Kernel Development

with MICROEJ SDK 6

© MICROEJ 2026

MICROEJ®

DISCLAIMER

All rights reserved. Information, technical data and tutorials contained in this document are proprietary under copyright law of MicroEJ S.A. Without written permission from MicroEJ S.A., copying or sending parts of the document or the entire document by any means to third parties is not permitted. Granted authorizations for using parts of the document or the entire document do not mean MicroEJ S.A. gives public full access rights.

The information contained herein is not warranted to be error-free.

MicroEJ® and all relative logos are trademarks or registered trademarks of MicroEJ S.A. in France and other Countries.

Other trademarks are proprietary of their respective owners.

Java™ is Sun Microsystems' trademark for a technology for developing application software and deploying it in cross-platform, networked environments. When it is used in this site without adding the "™" symbol, it includes implementations of the technology by companies other than Sun. Java™, all Java-based marks and all related logos are trademarks or registered trademarks of Sun Microsystems Inc, in the United States and other Countries.

AGENDA

1. Introduction to Multi-sandboxed Application
2. Creation of a new Kernel
3. Creation of a Shared Service
4. Manage permissions with Security Manager

© MICROEJ 2026

HARDWARE REQUIRED

- Hardware required:
  - NXP i.MX RT1170 Evaluation Kit (EVKB)
  - RK055HDMIPI4MA0 display panel
  - micro-SD card + adapter for PC
  - micro-USB cable
  - Ethernet cable to connect to a router

© MICROEJ 2026

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.

Note: the next slides are using IntelliJ IDEA. This training supports all other available IDEs (Android Studio, VS Code, …)

This training requires the Getting Started to be completed until the **Run an Application on the i.MX RT1170 Evaluation Kit** section (included).

⚠️ The path to the NXP i.MX RT1170 VEE Port sources should be as short as possible and contain **no whitespace** or **non-ASCII character.**

© MICROEJ 2026

Introduction

© MICROEJ 2026

WHAT IS MULTI-SANDBOX?

Multi-Sandbox is a functionality of MICROEJ VEE.

It is similar to any execution environment that power smart mobile devices (cellphone/tablets), with similar primary features, such as a **dynamic application loader** or **application life cycle management**.

Applications running in a Multi-Sandbox environment are called **Sandboxed Applications.**
Whereas applications running in a Mono-Sandbox environment are called **Standalone Applications.**

MICROEJ VEE Multi-Sandbox environnent enables:

- **Isolation ("Sandboxing")**: each Sandboxed Application operates independently, which helps prevent one application from affecting the performance or security of another.
- **Security**: additionally to isolation, security policies can be defined at system level to prevent Sandboxed Applications from accessing resources or calling specific system APIs.
- **Resource Management**: the system can allocate and monitor resources such as memory and processing power to different Sandboxed Applications, optimizing the overall performance of the system.
- **Flexibility**: developers can deploy multiple applications on the same hardware platform, making it easier to update or modify individual applications without impacting the entire system.

© MICROEJ 2026

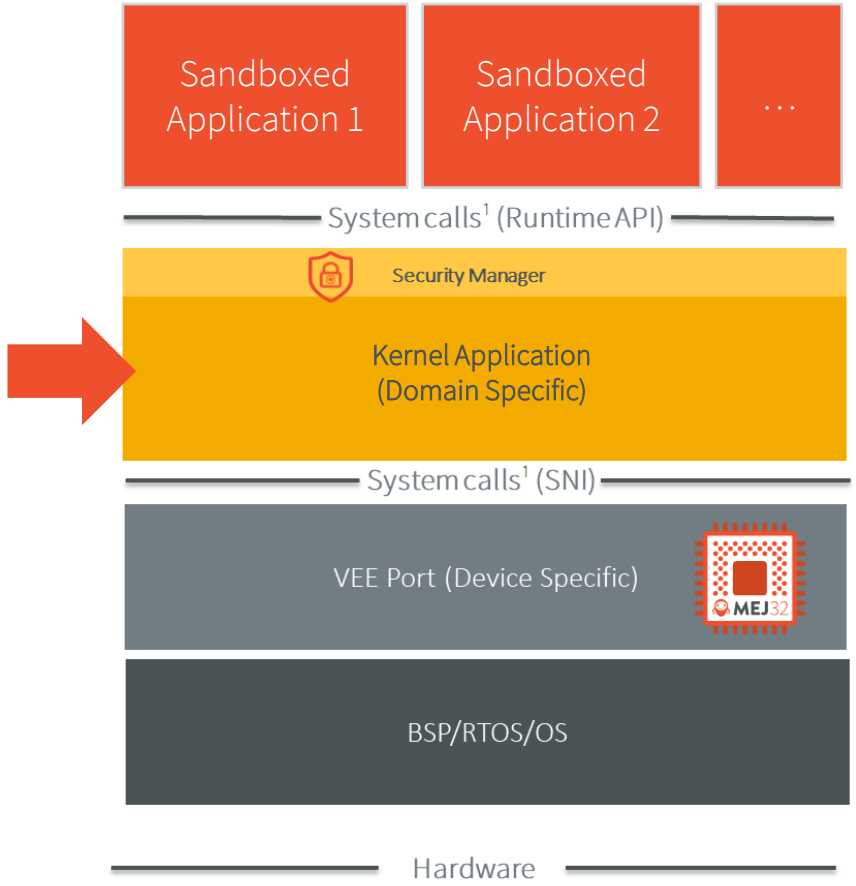
WHAT IS A KERNEL?

The Kernel Application is a Standalone Application that can be extended dynamically to install, run, and control the execution of new applications called Sandboxed Applications.

This training focuses on Kernel development.

Here is a non-exhaustive list of the activities to be done by Kernel Developers:

- Integrating the Kernel Application with a VEE Port to produce a Multi-Sandbox Executable and Virtual Device
- Defining the set of APIs that will be exposed to Applications, optionally by maintaining a custom Runtime API Environment
- Managing lifecycles of applications (deciding when to install, start, stop and uninstall them)
- Defining and applying permissions on system resources using the Security Manager (rules & policies)
- Managing connectivity
- Controlling and monitoring resources



© MICROEJ 2026

MULTI-SANDBOX LAYERS

MICROEJ

APIs Whitelist

Application developers have restricted access to the subset of APIs exposed by Kernel (Ecosystem)

