Sandboxed Applications Training

with MICROEJ SDK 6

MICROEJ®
SOFTWARE-DEFINED EVERYTHING

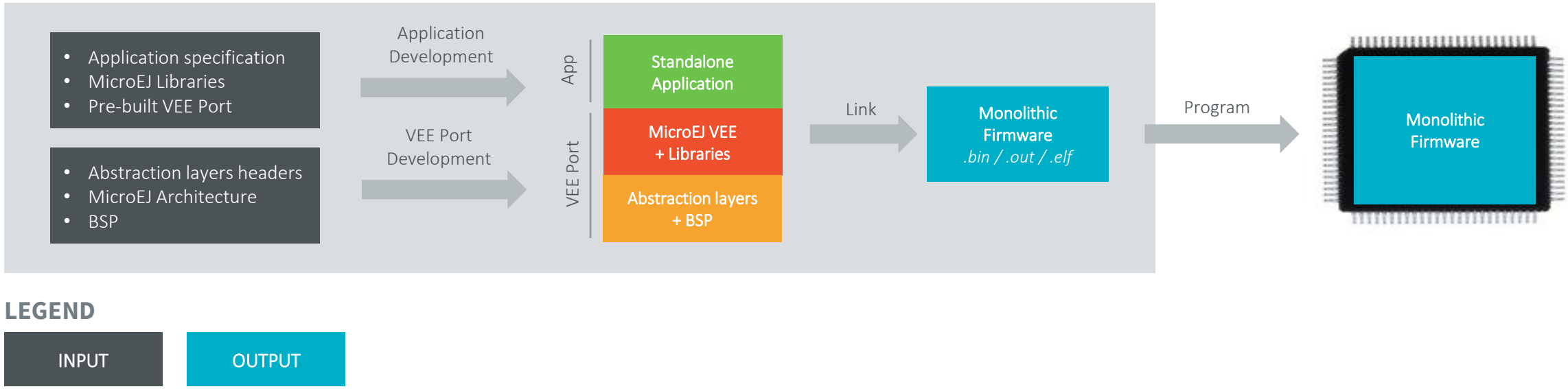
DISCLAIMER

AGENDA

1. Sandboxed Applications Concepts
2. Sandboxed Application Project Creation
3. Inter-Application Communication: focus on Shared Interfaces
4. AppConnect: interact with the Applications through a Web UI

