 Shoot A Pi! with Eclipse Kura

David Woodard @ Eurotech
Walt Bowers @ Eurotech
Agenda

- Presentation of Kura architecture for Java and OSGi based multi service gateway platforms. (Walt, 15 mins)
- Presentation of the Shoot-A-Pi arcade game simulator, game logic explanation, MQTT topics and metrics (David, 10 mins)
- Emulator
  - Creation of an empty OSGi Bundle (5 min)
  - Bundle lifecycle management (component.xml) (10 min)
  - Use of services and the OSGi folder (15 min)
  - OSGi Metatype definition and programmable properties (10 min)
  - Kura Cloud Service (15 min)
  - Executors and multi-threading (10 min)
  - Light-sensor runnable (15 min)
  - GPIO Actuator (10 min)
  - Game logic implementation (10 min)
  - Debug in Emulator (10 min)
- Pause (5 min)
- Remote Debug
  - Start Kura in debug mode (5 min)
  - Deploy using mToolkit (5 min)
  - Remote debugging session (5 min)
- Deploy to Target
  - Creation of the Deployment Package Project (5 min)
  - Creation of the Deployment Package (5 min)
  - Deploy to Target using the Kura WebUI (5 min)
  - Example of Remote Deployment through EDC (10 min)
- Dashboard Demo (10 min)
Gateways per IoT
Towards Real Time, Actionable Data
Encapsulating Complexity

Increase Productivity

Kura helps customers focus on their core business

Developer’s Productivity

Code

Code

OSGi

Java VM

Operating System

Hardware
**Esperienza di Sviluppo con Kura**

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" /></td>
<td><img src="image" alt="Deploy on Target" /></td>
<td><img src="image" alt="Cloud Managed" /></td>
</tr>
</tbody>
</table>

### Emulate on PC
- Start developing your IoT/M2M application in the comfort of your PC.
- Full Eclipse Integration
- Target Platform Definition
- Emulated Services
- Run/Debug from Eclipse
- Support Mac/Linux Hosts

### Deploy on Target
- When you are ready, deploy your application on the gateway.
- One-click Deployment
- Eclipse Plugin
- Remote Debugging

### Cloud Managed
- Provision and manage your applications in field devices from the Cloud.
- Remote OSGi Management via MQTT
- Web-based Console
Shoot-A-Pi Arcade Shooter Simulator

Architecture

- Eclipse Equinox OSGi
- Java VM
- Linux
- Hardware

Device/Gateway (data collection)

- MQTT
- Websockets REST APIs
- Web Dashboards

- I2C
- GPIO
- Laser
Shoot-A-Pi Arcade Shooter Simulator

MQTT Topics and Metric

- [account-id]/[device-id]/[application-defined-topic]
  - Shot!
  - Timestamp
- [account-id]/[device-id]/[application-defined-topic]
  - Miss!
  - Consecutive miss count
- [account-id]/[device-id]/[application-defined-topic]
  - Reset!
  - String
- [account-id]/[device-id]/[application-defined-topic]
  - Score!
  - Current score
Shoot-A-Pi Arcade Shooter Simulator

Game Logic

Shoot-A-Pi

HID Worker

I2C Worker

GPIO Actuator

Game Logic

start acquiring

1000ms

score

200ms
Developing the Game in Eclipse IDE

Start the Virtual Machine, start Eclipse, and start developing the Shoot-A-Pi
Class Diagram

- Shoot-A-Pi
- Utilities
- DigitalLightSensorWorker
  - Runnable
- GPIOActuator
- GameLogic
- Randomizer
  - Runnable
Creating a New OSGi Bundle

- New -> Plug-in Development -> Plug-in Project
- Name it with the fully qualified namespace
- OSGi Framework -> standard
- Give a name (Better equal to ID)
- Do not generate an Activator

- Project Properties -> Java Build Path -> Source
  - Add a main/java folder under src
  - Set src/main/java as Java Source Folder
  - Remove src from Java Source Folder
- Add class to package in src/main/java folder
Bundle Lifecycle Management