Strong Isolation

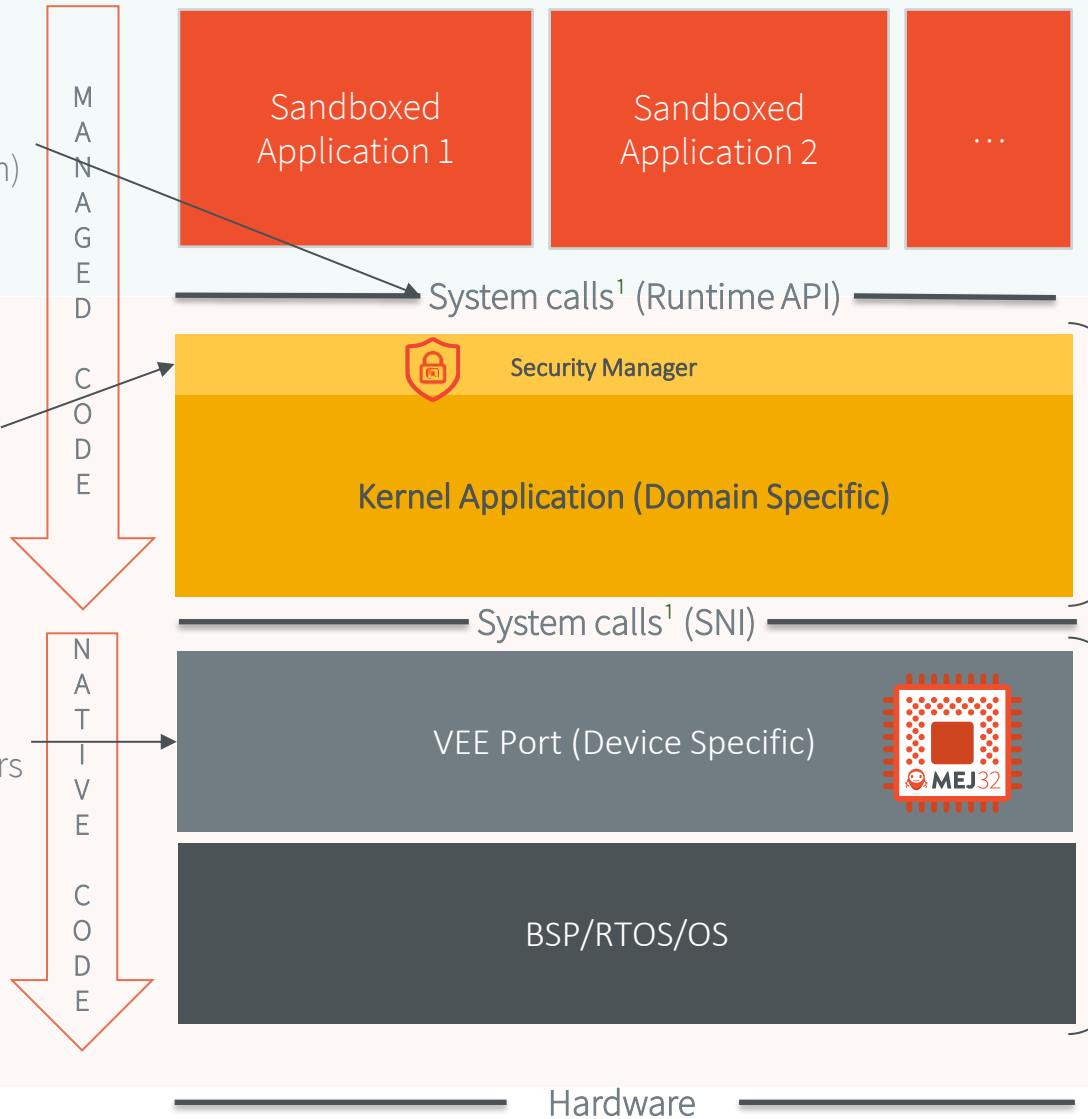
Build time checks

Access Control Permissions

Runtime checks

Abstraction Layer

Providing portability to upper layers



Application Lifecycle Management

- OTA Updates
- Dynamic Loading
- Sandboxed Execution

Hardware Independent Kernel

Portable on hardware with same capabilities

Software Defined Hardware

Virtual Device

1. System calls here refers to API invocation that manipulate critical resources

© MICROEJ 2026

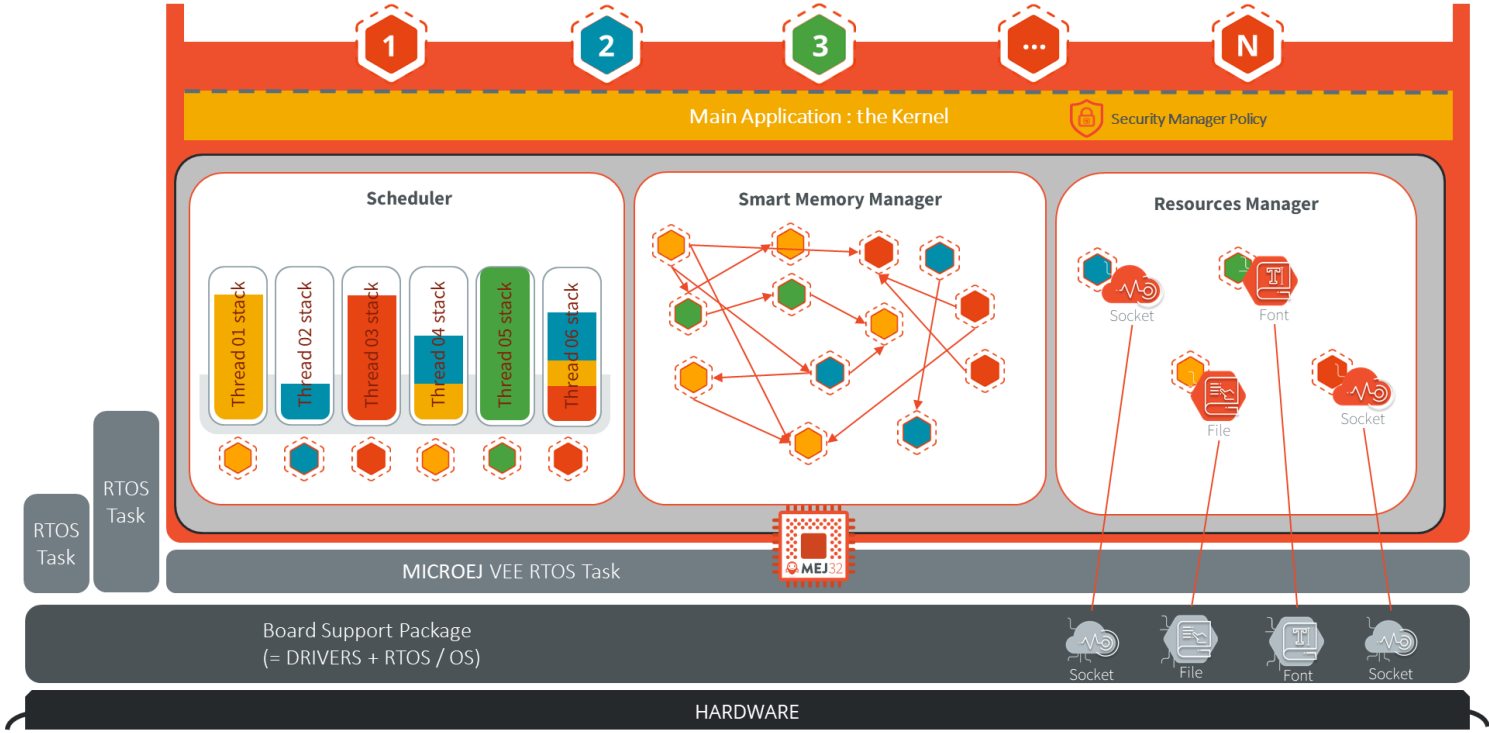
MULTI-SANDBOX ISOLATION CAPABILITY

Isolation is guaranteed thanks to the following mechanisms:

- **MICROEJ VEE Multi-Sandbox capability:**
  - No (bypassing) native code call allowed from Sandboxed Applications.
  - Sandboxed Applications can't access directly to code, objects, threads.
  - Native Resources are automatically reclaimed on Sandboxed Application kill (e.g. socket).

- **Kernel API**: make available at **build-time** a subset of system APIs to Sandboxed Applications.

- **Security Manager**: control the system APIs called by Sandboxed Applications **at runtime**.



© MICROEJ 2026

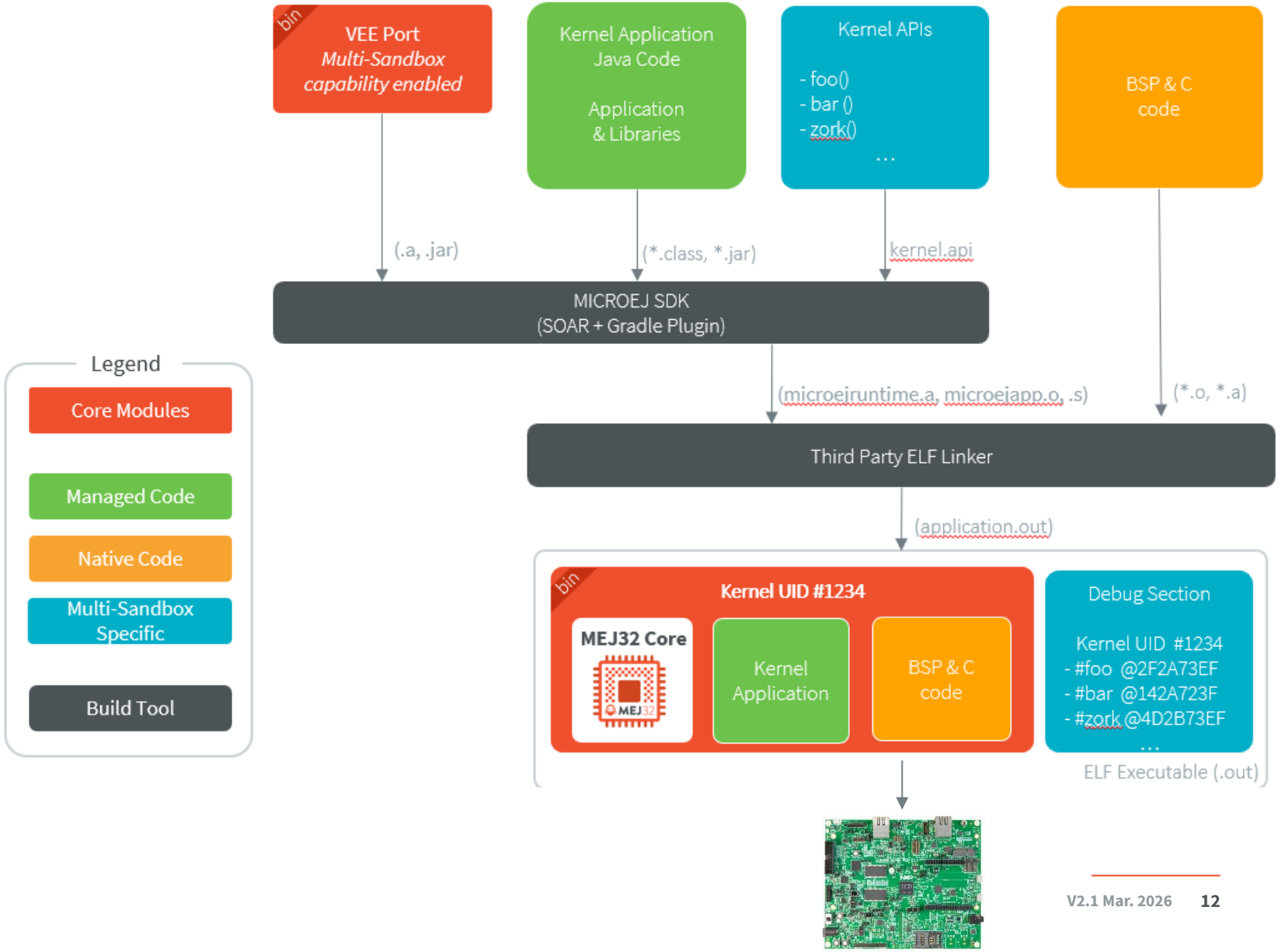
KERNEL & FEATURES CONCEPTS

- The build output file of a Sandboxed Application against a Kernel is called a Feature (.fo binary file).
- MICROEJ VEE Multi-Sandboxing is based on the Kernel & Features specification (KF).
  It allows an application code to be split between multiples parts: the main application, called the Kernel and zero or more Sandboxed Applications linked as Features.
- Kernel part is mandatory and is assumed to be reliable, trusted and cannot be modified.
- The Kernel is the main entry point that the system starts with. The **main()** function is implemented in the Kernel project.
- Features are fully controlled by the Kernel:
  - Features are considered as “code extensions”.
  - They can be dynamically installed, started, stopped and uninstalled at any time independently of the system state.
  - A Feature never depends on another Feature to be stopped.
  - Resources accesses (RAM, hardware peripherals, CPU time, …) are under control of the Kernel.

© MICROEJ 2026

KERNEL BUILD FLOW

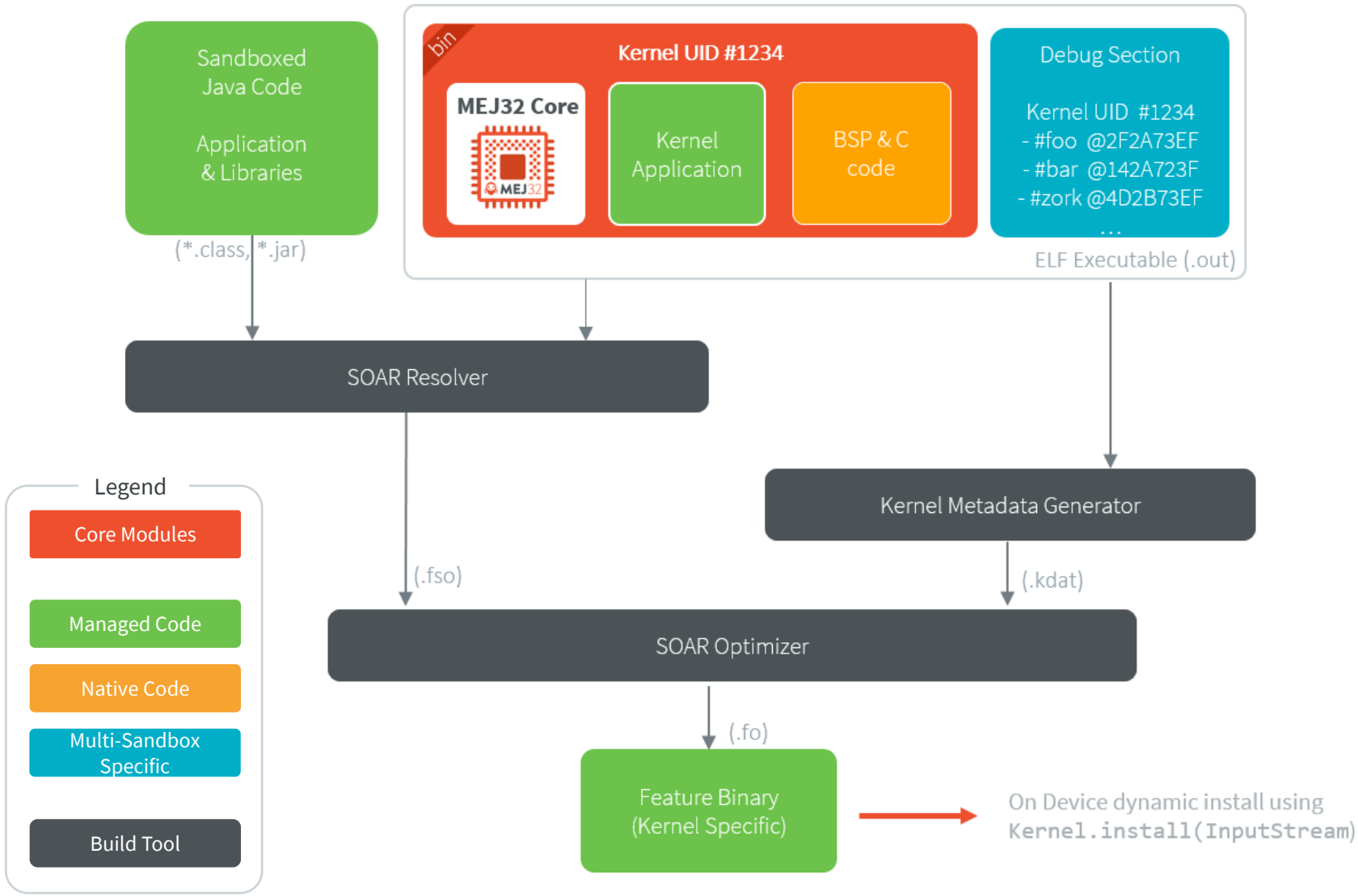
- The **Kernel Application** is the main application that is executed by MICROEJ VEE. It is linked statically to produce a **Mono-Sandbox Executable**.
- The **Mono-Sandbox Executable** is the firmware to flash on the device.
- The **Kernel UID** is a sequence of bytes that uniquely identifies the Kernel. Two Kernels built from the same Kernel Application code **will not share the same UID**.
- The **Kernel UID** is used by **Core Engine** to check if an Application can be installed on a Kernel. During the Application build, the resulting **.fo file embeds the Kernel UID** on which it has been built.