Sandboxed Applications Concepts

MONO-SANDBOX DEVELOPMENT WORKFLOW

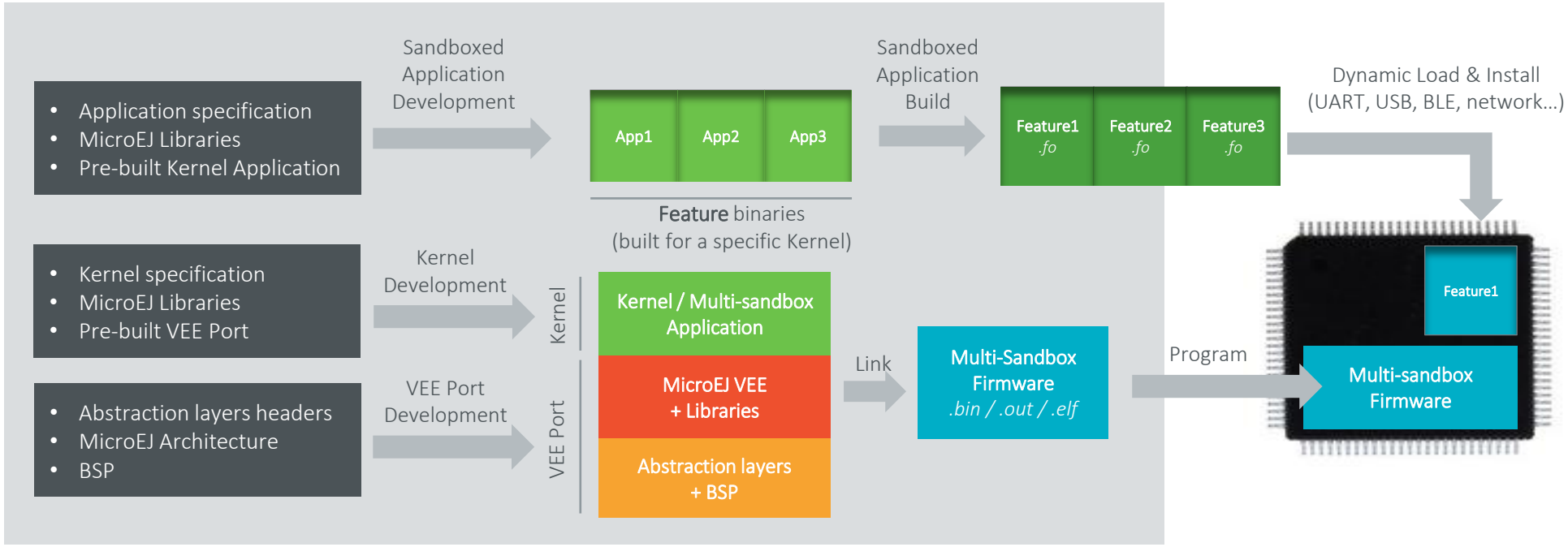


LEGEND

INPUT

OUTPUT

MULTI-SANDBOX DEVELOPMENT WORKFLOW

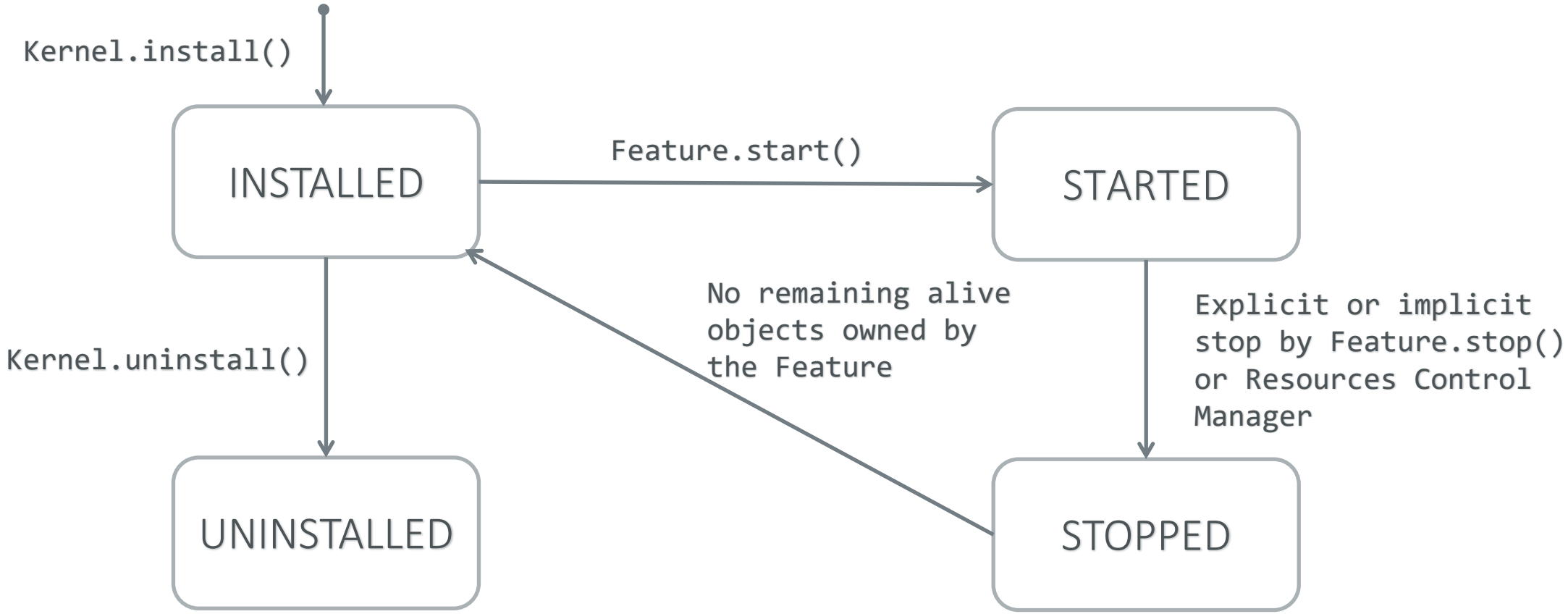


LEGEND

INPUT

OUTPUT

APPLICATION LIFECYCLE STATES (1/2)



APPLICATION LIFECYCLE STATES (2/2)

- **INSTALLED:**
  - The Application has been successfully linked to the Kernel and is not running. There are no references from the Kernel to objects owned by this Application.
- **STARTED:**
  - The Application has been started and is running.
- **STOPPED:**
  - The Application has been stopped and all its owned threads are terminated. The memory and resources are not yet reclaimed.
- **UNINSTALLED:**
  - The Application has been unlinked from the Kernel.

