables and wires everything together
Shoot A Pi
ConfigurableComponent and OSGi Component configuration

The ConfigurableComponent interface provides no methods. It will instead make the class appear as a Web UI component. The class will also exported as a OSGi Declarative Service

```java
public class ShootAPi implements ConfigurableComponent, CloudClientListener {
    private static final String APP_ID = "Shoot_A_Pi_Demo"; // Cloud App identifier
    // Publishing Property Names
    private static final String PUBLISH_TOPIC_PROP_NAME = "publish.appTopic";
    private static final String PUBLISH_QOS_PROP_NAME = "publish.qos";
    private static final String PUBLISH_RETAIN_PROP_NAME = "publishretain";
    // Configurable Properties Names
    private static final String PROP_CLIP_SIZE = "clip.size";
    private static final String PROP_DETECTION_WINDOW = "dls.detect.window";
    private static final String PROP_DETECTION_THRESHOLD = "dls.detect.threshold";
    private static final String PROP_DETECTION_THRESHOLD = "dls.detect.threshold";
    private static final String PROP_DETECTION_THRESHOLD = "dls.detect.threshold";
    private static final String PROP_DETECTION_THRESHOLD = "dls.detect.threshold";
    Map<String, Object> m_properties;

    public void updated(Map<String, Object> properties) {
        // store the properties received
        m_properties = properties;
        for (String s : properties.keySet()) {
            s_logger.info("Update - " + s + " : " + properties.get(s));
        }
        // try to kick off a new job
        doUpdate();
    }
}
```

Stub file: Main.stub.java
The CloudService will be used to publish data to the Broker, while the CloudClientListener will listen for MQTT messages on the «Commands» topic.

```java
public class ShootAPi implements ConfigurableComponent, CloudClientListener {
    private CloudService m_cloudService;
    private static CloudClient m_cloudClient;

    public void setCloudService(CloudService cloudService) {
        m_cloudService = cloudService;
    }

    public void unsetCloudService(CloudService cloudService) {
        m_cloudService = null;
    }

    protected void activate(ComponentContext componentContext,
                            Map<String, Object> properties) {
        try {
            m_cloudClient = m_cloudService.newCloudClient(APPLICATION_ID);
            m_cloudClient.addCloudClientListener(this);
            doUpdate();
        } catch (Exception e) {
        }
    }

    public void onConnectionEstablished() {
        try {
            m_cloudClient.subscribe("Commands/#", 0);
        } catch (KuraException ex) {
        }
    }

    public void onMessageArrived(String deviceId,
                                  String appTopic,
                                  KuraPayload msg, int qos, boolean retain) {
        Object command = msg.getMetric("Command");
        if (command != null) {
            switch (command.toString()) {
                case "NewGame":
                    GameLogic.startGame();
                    break;
                case "StopGame":
                    GameLogic.stopGame();
                    break;
                default:
                    break;
            }
        }
    }
}
```

Stub file: Main.stub.java
Executors are used to start the Runnables.

```java
public class ShootAPi implements ConfigurableComponent, CloudClientListener {
    // Executors
    private static ScheduledExecutorService s_pen_poller;
    private static ScheduledExecutorService s_light_sensor;
    private static ExecutorService s_activator;
    // Handles
    private static ScheduledFuture<?> s_pen_handle;
    private static ScheduledFuture<?> s_sensor_handle;
    private static Future<?> s_activator_handle;

    public ShootAPi() {
        s_pen_poller = Executors.newSingleThreadScheduledExecutor();
        s_light_sensor = Executors.newSingleThreadScheduledExecutor();
        s_activator = Executors.newSingleThreadExecutor();
    }

    protected void deactivate(ComponentContext componentContext) {
        s_activator.shutdown();
        s_light_sensor.shutdown();
        s_pen_poller.shutdown();
    }

    private void doUpdate() {
        // cancel a current worker handle
        if (s_pen_handle != null) {
            s_pen_handle.cancel(true);
        }
        if (s_activator_handle != null) {
            s_activator_handle.cancel(true);
        }
        if (s_sensor_handle != null) {
            s_sensor_handle.cancel(true);
        }
        penWorkerRunnable = new LaserPenWorker(penButtonPressed);
        s_pen_handle =
            s_pen_poller.scheduleWithFixedDelay(
                penWorkerRunnable,
                1, 2, TimeUnit.SECONDS);

        sensorWorkerRunnable =
            new DigitalLightSensorWorker(laserDetected);
        s_sensor_handle =
            s_light_sensor.scheduleWithFixedDelay(
                sensorWorkerRunnable,
                1, 2, TimeUnit.SECONDS);

        GameLogic.startGame();
    }
}
```

Stub file: `Main.stub.java`
Shoot-A-Pi Arcade Shooter Simulator
Web Dashboard Architecture

 COMPLETE DASHBOARD IN THE Dashboard FOLDER
You are important!
Kura helps you ... Kura needs you

I was lucky to be involved and get to contribute to something that was important, which is empowering people with software. (By Bill Gates)
Evaluate the sessions

Sign in: www.eclipsecon.org