© MICROEJ 2026

FEATURE BUILD FLOW

- Once a Kernel has been generated, additional Sandboxed Application code (Feature) can be built against the Kernel.
- The binary file produced (the .fo file) can be installed **only** on the Kernel on which it was generated (based on the Kernel UID).
- A .fo can be made compatible with other Kernel UIDs, see Feature Portability Control.



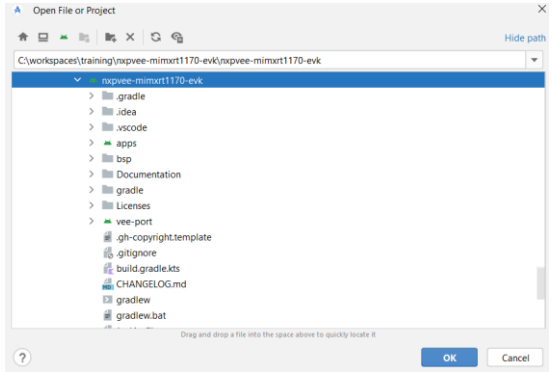
© MICROEJ 2026

Run a Sandboxed Application on a Kernel

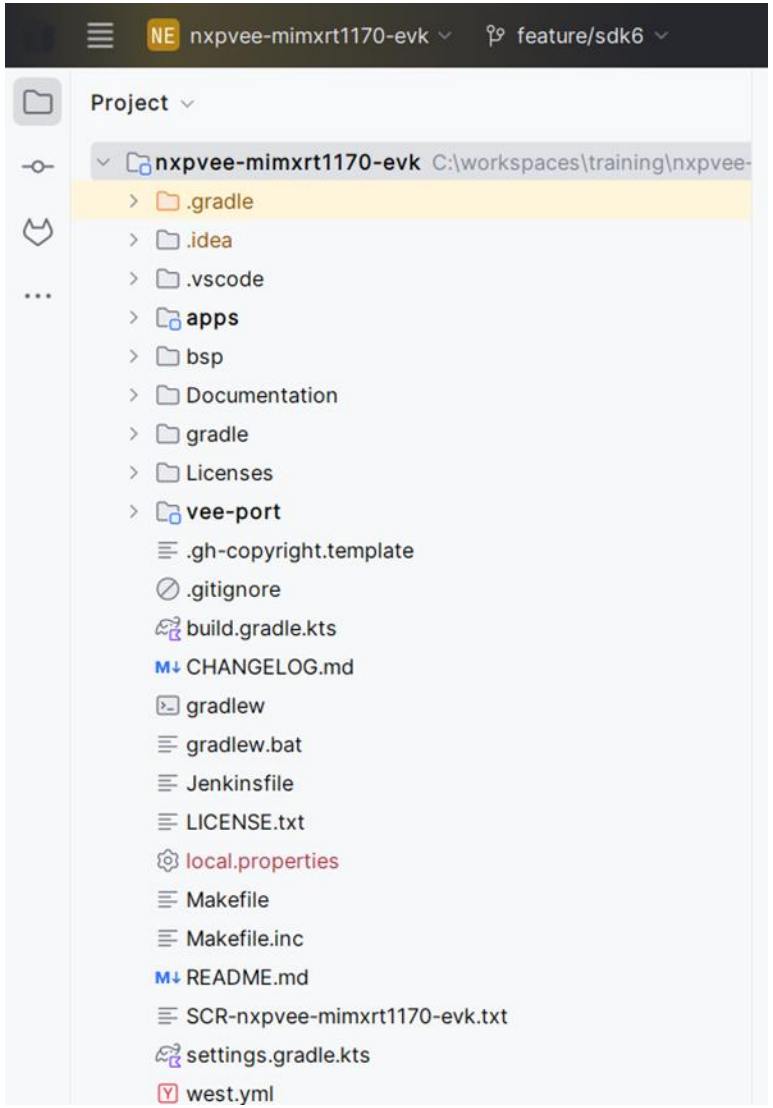
© MICROEJ 2026

OPEN THE VEE PORT SOURCES IN THE IDE

- Open your IDE.
- If no project is opened in the IDE yet, click on **Open.**
  Otherwise, go to **File > Open.**
- Browse to the VEE Port sources folder: **nxpvee-mimxrt1170-evk:**



- Click on **OK**.
- The project sources appear in the Projects view.
- The next slide explains how to run an application on the Simulator, to validate the environment setup.



© MICROEJ 2026

ENABLE THE MULTI-SANDBOX CAPABILITY

Enable the Multi-Sandbox capability of the Core Engine in the NXP i.MXR1170 VEE Port:

- Open the **configuration.properties** file of the VEE Port (vee-port/configuration.properties)
- Change the property **com.microej.runtime.capability** to **multi**

```
# Enable Multi-Sandbox (default value is "mono", available values are "mono", "tiny" and "multi").
com.microej.runtime.capability=multi
```

The Multi-Sandbox capability requires the LLAPIs provided in **LLKERNEL_RAM.h** to be implemented.

The NXP i.MXRT1170 VEE Port already provides a default implementation in **bsp/vee/port/kf/src/LLKERNEL_RAM.c.**

© MICROEJ 2026

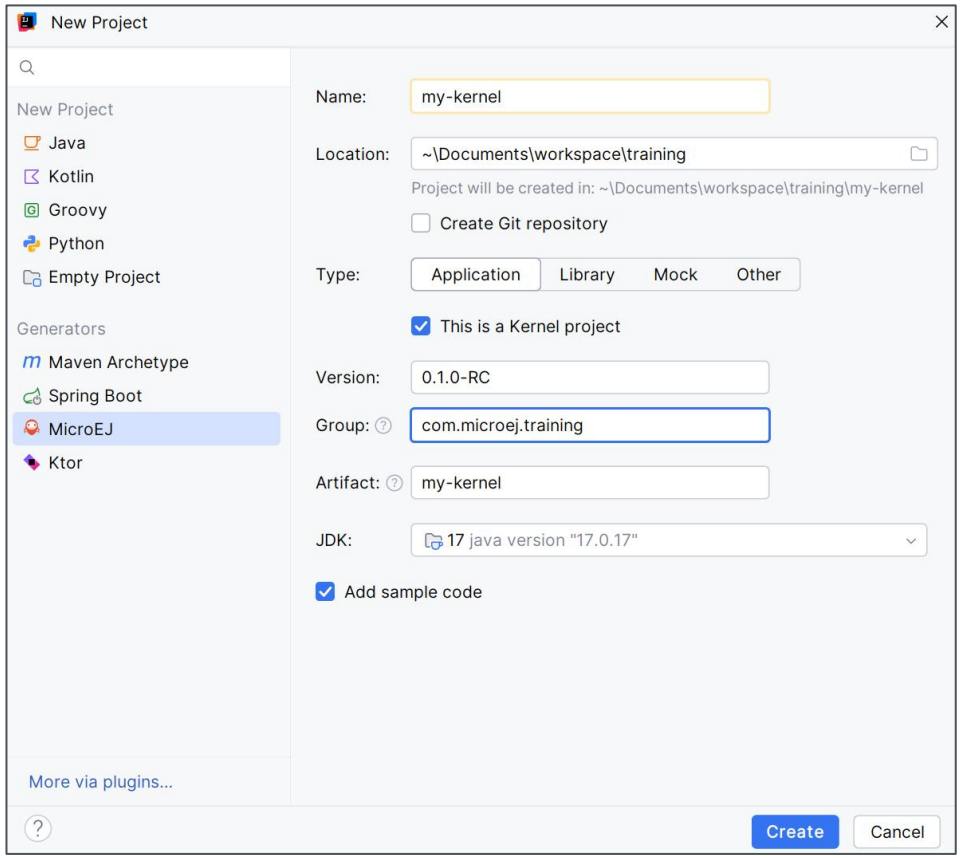
Kernel Project Creation

© MICROEJ 2026

KERNEL PROJECT CREATION

In IntelliJ:

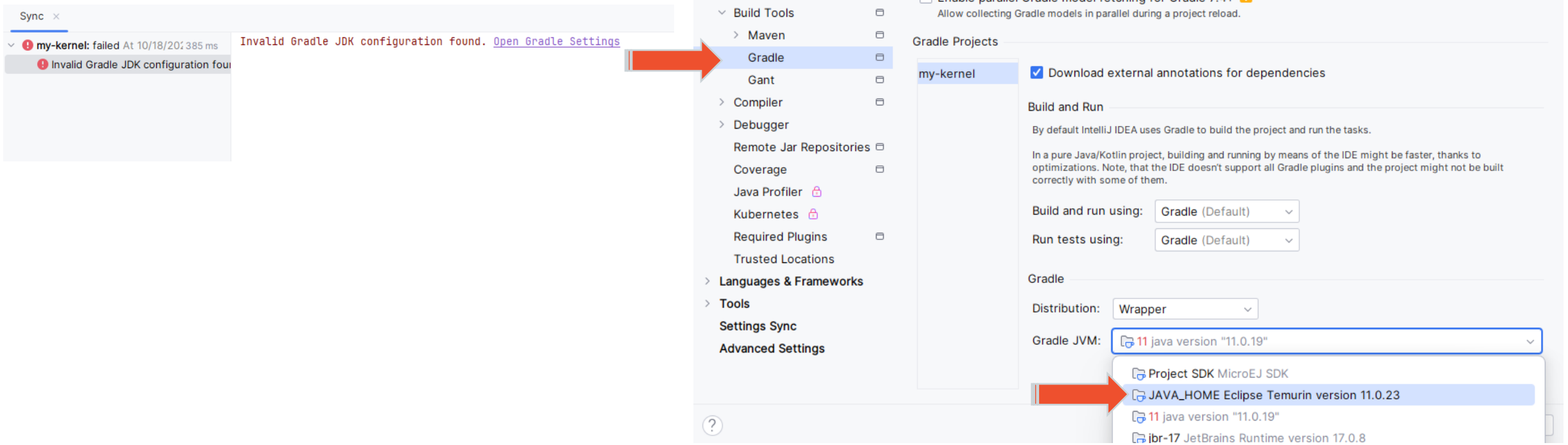
- Go to **File** -> **New** -> **New Project...**
- Select the **MicroEJ project**,
- Change the **Name** and the **Project Location** if needed,
- Select the **Application** type.
- Select **This is a Kernel project**.
- Fill the other input fields.
- Click on **Create**.
- The project opens in a new IDE window.