Part 1

- New -> Plug-in Development -> Component Definition
- Place it in OSGI-INF folder
- Name it (better use class name)
- Open source in editor and add:
  - activate="activate"
  - deactivate="deactivate"
  - modified="updated"
  - enabled="true"
  - immediate="true"
Bundle Lifecycle Management

Part 2

- Edit class file and add Lifecycle Hooks:
  - Add method `protected void activate(ComponentContext componentContext, Map<String, Object> properties)`
  - Add method `protected void deactivate(ComponentContext componentContext)`
  - Add method `public void updated(Map<String, Object> properties)`

- Acquire a static Logger instance to the source and log something on the hooks:
  - `private static final Logger s_logger = LoggerFactory.getLogger(ClassName.class);`
  - `s_logger.info(“Something to log”);`

- Fix project dependencies in `MANIFEST.MF`

- First test run on emulator
OSGi Services Overview

- **GPIOService**
  - Service used to access GPIO resources on the Gateway
  - This service is emulated in the Kura Emulator, and uses jdk.dio on the Raspberry Pi distribution

- **I2CService**
  - Service used to access I2C resources on the Gateway
  - This service is emulated in the Kura Emulator, and uses jdk.dio on the Raspberry Pi distribution
  - Source code for both APIs and implementation is provided in the work tree for reference

- **CloudService**
  - Service used to publish data to the connected MQTT Broker
Use of Services

Part 1

- Acquire needed Declarative Services from the component definition. (i.e. GPIOService)
  - Add service reference to GPIO Service in Component Definition
    - name="GPIOService"
    - policy="static"
    - bind="setGPIOService"
    - unbind="unsetGPIOService"
    - cardinality="1..1"
    - interface="org.eclipse.kura.gpio.GPIOService"

- Add service hooks to main class:
  - private static GPIOService m_GPIOService
  - protected void setGPIOService(GPIOService gpioService)
  - protected void unsetGPIOService(GPIOService gpioService)
Use of Services

Part 2

- Test GPIO Service on emulator
  - Activate GPIO logging
    - Edit emulator log4j.properties file and add
      log4j.logger.org.eclipse.kura.emulator.gpio=DEBUG
  - Acquire a GPIO Pin in the activation hooks and log activity
    - KuraGPIOPin pin = m_GPIOService.getPinByTerminal(18)
    - pin.setValue(true)

- Fix MANIFEST.MF file

- Test run on emulator
OSGi Metatype

Part 1

- Create a new metatype definition in the OSGI-INF folder:
  - New -> XML -> XML File
  - Name it using the fully qualified namespace of the class
  - Add Metatype Definition prototype
  - Set OCD id to fully qualified namespace
  - Set Designate pid to fully qualified namespace
  - Set Object ocdref to fully qualified namespace

- Add Object Component Definitions details

- Add sample Attribute Definition

- Have the main class implement ConfigurableComponent interface

- Add service pid on Component Definition xml

- Test run on emulator

- Open the WebUI at 127.0.0.1:8080 and check for the Attribute
OSGi Metatype

Part 2

- Change main class to use the new AD as the GPIO Pin to use:
  - Add a KuraGPIOPin member to the class
  - Add a static String to store the AD property ID
  - Retrieve the property value before acquiring the GPIO Pin

- Test run on the emulator
  - change AD value and test results on OSGi Console
  - stop the bundle and test results on OSGi Console and WebUI
  - start the bundle and test results on OSGi Console

- Experiment with new ADs and Services
  - Navigate through the Kura API
  - Explore cardinality, defaults, validation
Cloud Service

Part 1

- Add a service reference to CloudService
  - Interface: org.eclipse.kura.cloud.CloudService
  - setCloudService(CloudService cloudService)
  - unsetCloudService(CloudService cloudService)

- Add a CloudClient member to the main class

- Have the main class implement CloudClientListener

- Acquire the CloudClient using the CloudService upon bundle activation and set main class as listener:
  - cloudClient = cloudService.newCloudClient(APP_ID)
  - cloudClient.addCloudClientListener(this)