Prerequisites

PREREQUISITES

Hardware required:

- NXP i.MX RT1170 Evaluation Kit (EVKB) + micro-USB cable + RK055HDMIPI4MA0 display panel.

Environment setup:

- Follow the NXP i.MX RT1170 Evaluation Kit Getting Started to setup your environment and run a demo application on the Virtual Device and on the i.MX RT1170 Evaluation Kit.
- Follow the Get Started with Sandboxed Applications to learn how to create a Sandboxed Application.

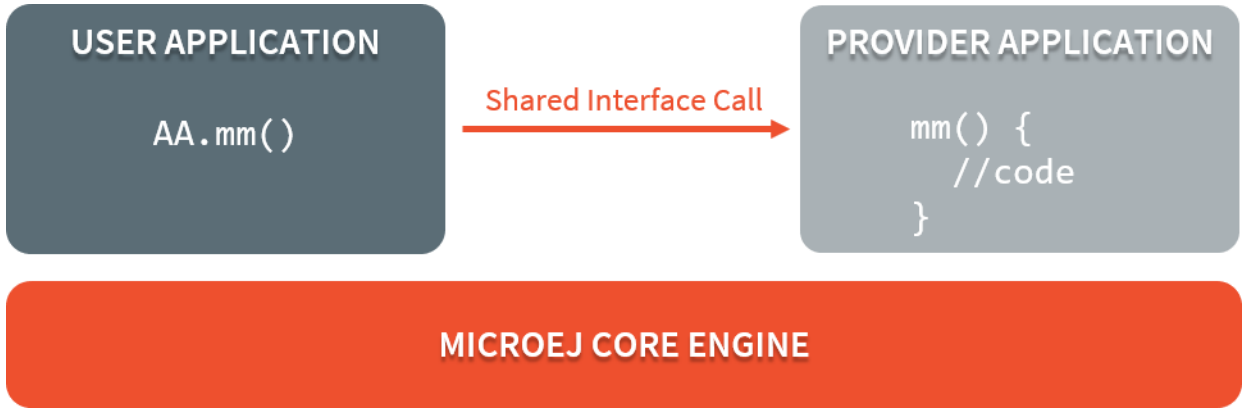
Make sure your board is flashed with the Green Kernel (following the Get Started with Sandboxed Applications)

- Note: the next slides are using **IntelliJ IDEA** with **MicroEJ plugin for IntelliJ IDEA 1.5.0**. This training supports all other available IDEs (Android Studio, VS Code, …)

Shared Interfaces

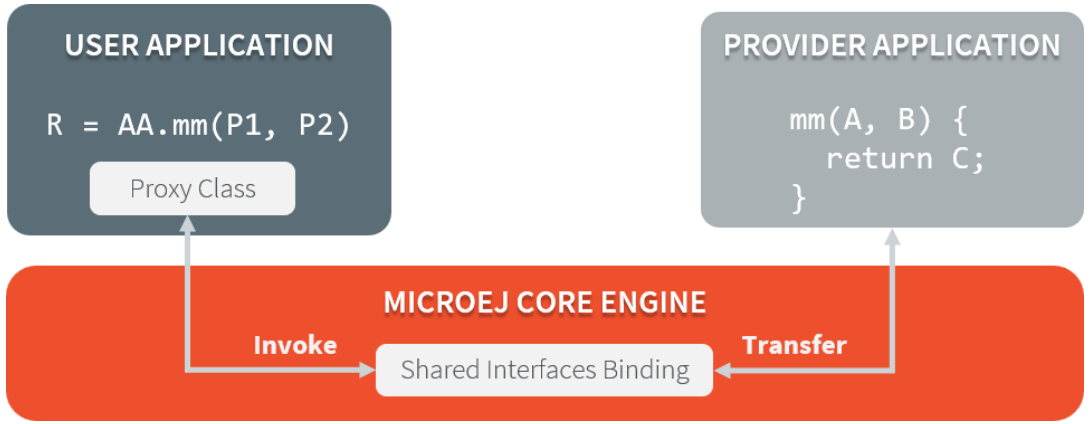
OVERVIEW

- The Shared Interface mechanism provided by the MicroEJ Core Engine is an object communication bus based on plain Java interfaces, where method calls are allowed to cross MicroEJ Sandboxed Applications boundaries.
- The Shared Interface mechanism is the cornerstone for designing reliable Service Oriented Architectures on top of MicroEJ. Communication is based on the sharing of interfaces defining APIs (Contract Oriented Programming).
- The basic schema:
    - A provider application publishes an implementation for a shared interface into a system registry.
    - A user application retrieves the implementation from the system registry and directly calls the methods defined by the shared interface.