© MICROEJ 2026

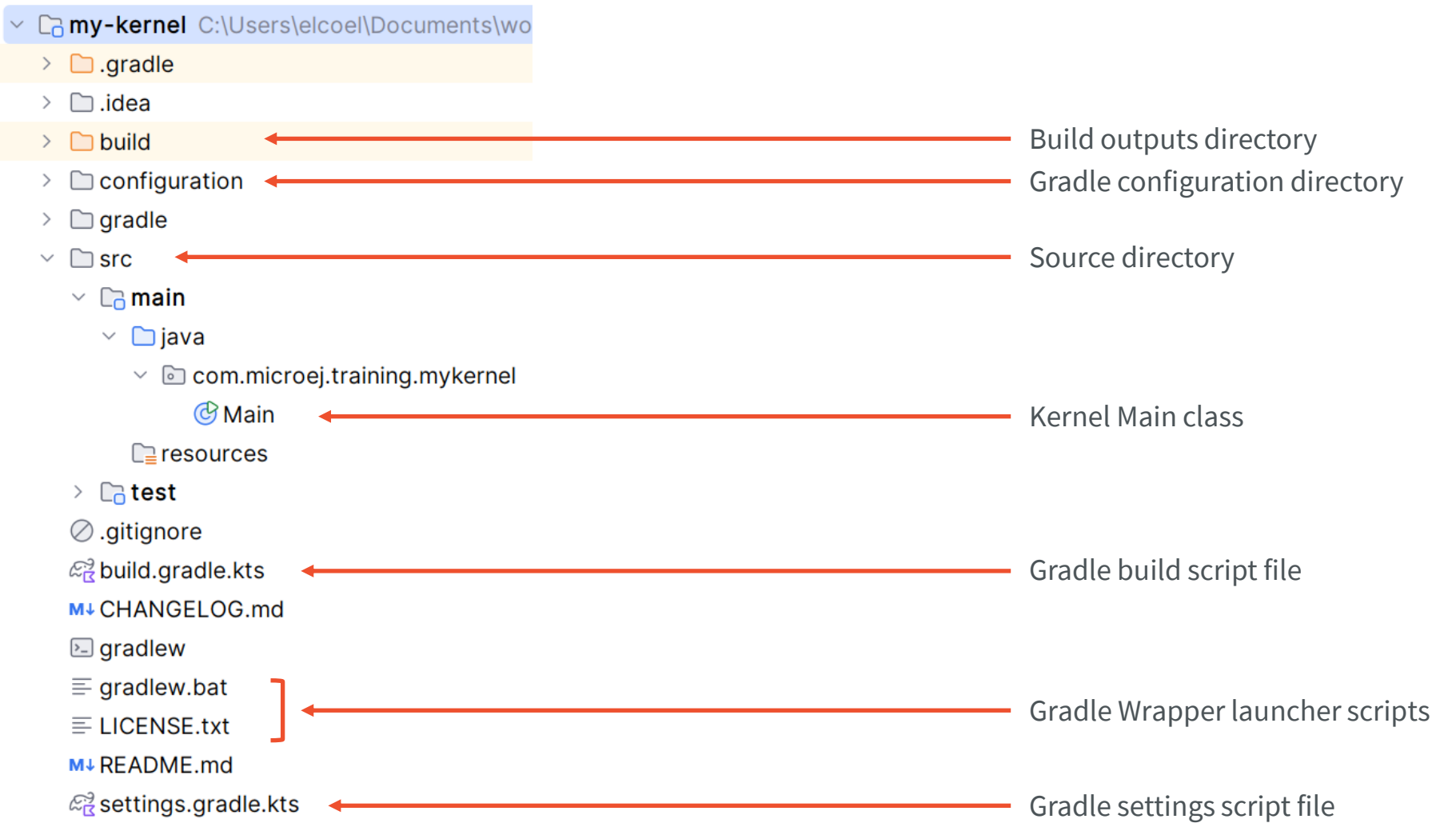
(OPTIONAL) SELECT JDK LOCATION

If the following error shows up, select the JDK installed in the **JAVA_HOME** location:



© MICROEJ 2026

PROJECT STRUCTURE (INTELLIJ)



© MICROEJ 2026

DEFAULT KERNEL BEHAVIOR

- Prints the Kernel name
- Automatically starts already installed Features:

```
public class Main {

   public static void main(String[] args) {
      String kernelName = Kernel.getInstance().getName();
      System.out.println(kernelName + " Hello World!");
      for (Feature feature : Kernel.getAllLoadedFeatures()) {
         System.out.println("=> Starting Feature " + feature.getName());
         feature.start();
      }
   }
}
```

- See ej.kf.Kernel Javadoc API for more details

© MICROEJ 2026

VEE PORT SELECTION

The Kernel project requires a project in order to be built:

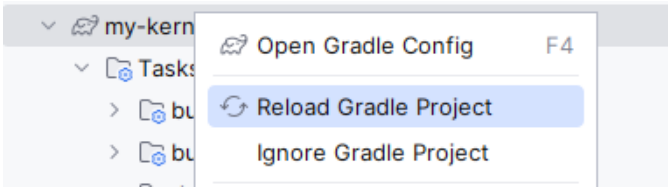
- Get the path to the **NXP i.MX RT1170 VEE Port** (e.g. C:\workspaces\training\nxpvee-mimxrt1170-evk\nxpvee-mimxrt1170-evk)
- Add the path to the VEE Port in the **settings.gradle.kts** file of the **my-kernel** project:

```
rootProject.name = "my-kernel"

includeBuild("C:\\workspaces\\training\\nxpvee-mimxrt1170-evk\\nxpvee-mimxrt1170-evk")
```

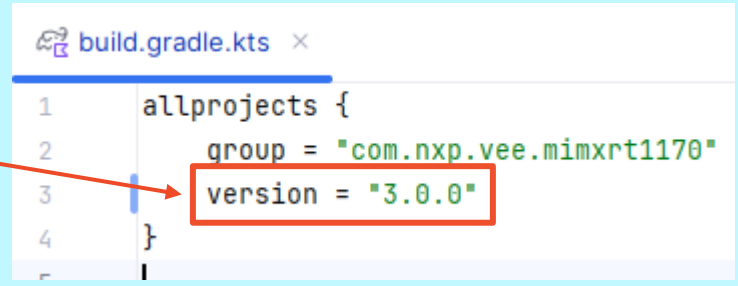
- In the **build.gradle.kts** file of the **my-kernel** project, add the dependency to the VEE Port:

```
dependencies {
    implementation("ej.api:edc:1.3.7")
    implementation("ej.api:kf:1.7.0")
    implementation("com.microej.kernelapi:edc:1.2.0")

    //Uncomment the microejVee dependency to set the VEE Port or Kernel to use
    //microejVee("com.mycompany:myvee:1.0.0")
    microejVee("com.nxp.vee.mimxrt1170:vee-port:3.0.0")

}
```

- Once done, reload the **my-kernel** Gradle project:



The version of the VEE port can be checked in the **nxpvee-mimxrt1170-evk\build.gradle.kts** file:



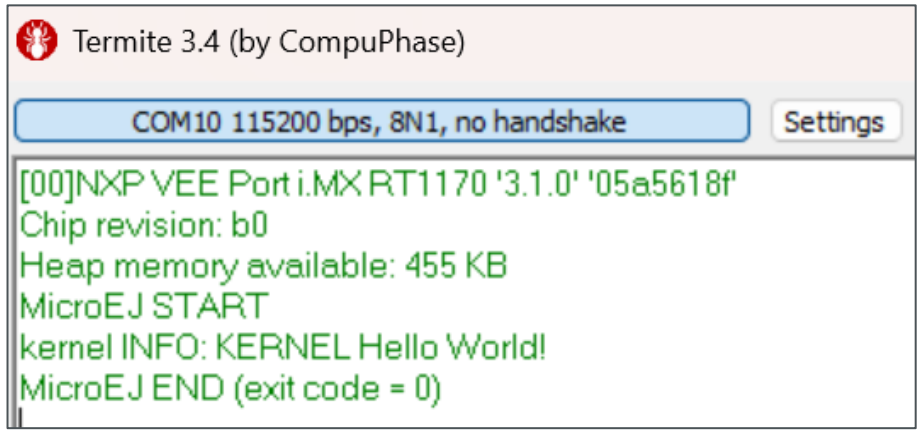
© MICROEJ 2026

RUN THE KERNEL ON DEVICE

- Open the **Gradle** view.
- Open **my-kernel > tasks > microej** and double-click on **runOnDevice**.

Once the Kernel is built and flashed on the device:

- Open the Termite serial terminal.
- Click the **Settings** button.
- Select the NXP i.MX RT1170 EVK board COM port.
- Reset the NXP i.MX RT1170 EVK board using Reset button
- The Kernel starts and the **Hello World** message is printed in the console!



© MICROEJ 2026

Run a Sandboxed Application on the Kernel

Load a Feature from an SD Card

© MICROEJ 2026

FEATURES CONCEPTS

- A Sandboxed Application is an Application that can run over a Kernel.

- The build output file of a Sandboxed Application against a Kernel is called a Feature (`.fo` file).

- By default, a .fo file is specific to a Kernel:
  - It can only be installed on the Kernel it has been linked to.
  - Rebuilding a Kernel implies to build the Feature again.

- Feature Portability can be enabled to use a same Feature on different Kernels.
  Some portability rules need to be respected to make it possible.
  Feature portability is not demonstrated in this training.

© MICROEJ 2026

FEATURE INSTALLATION (1/2)

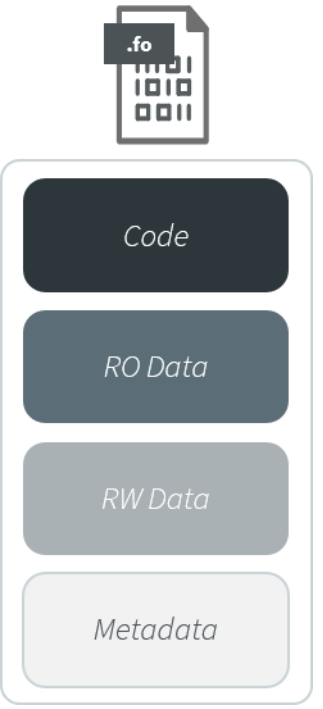
The Feature file (.fo) is retrieved as an **InputStream** by the Kernel.

Feature installation is triggered by a call to the **Kernel.install(InputStream)** method. It consists of the following steps:

- loading Feature's content from .fo file,
- linking Feature's code with the Kernel,
- storing Feature's content into the target memory.