- Release CloudClient upon bundle deactivation
Cloud Service

Part 2

- Implement all the methods of the CloudClientListener

- Add a `doPublish(String metricName, Object value)` to the source
  - Build a new KuraPayload
  - Set its timestamp
  - Add metrics to the payload
  - Define a MQTT topic to send data to
  - Publish data to the Cloud using the CloudClient

- Add calls to `doPublish(s,o)` inside the source when GPIO events occur
  - `doPublish(pinNumber, pinValue)`
Cloud Service

Part 3

- Launch the emulator and check for bundle statuses (ss)
- Open the WebUI at 127.0.0.1:8080
- Edit MqttDataTransport tab:
  - broker-url: mqtts://broker-sandbox.everyware-cloud.com:8883
  - topic.context.account-name: EclipseCon
  - username: EclipseCon_broker
  - password: EclipseCon#54321
- Edit DataService
  - connect.auto-on-startup: true
- Check results on the emulator Console
Executors and Multi Threading

Part 1

- Create a Randomizer class implementing Runnable
  - Add randomDelay and randomizer variables (programmable)
  - Add an infinite loop in the run() method
  - Wait for a random (programmable) timeout
  - Start acquiring the laser and log (complete later)
  - Wait for programmable acquiring period
  - Stop acquiring the laser and log (complete later)

- Add a static ExecutorService and Future<?> to the main class

- Initialize ExecutorService in the main class constructor
  - m_impulse_executor = Executors.newSingleThreadExecutor()

- Remove debug code from activation hooks
Executors and Multi Threading

Part 2

- Add a new Randomizer member to the main class

- Create the `doUpdate()` method in main class
  - Check for `Future<?>`
  - Start a new Future using the `ExecutorService` and assigning the Randomizer instance

- Invoke `doUpdate()` both from the Activate and the Updated methods

- Check for `Future<?>` and `ExecutorService` in the Deactivate method

- Fix project dependencies

- Test run on the emulator
Digital Light Sensor Class

Part 1

- Add KuraI2CService to the main class

- Create a new DigitalLightSensorWorker class and have it implement Runnable

- Pass KuraI2CService in the constructor and store it in a static member

- Add a `private static initDevice()` method
  - Add a KuraI2CDevice static member to the class
  - Init the KuraI2CDevice as per Grove DLS specifications

- Add a `private static closeDevice()` method and close the I2C Device

- Add `public static void startAcquiringLaser()`, `public static void stopAcquiringLaser()`, and `public static boolean isAcquiringLaser()`
Digital Light Sensor Class

Part 2

- Implement the run() method
  - Check for KuraI2CSensor and init() if needed
  - Start an infinite loop
  - Read values from the KuraI2CSensor
  - Calculate LUX value
    - Use a thresholdLux local member to store first Lux read and calculate variations
    - Refer to Utilities class
  - Alert GameLogic if needed and log (complete later)
    - Check if value read is above programmed variation
    - Call GameLogic.laserDetected(luxValue)
  - Add a programmable delay for I2C read resolution
  - Close the KuraI2CDevice in a finally clause
Digital Light Sensor Class

Part 3

- Edit the Randomizer
  - Add call to `startAcquiringLaser()`
  - Add call to `stopAcquiringLaser()`

- Add a new `ScheduledExecutorService` and `ScheduledFuture<?>` to the main class

- Init the `ScheduledExecutorService` in main class constructor
  - `m_dls_executor = Executors.newSingleThreadScheduledExecutor()`

- Add executor checks to `doUpdate()` method:
  - Check if the `ScheduledFuture` is already running and stop it if needed
  - Schedule a new `DigitalLightSensor` worker at fixed delay:
  - `m_dls_handle = m_dls_executor.scheduleWithFixedDelay(new DigitalLightSensor(m_I2CService), 1, 2, TimeUnit.SECONDS)`