TRANSFERABLE TYPES (1/3)

- In the process of a cross-application method call, parameters and return value of methods declared in a Shared Interface must be transferred back and forth between application boundaries.



- Some restrictions apply to Shared Interfaces compared to standard java interfaces:
  - Types for parameters and return values must be **transferable types** (Look at the Object Binding documentation to learn why).
  - Thrown exceptions must be classes owned by the MicroEJ Firmware.

TRANSFERABLE TYPES (2/3)

- The table bellow describes the rules applied depending on the element to be transferred:

Type	Type Owner	Instance Owner	Rule
Primitive Type	N/A	N/A	Passing by value. (boolean, byte, short, char, int, long, double, float)
Any Class, Array or Interface	Kernel	Kernel	Passing by reference
Any Class, Array or Interface	Kernel	Application	MicroEJ Kernel specific or forbidden
Array of base types	Any	Application	Clone by copy
Arrays of references	Any	Application	Clone and transfer rules applied again on each element
Shared Interface	Application	Application	Passing by indirect reference (Proxy creation)
Any Class, Array or Interface	Application	Application	Forbidden

TRANSFERABLE TYPES (3/3)

- Objects created by a Sandboxed Application which type is owned by the Kernel can be transferred to another Sandboxed Application provided this has been authorized by the Kernel.

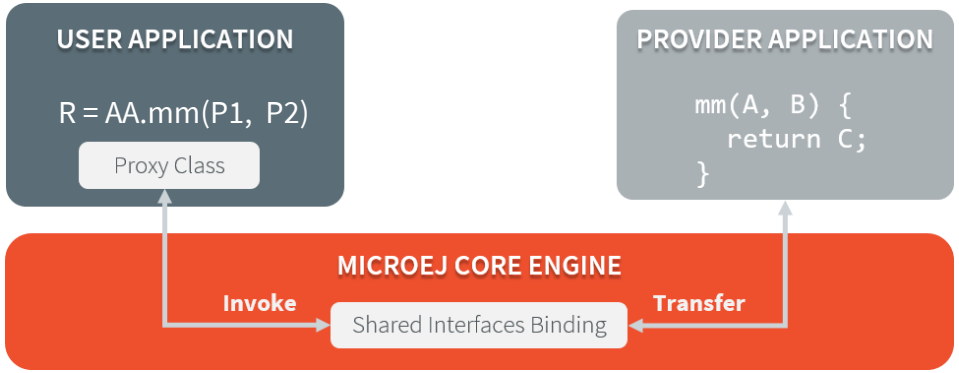
- The list of Kernel types that can be transferred is Kernel specific, so you have to consult your Kernel specification.

- When an argument transfer is forbidden, the call is abruptly stopped and a **java.lang.IllegalAccessError** is thrown by the Core Engine.

- For the forbidden types to be transferable, a dedicated Kernel Type Converter must have been registered in the Kernel.

PROXY CLASS (1/2)

- The Shared Interface mechanism is based on automatic proxy objects created by the underlying MicroEJ Core Engine, so that each application can still be dynamically stopped and uninstalled.
- This offers a reliable way for users and providers to handle the relationship in case of a broken link.
- Once a Java interface has been declared as Shared Interface, a dedicated implementation is required (called the Proxy class implementation).



- Its main goal is to perform the remote invocation and provide a reliable implementation regarding the interface contract even if the remote application fails to fulfill its contract (unexpected exceptions, application killed…).
- The MicroEJ Core Engine will allocate instances of this class when an implementation owned by another application is being transferred to this application.

PROXY CLASS (2/2)

- A proxy class is implemented and executed on the client side; each method of the implemented interface must be defined according to the following pattern:

```
package mypackage;

public class MyInterfaceProxy extends Proxy<MyInterface> implements MyInterface {

    @Override
    public void foo(){
        try {
            invoke(); // perform remote invocation
        } catch (Throwable e) {
            e.printStackTrace();
        }
    }
}
```

- Each implemented method of the proxy class is responsible for performing the remote call, catching all errors from the server side and providing an appropriate answer to the client application call according to the interface method specification (contract).
- Remote invocation methods are defined in the super class **ej.kf.Proxy** and are named **invokeXXX()** where **XXX** is the kind of return type.