A Feature .fo file is composed of the following elements:

- **Code**: Application code (methods, types, …) as well as built-in objects (strings and immutables),
- **RO Data**: Application Resources that do not require content modification,
- **RW Data**: Reserved memory for Feature execution (Application static fields and Feature internal structures),
- **Metadata**: Temporary information required during the installation phase, such as code relocations.



Feature .fo File Content

© MICROEJ 2026

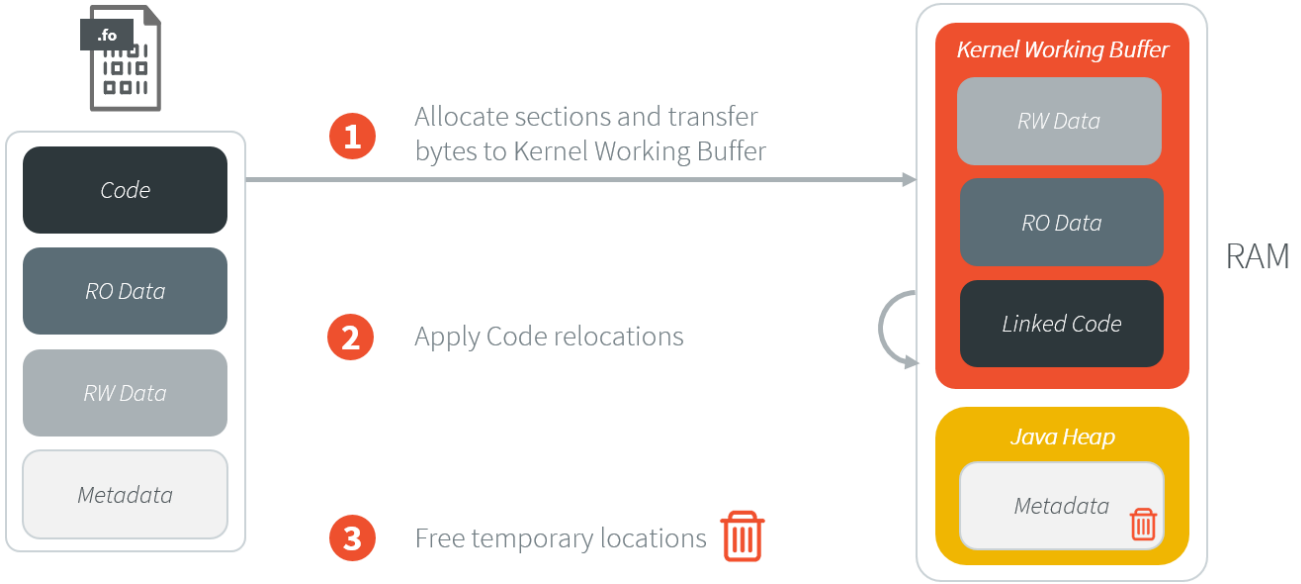
FEATURE INSTALLATION (2/2)

Feature installation flow allows to install Features in any byte-addressable memory mapped to the CPU's address space.
The Feature content is read chunk-by-chunk from the InputStream and progressively transferred to the target memory.

By default, the Feature is loaded and executed from RAM in a Kernel Working Buffer. See more about the installation flow.

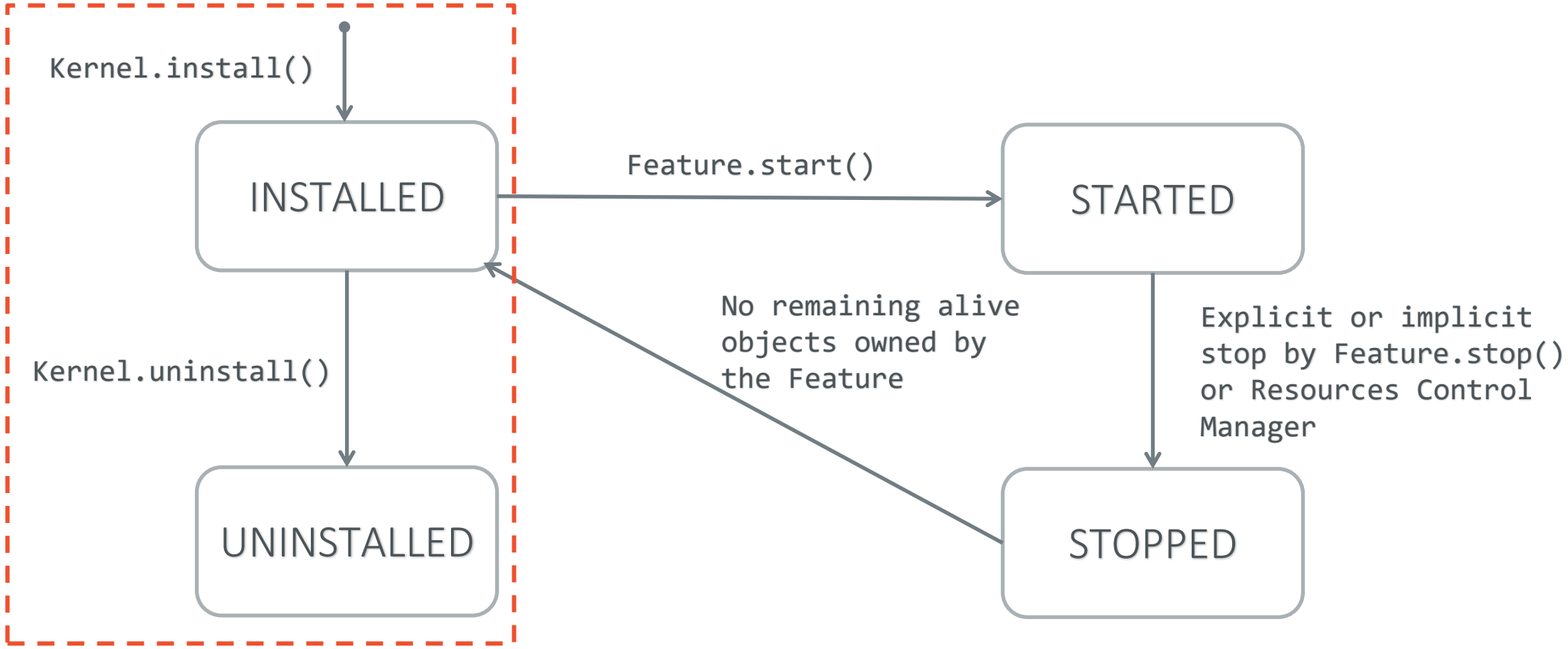
The Feature can also be installed in ROM to allow Feature Persistency, see Custom Installation. This ensures that the Features remain available even after the device restarts.

The LLKernel implementation of the NXP i.MXRT1170 VEE Port executes Features from RAM (in-place installation).



© MICROEJ 2026

FEATURE STATE DIAGRAM



© MICROEJ 2026

LOAD A FEATURE FROM THE SD CARD (1/3)

The NXP i.MXRT1170 VEE Port features a file system interface.

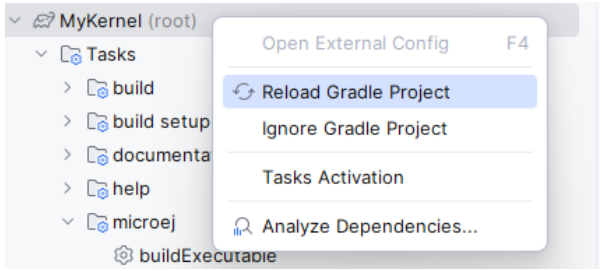
An SD card is used for the storage (should be previously formatted to a FAT32 file system).

Use the FS APIs to load a Feature from the SD Card:

1. Add the FS dependency to the Kernel project (in **build.gradle.kts**):

```
dependencies {
  implementation("ej.api:edc:1.3.7")
  implementation("ej.api:kf:1.7.0")
  implementation("ej.api:fs:2.1.1")
...
```

2. Save the **build.gradle.kts** file and reload the Gradle project:



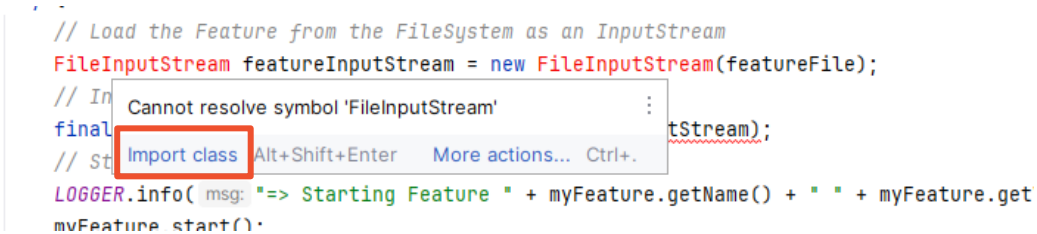
© MICROEJ 2026

LOAD A FEATURE FROM THE SD CARD (2/3)

3. Add the following code in the **Main.java** class:

```
public class Main {

  private static final String FEATURE_PATH = "/application.fo";

  public static void main(String[] args) {
    String kernelName = Kernel.getInstance().getName();
    System.out.println(kernelName + " Hello World!");
    for (Feature feature : Kernel.getAllLoadedFeatures()) {
      System.out.println("=> Starting Feature " + feature.getName());
      feature.start();
    }

  File featureFile = new File(FEATURE_PATH);

    try {
      // Load the Feature from the FileSystem as an InputStream
      FileInputStream featureInputStream = new FileInputStream(featureFile);
      // Install the feature
      final Feature myFeature = Kernel.install(featureInputStream);
      // Start the installed feature
     System.out.println("=> Starting Feature " + myFeature.getName() + " " +
     myFeature.getVersion() );
      myFeature.start();

      // Wait for 3 seconds and stop it
      Thread.sleep(3000);

      System.out.println("Stopping " + myFeature.getName());
      myFeature.stop();
    }
```

4. Import all the necessary classes.
   You can do it automatically by moving the mouse over the errors and click on **Import class**



5. Add a catch clause to handle exceptions. You can do it automatically by moving the mouse over the errors and click on **Add "catch" clause**

LOAD A FEATURE FROM THE SD CARD (3/3)

Several exceptions should be handled:

- **FileNotFoundException:**
  - Throw if the file is not found on SD card.
- **IOException**:
  - Thrown if an error occurred during Feature file load.
- **IncompatibleFeatureException**:
  - Thrown if the Feature being loaded is not compatible with the current Kernel.
- **InvalidFormatException**:
  - Thrown when a Feature data being loaded has an invalid format.
- **InterruptedException**:
  - If any thread has interrupted the current thread
- **AlreadyLoadedFeatureException**:
  - Thrown if the Feature being loaded has already been loaded.

```
try {
   // Load the Feature from the FileSystem as an InputStream
   FileInputStream featureInputStream = new FileInputStream(featureFile);
   // Install the feature
   final Feature myFeature = Kernel.install(featureInputStream);
   // Start the installed feature
  System.out.println("=> Starting Feature " + myFeature.getName() + " " +
  myFeature.getVersion() );
   myFeature.start();

   // Wait for 3 seconds and stop it
   Thread.sleep(3000);

   System.out.println("Stopping " + myFeature.getName());
   myFeature.stop();
} catch (FileNotFoundException e) {
   System.out.println("Feature file not found on the SD Card");
} catch (IOException e) {
   System.out.println("Feature loading error");
   throw new RuntimeException(e);
} catch (IncompatibleFeatureException e) {
   System.out.println("Feature not compatible with the Kernel");
} catch (InvalidFormatException e) {
   System.out.println("Wrong Feature format");
} catch (InterruptedException e) {
   throw new RuntimeException(e);
} catch (AlreadyLoadedFeatureException e) {
   System.out.println("Feature has already been loaded");
}
```