- Fix dependencies and test run on the emulator
GPIO Actuator Class

Part 1

- Create a new GPIOActuator class

- Add 2 Runnables to the class. They will be used for LED and Buzzer

- Add 2 static KuraGPIOPin members, for LED and Buzzer

- Create a public static void init() method
  - Initialize the GPIO pins and pass the GPIOService
  - Initialize the runnables so to blink the LED and make the buzzer sound
  - Create a CachedThreadPool executor

- Add a public static blinkLed() method
  - Invoke the executor service and pass the led runnable

- Add a public static buzz() method
  - Invoke the executor service and pass the buzzer runnable
GPIO Actuator Class

Part 2

- Add a public static method
  - Close the pins
  - Invoke the method from Deactivate hook on main class

- Initialize the GPIO Actuator class from the main class
  `doUpdate()`

- Edit the `startAcquiringLaser()` inside `DigitalLightSensor`
  - Add a call to `GPIOActuator.blinkLed()`

- Fix project dependencies

- Test run on emulator
Game Logic Implementation

Part 1

- Create a new GameLogic class
- Create a `public static void trigger()` method
  - Have it blink the LED
  - Invoke the main class `doPublish(S,O)` method and publish a timestamp
  - Remove `blinkLed()` call from DigitalLightSensor
  - Move DigitalLightSensor activation logic from the `Randomizer.run()` method
  - Add the «missfire» message publishing logic
- Create a `public static void score()` method
  - Have it sound the buzzer
  - Stop DLS laser acquisition
  - Remove `stopAcquiringLaser()` from DigitalLightSensor.run()
  - Invoke the main class `doPublish(S,O)` method and publish a timestamp
Game Logic Implementation

Part 2

The game logic should include the following:

- Increment a miss counter every time the laser doesn’t get detected.

- Increment a score counter for each scored hit, and reset the miss.

- If the miss counter is greater than a threshold (programmable), reset the score.

Data will be published in real time and can be used by a dashboard to visualize the game logic.
Have fun with the emulator!

Experiment with the emulator, adding programmable properties and tweaking the game logic.
Remote Debugging

Deploy and debug your application directly from Eclipse

#DEBUGALLTHETHINGS
Start mToolkit and Deploy

- Create a new mToolkit server and give it the Pi IP address
- Connect mToolkit to the Pi

Create a jar file for the ShootAPi Bundle
- Right click on project tree -> Export -> Plug-in Development
- Deployable plug-ins and fragments
- Export the jar file to a folder

Install ShootAPi into Kura on the Pi
- Right click on mToolkit tree -> Install new bundle
- Choose the previously created jar file

Start a remote debugging session
- Right click on project tree -> Debug as... -> Debug configurations... -> Remote Java Application
- Select the ShootAPi project
- Fill in Host and Port fields
- Debug
Deploy on Target

- Create a new Deployment Package Definition
  - Right click on main project tree -> New... -> OSGi -> Deployment Package Definition
  - Fill in the fields
  - Open the file and navigate to the Bundles tab
  - Add a new bundle and chose the previously created jar file

- Create a Deployment Package
  - Right click on Deployment Package Definition -> Quick build...

- Access the Kura Web UI and deploy the package
  - Open 192.168.2.10 (admin / admin)
  - Navigate to the Packages tab
  - Install new package -> File
  - Chose the newly created .dp file  Ok

- Access the Kura log on the Pi
  - ssh to the Pi
  - tail -f /var/log/kura.log
Appendix

Quick reference slides
Condividere il WiFi con la 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
Shoot-A-Pi Arcade Shooter Simulator

Hardware Setup

I2C

Raspberry Pi B+ J8 Header

SDA1 - I2C
SCL1 - I2C

3.3v DC Power
5v DC Power

Ground

GPIO17

5v DC Power

GPIO18

Ground

GPIO
Shoot-A-Pi Arcade Shooter Simulator

Web Dashboard Architecture

MQTT Broker

Shoot A Pi

Web Dashboard

- MQTT Broker
- PAHO for JavaScript
- JSX Compressor
- Google Protocol Buffers
- Dashboard Logic
- ByteBuffer
Debugging e Logging

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.
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 and vote at eclipsecon.org

-1 0 +1