Hands On

HANDS ON PACKAGE

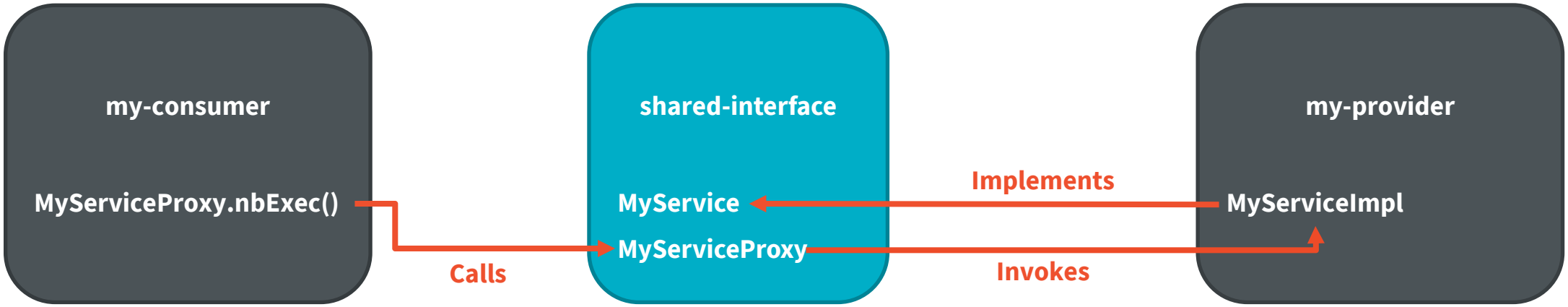
Get the shared-interfaces-hands-on training package provided next to those slides.

The training package contains one directory:

- **shared-interfaces-hands-on**: This directory contains the project structure and code basis to follow this hands on.

HANDS ON OVERVIEW

- A **shared-interface** library provides a **MyService** interface with a method **nbExec**.
- **my-provider** application provides an implementation of **MyService**.
- **my-consumer** will be updated to call this methods using the shared interface mechanism.



Create the Shared Interface

INTERFACE DEFINITION (1/2)

- The definition of a Shared Interface starts by defining a standard Java interface.
- Open the **SharedInterface** project from the **shared-interfaces-hands-on** folder.
- In the **shared-interface** module, create a new class MyService in the **com.microej.example.si** package:

```
package com.microej.example.si;

/**
 * Shared interface MyService
 */
public interface MyService {

   /**
    * Prints and returns the number of times this method has been executed.
    * @return the number of times this method has been executed.
    */
   int nbExec();

}
```

INTERFACE DEFINITION (2/2)

- To declare an interface as a Shared Interface, it must be registered in a Shared Interfaces identification file.
- A Shared Interface identification file is an XML file with the **.si** suffix with the following format:

```
<sharedInterfaces>
   <sharedInterface name="com.microej.example.si.MyService" />
</sharedInterfaces>
```

- Shared Interface identification files must be placed at the root of a path of the application classpath.
- For a MicroEJ Sandboxed Application project, it is typically placed in **src/main/resources** folder.

- **Hands on:**
    - Add a **sharedInterfaces.si** file in the **src/main/resources** folder of the shared-interface module.

PROXY IMPLEMENTATION

- In the **shared-interface** module:
  - Create a sub-class of **Proxy** that implements **MyService**:
  - **Note: the Proxy class name must follow the following pattern: {InterfaceName}Proxy and should be put in the same package as the interface.**

```
package com.microej.example.si;

import ej.kf.Proxy;
import java.util.logging.Logger;

/**
 * Proxy class for {@link MyService} shared interface.
 */
public class MyServiceProxy extends Proxy<MyService> implements MyService {

   private static final Logger LOGGER = Logger.getLogger(MyServiceProxy.class.getSimpleName());

   @Override
   public int nbExec() {
      try {
         return invokeInt();
      } catch (Throwable t) {
         LOGGER.severe(t.getMessage());
      }
      return -1;
   }
}
```

The interface

Returned Type

USE THE SERVICE

- In the **my-consumer** module:
  - In the **build.gradle.kts**, add the following dependency:

```
implementation("ej.library.runtime:service:1.2.0")
```

  - In **MyConsumer**, update the code of the **start()** method to use a Timer task that periodically invokes the service:

```
timer.schedule(new TimerTask() {
    @Override
    public void run() {
        MyService myService = ServiceFactory.getService(MyService.class);
        if (myService != null) {
            myService.nbExec();
        }
    }
}, 0, PRINT_PERIOD);
```

  - Don’t forget to stop the TimerTask in the **stop()** method.