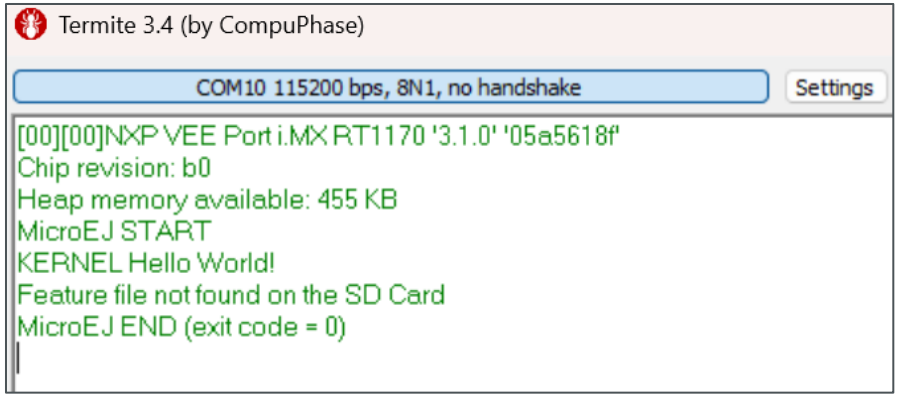
© MICROEJ 2026

RUN THE KERNEL ON DEVICE

- Open the **Gradle** view.
- Open **my-kernel > tasks > microej** and double-click on **runOnDevice**.
- Insert the SD Card in the NXP i.MXRT1170

Once the Kernel is built and flashed on the device:

- Open the Termite serial terminal.
- Click the **Settings** button.
- Select the NXP i.MX RT1170 EVK board COM port.
- Reset the NXP i.MX RT1170 EVK board using Reset button
- The Kernel starts and an exception is thrown. **The Feature file is not found.**

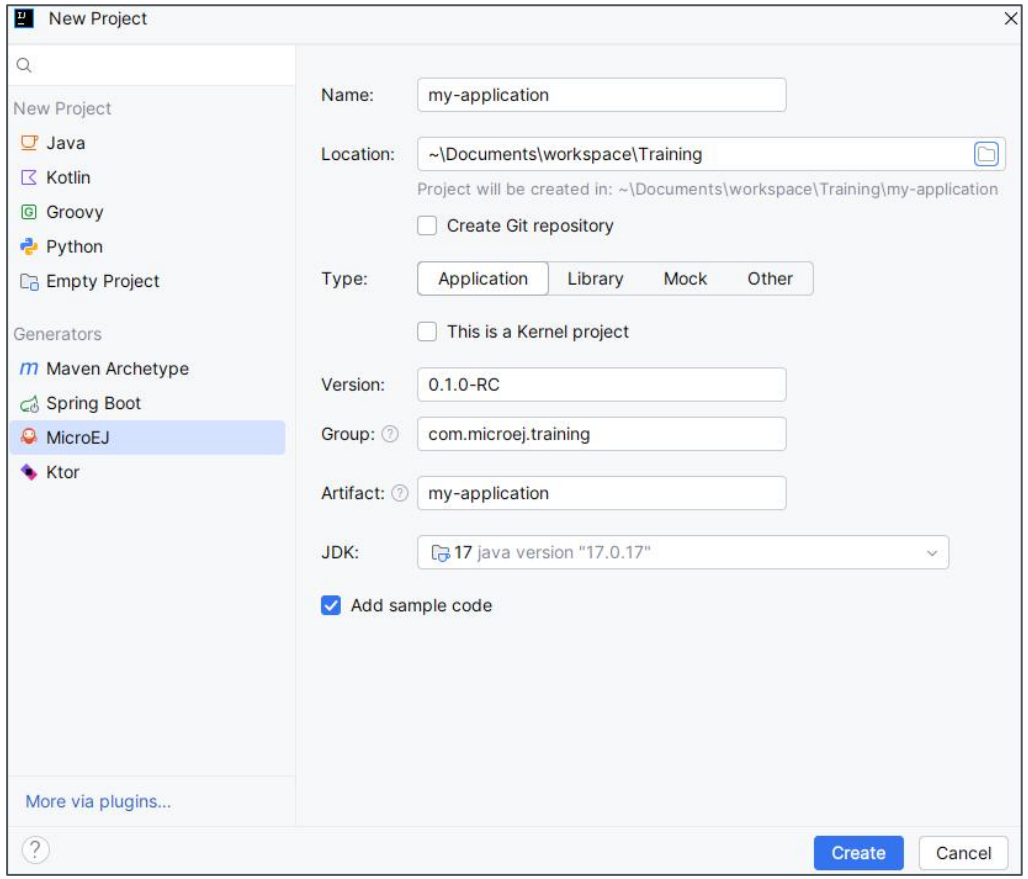


© MICROEJ 2026

CREATE AN APPLICATION PROJECT

In IntelliJ:

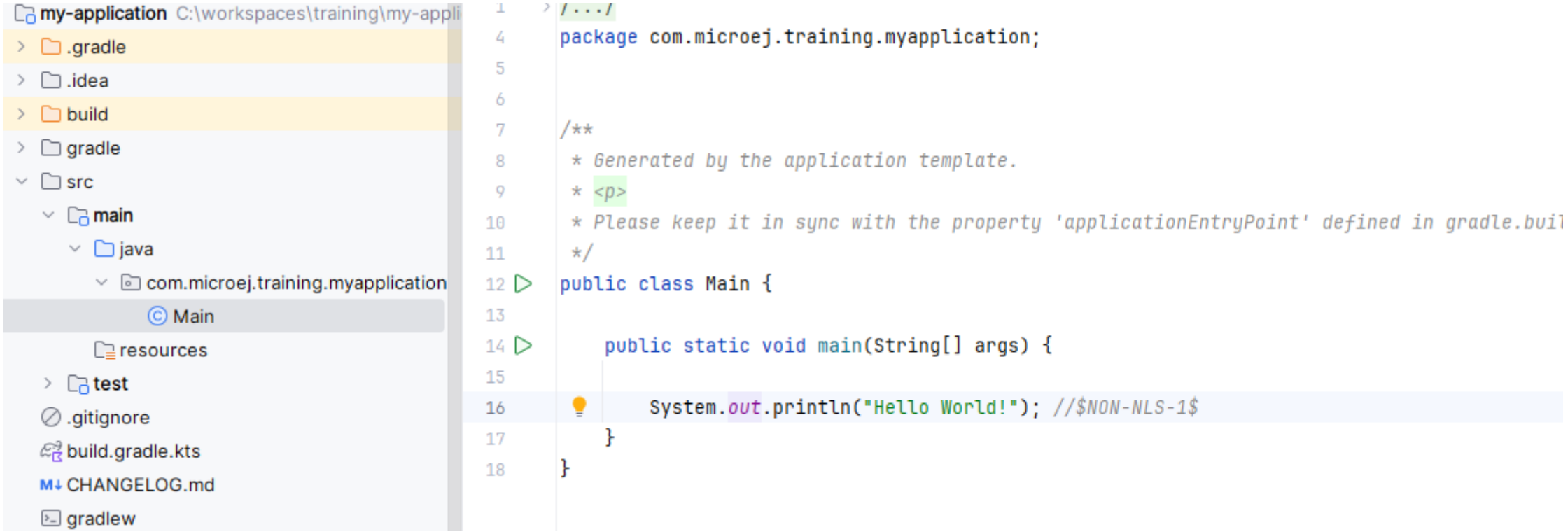
- Go to **File** -> **New** -> **New Project...**
- Select the **MicroEJ project**,
- Change the **Name** and the **Project Location** if needed,
- Select the **Application** type.
- Fill the other input fields.
- Click on **Create**.
- The project opens in a new IDE window.



© MICROEJ 2026

DEFAULT APPLICATION BEHAVIOR

By default, the application prints a **Hello World** message when starting:



© MICROEJ 2026

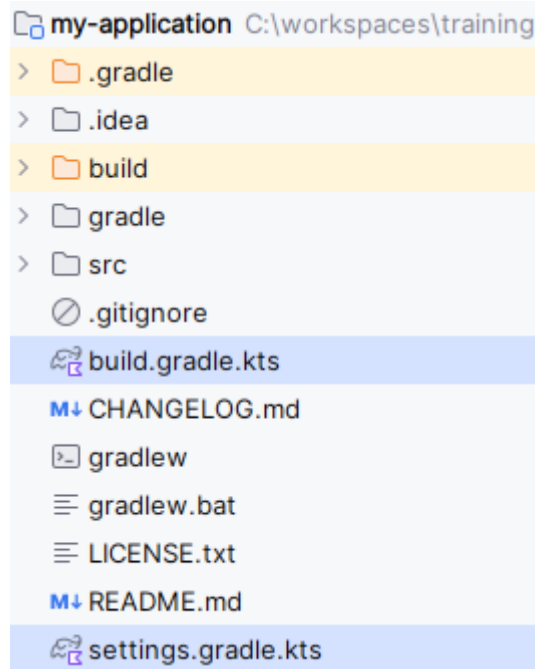
SELECT A KERNEL

Building the Feature file of an Application with the SDK requires a Kernel.

Add the **my-kernel** project as a sub-project:

- Get the absolute path of the **my-kernel** project (e.g. C:\workspaces\training\my-kernel)
- Add this path to the **settings.gradle.kts** file of the **my-application** project

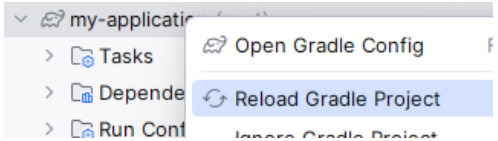
```
rootProject.name = "my-application"

includeBuild("C:\\workspaces\\training\\my-kernel")
```



Declare the **my-kernel** project dependency in the **build.gradle.kts** file of the **my-application** project, with the **microejVee** configuration:

```
dependencies {
  implementation("ej.api:edc:1.3.5")

  //Uncomment the microejVee dependency to set the VEE Port or Kernel to use
  //microejVee("com.mycompany:myvee:1.0.0")
  microejVee("com.microej.training:my-kernel:0.1.0-RC")
}
```

Once done, reload the **my-application** Gradle project:

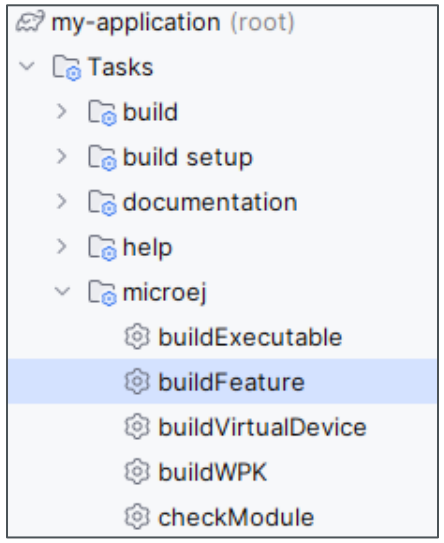


© MICROEJ 2026

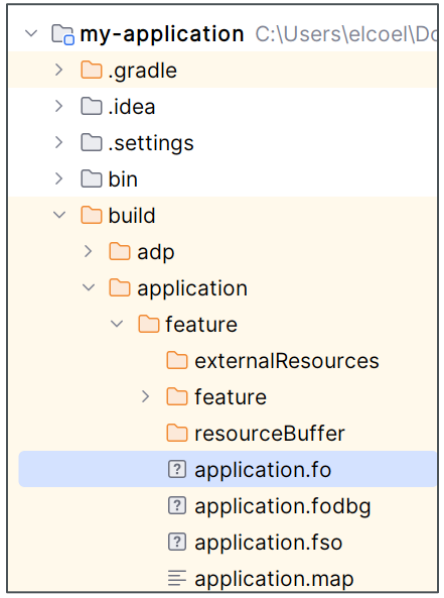
BUILD THE FEATURE FILE

Start now, we'll be able to use only the project "my-application" to modify the Kernel and the Application at the same time.