Implement the Shared Interface

Implement **MyService** in the **my-provider** application

IMPLEMENT THE SERVICE

- In the **my-provider** module:
  - Create a new class **MyServiceImpl** that implements **MyService**:

```
package com.microej.example.myprovider;

import com.microej.example.si.MyService;

public class MyServiceImpl  implements MyService {

   private int nbExec = 0;

   @Override
   public int nbExec() {
      this.nbExec++;
      System.out.println("Number of executions: " + this.nbExec);

      return this.nbExec;
   }

}
```

REGISTER THE SERVICE

- In **my-provider**:
  - Add the following dependency to the **build.gradle.kts**:

```
implementation("ej.library.runtime:service:1.2.0")
```

  - Update the **MyProvider** class.

```
@Override
public void start() {
   MyService myInstance = new MyServiceImpl();

   // registers the interface instance with the service registry
   ServiceFactory.register(MyService.class, myInstance);

   LOGGER.info("My Provider started.");//$NON-NLS-1$
}

@Override
public void stop() {
   ServiceFactory.unregister(MyService.class, ServiceFactory.getService(MyService.class));
   LOGGER.info("My Provider stopped.");//$NON-NLS-1$
}
```

RUN THE EXAMPLE IN SIM

- Run the **my-consumer** application with the runOnSimulator gradle task.
    - A dependency to **my-provider** was already added to the **my-consumer** **build.gradle.kts** so running this feature will automatically run **my-provider** too (see documentation).

```
dependencies {
    implementation("ej.api:edc:1.3.7")
    implementation("ej.api:kf:1.7.0")

    implementation("ej.library.eclasspath:logging:1.2.1")

    microejApplication(project(":my-consumer"))
    implementation(project(":shared-interface"))

    microejVee("$defaultKernelGroup:$defaultKernelModule:$defaultKernelVersion")
}
```

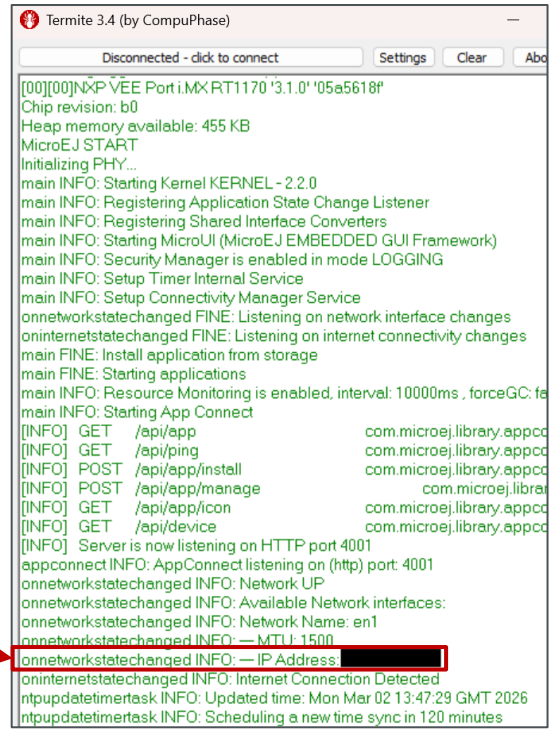
my-consumer/build.gradle.kts

```
============== [ Initialization Stage ] ==============
============== [ Converting fonts ] ==============
============== [ Converting images ] ==============
============== [ Launching on Simulator ] ==============
main INFO: Starting Kernel KERNEL - 2.1.0
main INFO: Registering Application State Change Listener
main INFO: Registering Shared Interface Converters
main INFO: Starting MicroUI (MicroEJ EMBEDDED GUI Framework)
main INFO: Security Manager is enabled in mode LOGGING
main INFO: Setup Timer Internal Service
main INFO: Setup Connectivity Manager Service
main INFO: Application running: my-provider
main INFO: Application running: my-consumer
main INFO: Starting App Connect
myconsumer INFO: My Consumer started.
myprovider INFO: My Provider started.
Number of executions: 1
appconnect INFO: AppConnect listening on (http) port: 4001
Number of executions: 2
Number of executions: 3
Number of executions: 4
```

RUN THE EXAMPLE ON DEVICE (1/2)

Get IP address of the board:

- Open the Termite serial terminal.
- Click the **Settings** button.
- Select the NXP i.MX RT1170 board COM port.
- Reset the NXP i.MX RT1170.
- Wait until you see the **IP address**



Set IP address for deployment

- Open **gradle.properties** on the Hands-on root folder
- Set ipAddress variable

```
#########################
# Application Deployment
#########################

# Board IP address
ipAddress=YOUR_IP
# Board port, default port is 4001
port=4001
```

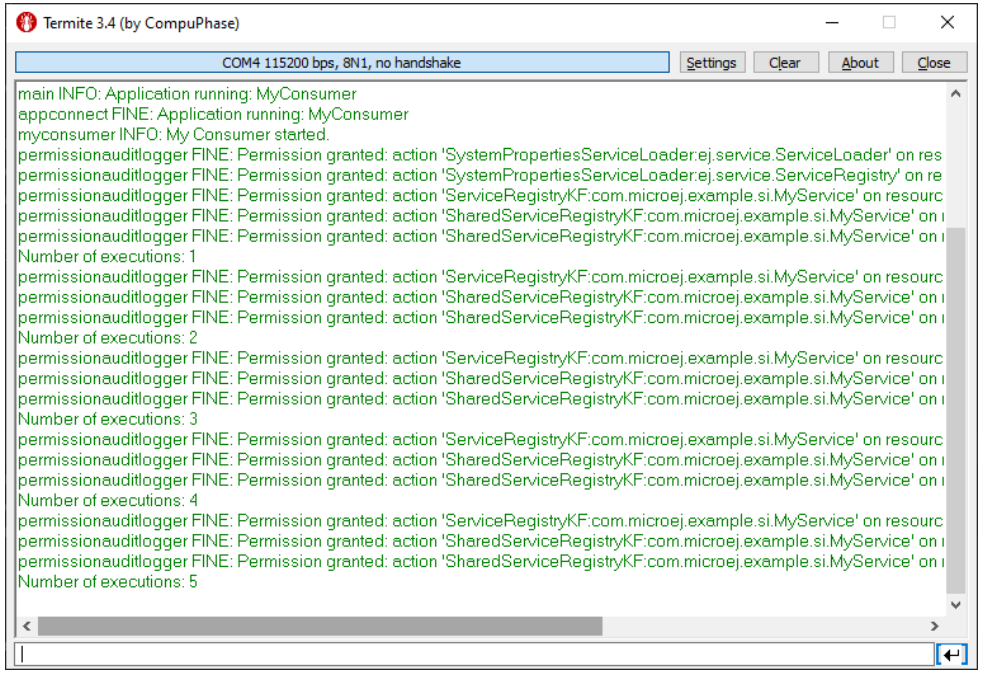
gradle.properties

RUN THE EXAMPLE ON DEVICE (2/2)

Deploy both **my-consumer** and **my-provider** features on the NXP i.MX RT1170:

- Run **my-consumer > tasks > microej > localDeploy**.
- Run **my-provider > tasks > microej > localDeploy**.

- Open the Termite serial terminal.
- Click the **Settings** button.
- Select the NXP i.MX RT1170 board COM port.
- Reset the NXP i.MX RT1170.
- The GREEN Kernel with the two features are started. **my-consumer** uses the service provided by **my-provider.**