To build the Feature file (.fo) of an Application, the SDK provides the Gradle **buildFeature** task:



The Feature file is generated in the **build/application/feature** folder of the project:



© MICROEJ 2026

TROUBLESHOOTING

If the kernel is not found when the buildFeature task runs, make sure the following lines are present in the **build.gradle.kts** of the kernel application. This is necessary for the application to find the kernel executable.

build.gradle.kts (:my-kerr

```
4   plugins {
5       id("com.microej.gradle.application") version "1.0.0"
6   }
7
8   group="com.microej.training"
9   version="0.1.0-RC"
10
11  microej {
12      applicationEntryPoint = "com.microej.training.mykernel.Main"
13      produceExecutableDuringBuild()
14      produceVirtualDeviceDuringBuild()
15  }
```

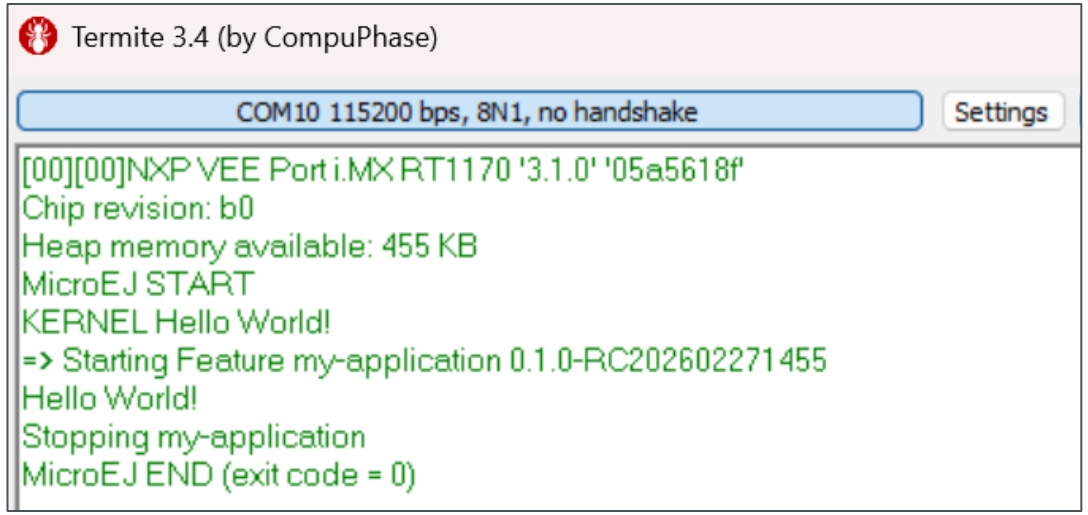
© MICROEJ 2026

RUN ON THE DEVICE

- Copy **application.fo** at the root of a SD Card.
- Insert the SD Card in the NXP i.MXRT1170

Get the traces:

- Open the Termite serial terminal.
- Click the **Settings** button.
- Select the NXP i.MX RT1170 EVK board COM port.
- Reset the NXP i.MX RT1170 EVK board using Reset button
- The Kernel starts, starts and stops the **Feature** !



© MICROEJ 2026

ABOUT THE FEATURE NAME & VERSION (1/2)

By default:

- The feature **name** will be the module name, in our case : **my-application**
- The feature **version** will be the version provided by the **build.gradle. kts** file

© MICROEJ 2026

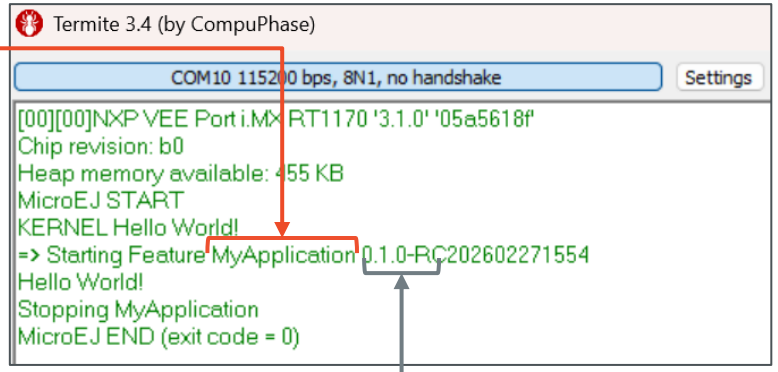
ABOUT THE FEATURE NAME & VERSION (2/2)

In order to override the default feature name, you can add **applicationName** property in **build.gradle.kts** (see documentation).

Here is an example:

build.gradle.kts (my-application)

```
1   /*
2    * Kotlin
3    */
4   plugins {
5       id("com.microej.gradle.application") version "1.5.0"
6   }
7
8   group = "com.microej.training"
9   version = "0.1.0-RC"
10
11  microej {
12      applicationName = "MyApplication"
13      applicationEntryPoint = "com.microej.training.myapplication.Main"
14      // Uncomment to use "prod" architecture when using a VEE Port (defaults to "eval")
15      // architectureUsage = "prod"
16  }
```

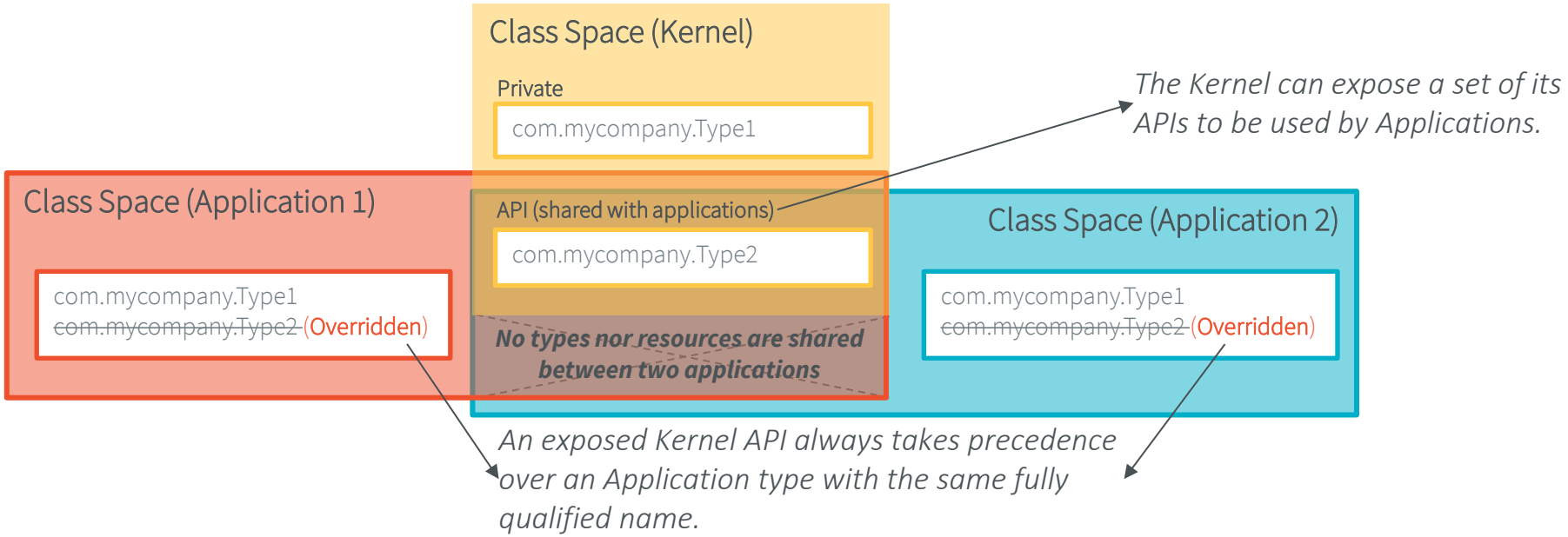


© MICROEJ 2026

Kernel API

© MICROEJ 2026

CLASS SPACES



- Kernel and Applications define their own private class spaces.
- Types and Resources in a class space can be accessed only by the class space owner.
- Communication between two Applications is possible in a controlled manner using **Shared Interfaces** *(to be explained in the Application development Training)*

© MICROEJ 2026

KERNEL API DEFINITION

- By default, no API is exposed by the Kernel from its Class Space except the interface ***ej.kf.FeatureEntryPoint***.

- Kernel types, methods and static fields allowed to be accessed by Features must be declared in a ***kernel.api*** file.

- *A **kernel.api*** file is an XML file (see beside schema). Here is an example showing the exposition of **Type1** and its method **log():**

```
<require>
    <type name="com.mycompany.Type1"/>
    <method name="com.mycompany.Type1.log(java.lang.String)void"/>
</require>
```

  The method **log()** of **Type1** can be invoked by any application.

```
1  <xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>
2      <xs:element name='require'>
3          <xs:complexType>
4              <xs:choice minOccurs='0' maxOccurs='unbounded'>
5                  <xs:element ref='type'/>
6                  <xs:element ref='field'/>
7                  <xs:element ref='method'/>
8              </xs:choice>
9          </xs:complexType>
10     </xs:element>
11     <xs:element name='type'>
12         <xs:complexType>
13             <xs:attribute name='name' type='xs:string' use='required'/>
14         </xs:complexType>
15     </xs:element>
16     <xs:element name='field'>
17         <xs:complexType>
18             <xs:attribute name='name' type='xs:string' use='required'/>
19         </xs:complexType>
20     </xs:element>
21     <xs:element name='method'>
22         <xs:complexType>
23             <xs:attribute name='name' type='xs:string' use='required'/>
24         </xs:complexType>
25     </xs:element>
26 </xs:schema>
27
```