Traces of the my-consumer execution

AppConnect

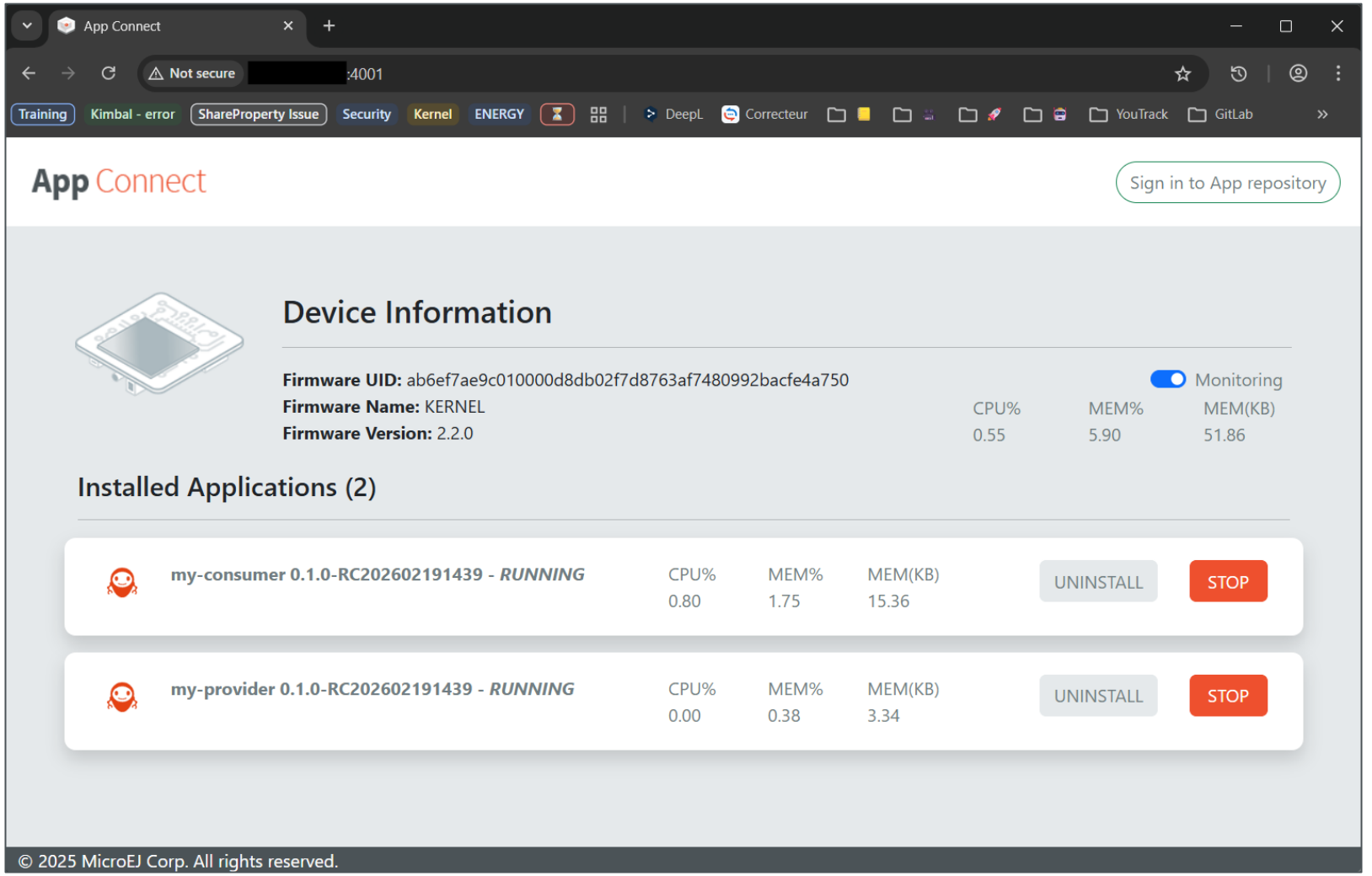
Interact with the applications through a Web UI

APPCONNECT

- AppConnect a library that allows managing on-board applications and installing compatible applications from a remote application store.
- The **localDeploy** task previously used, uses AppConnect to install the applications on the board.
- AppConnect also provides a Web UI that allows to interact with the applications.
- AppConnect is already available with the GREEN Kernel. The Web UI can be accessed on any browser by connecting to the board's IP address and the port 4001

See Documentation

APPCONNECT



AppConnect Web UI

APPCONNECT

- The START/STOP button allows to start or stop the application. Stopping my-consumer gives the following traces:

```
Number of executions: 4
myconsumer INFO: My Consumer stopped.
main INFO: Application stopped: MyConsumer
appconnect FINE: Application stopped: MyConsumer
main INFO: Application completely stopped, and can be uninstalled: MyConsumer
appconnect FINE: Application completely stopped, and can be uninstalled: MyConsumer
```

- The UNINSTALL button uninstall the feature from the board:

```
main INFO: Application uninstalled: MyConsumer
appconnect FINE: Application uninstalled: MyConsumer
```

GOING FURTHER

Seeing for resource management of:

- CPU
- RAM
- Flash storage
- Network bandwidth
- Priority

The Resource Manager will allow you to customize each feature's configuration in the Kernel.

THANK YOU

for your attention !