Kernel API XML Schema

© MICROEJ 2026

Hands-on

Use the Logging library in the Kernel and the Application

© MICROEJ 2026

HANDS-ON SUMMARY

1. Import the Logger library in the Kernel project
2. Use it in the Kernel project

3. Import the Logger library in the Application project
4. Use it in the Application project
5. Run on Device

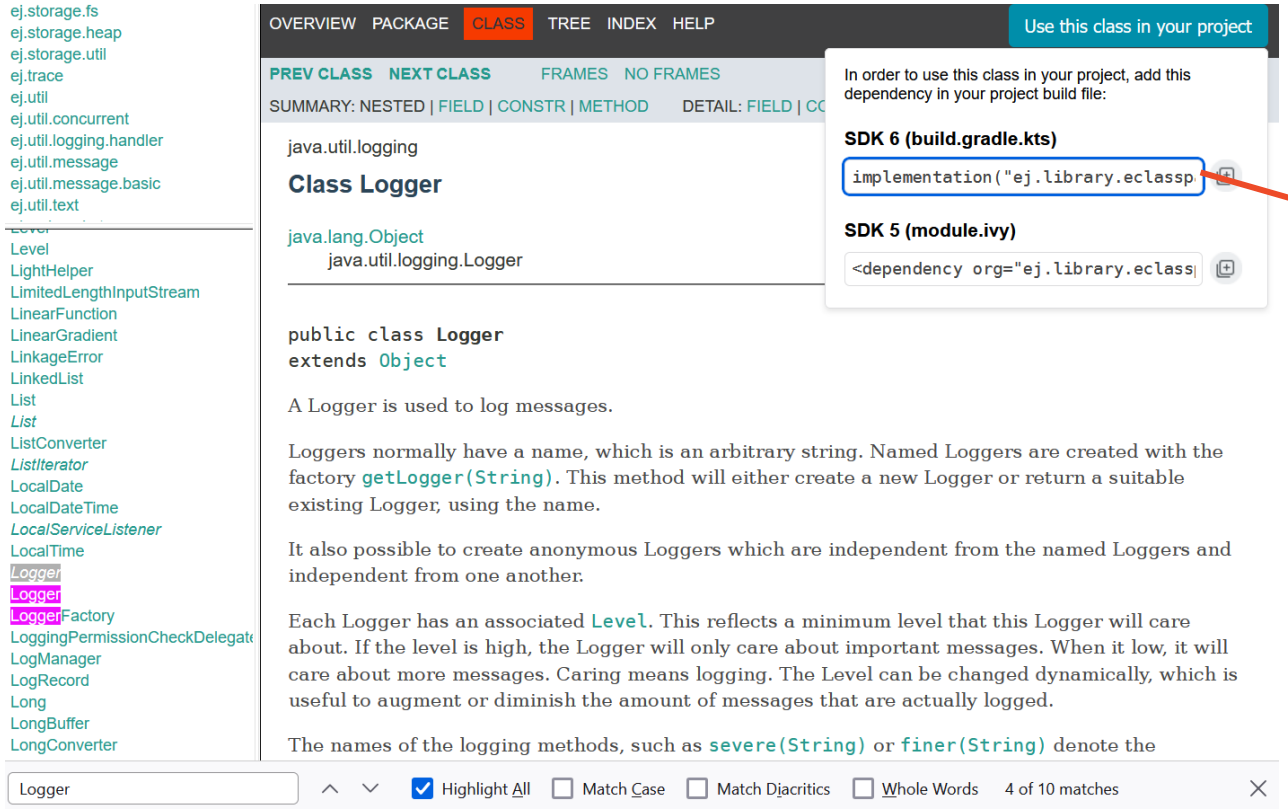
6. Make the Logger class belong to the Kernel
7. Expose the Logger library to Applications
8. Run on Device

9. Footprint considerations

© MICROEJ 2026

IMPORT THE LOGGER LIBRARY IN THE KERNEL

- Look for the Logger library in the Javadoc.
- Search for the **java.util.logging.Logger** class.
- Add the dependency line to the **build.gradle.kts** file of the **my-kernel** project.
- Reload the **my-kernel** Gradle project.




```
dependencies {
    implementation("ej.api:edc:1.3.7")
    implementation("ej.api:kf:1.7.0")
    implementation("ej.api:fs:2.1.1")
    implementation("com.microej.kernelapi:edc:1.2.0")
    implementation("ej.library.eclasspath:logging:1.2.1")
```

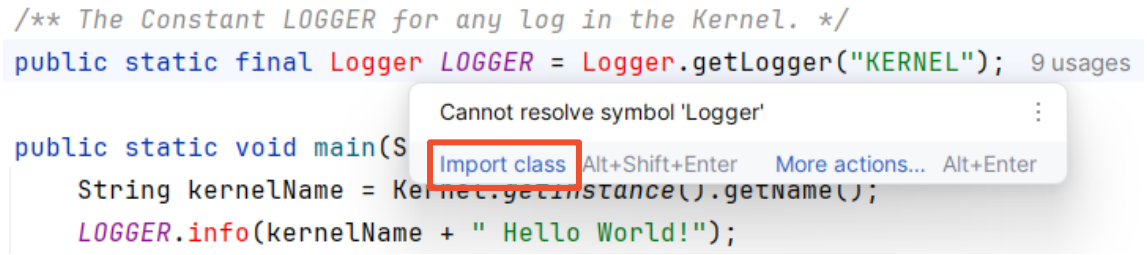
© MICROEJ 2026

USE THE LOGGER IN THE KERNEL PROJECT

Replace all calls to **System.out.println()** by:

- **LOGGER.info()** for informational messages (e.g. Kernel startup message).
- **LOGGER.severe()** for indicating a serious failure (e.g. Exception message).

Import the class Logger (can be done automatically by the IDE, see example on IntelliJ) :




```
public class Main {

   private static final String FEATURE_PATH = "/application.fo";

   /** The Constant LOGGER for any log in the Kernel. */
   public static final Logger LOGGER = Logger.getLogger("KERNEL");

   public static void main(String[] args) {
      String kernelName = Kernel.getInstance().getName();
      LOGGER.info(kernelName + " Hello World!");
      for (Feature feature : Kernel.getAllLoadedFeatures()) {
         LOGGER.info("=> Starting Feature " + feature.getName());
         feature.start();
      }

      File featureFile = new File(FEATURE_PATH);

      try {
         // Load the Feature from the FileSystem as an InputStream
         FileInputStream featureInputStream = new
FileInputStream(featureFile);
         // Install the feature
         final Feature myFeature = Kernel.install(featureInputStream);
         // Start the installed feature
         LOGGER.info("Starting: " + myFeature.getName());
         myFeature.start();

         // Wait for 3 seconds and stop it
         Thread.sleep(3000);

         LOGGER.info("Stopping " + myFeature.getName());
         myFeature.stop();
      } catch (FileNotFoundException e) {
         LOGGER.severe("Feature file not found on the SD Card");
      } catch (InvalidFormatException e) {
   ...
```

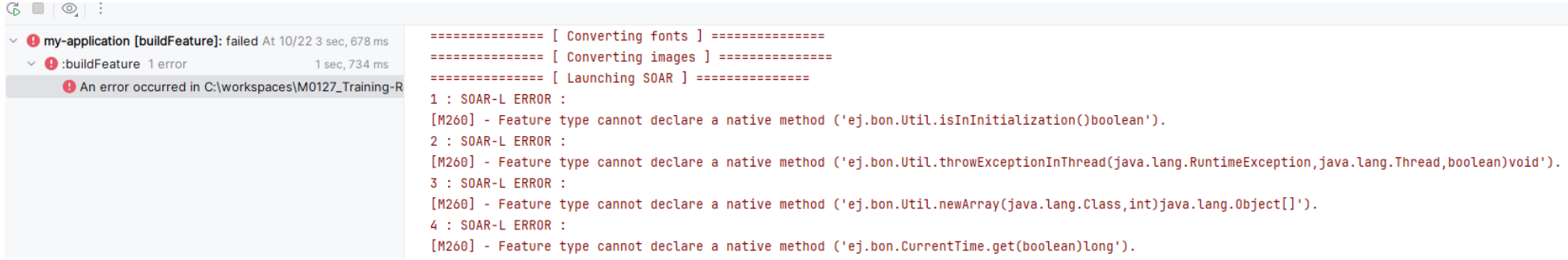
© MICROEJ 2026

USE THE LOGGER IN THE APPLICATION PROJECT

- Add the same logging dependency line to the **build.gradle.kts** file of the **my-application** project.
- Reload the **my-application** Gradle project.
- Replace the call to **System.out.println()** by **LOGGER.info():**

```
public class Main {

   public static final Logger LOGGER = Logger.getLogger("MY-APPLICATION");

   public static void main(String[] args) {

      LOGGER.info("Hello World!"); //$NON-NLS-1$
   }
}
```

- Import the Logger class
- Run the **buildFeature** task.
- The following build errors can be observed:



© MICROEJ 2026

NATIVE METHOD DECLARATION (1/2)

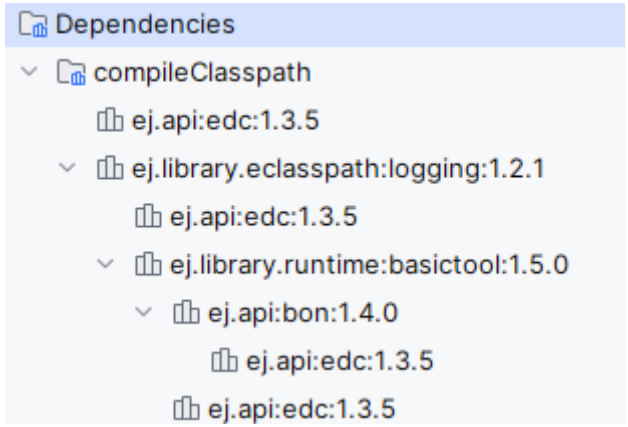
The KF specification defines several rules regarding the interactions between Kernel and Features.

One of them is the Native Method Declaration rule: a class owned by a Feature cannot declare a native method.

→ A SOAR ERROR occurs during the application build because the dependencies used by the logging library are declaring native methods

More precisely:

- The logging library relies on the following dependencies (can be seen in the Gradle task view: **my-application > Dependencies > compileClasspath**):



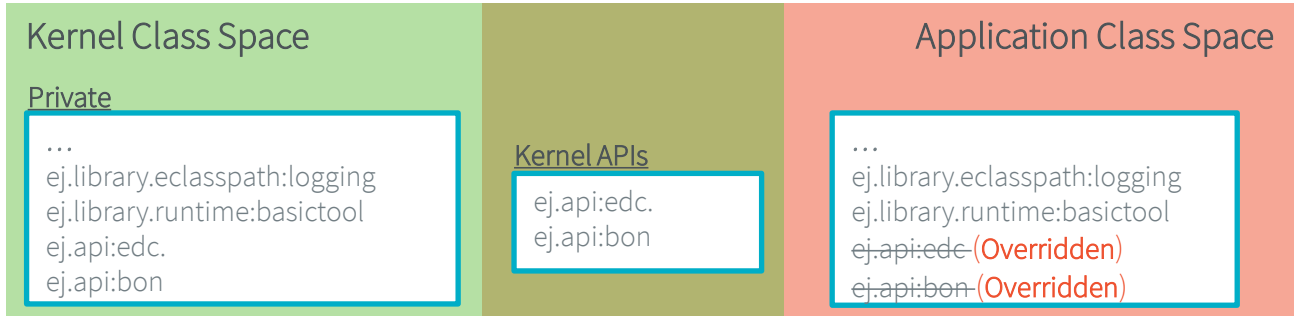
- The EDC and the BON are Foundation Libraries. They are declaring several native methods that require to be implemented through a native interface (SNI).

© MICROEJ 2026

NATIVE METHOD DECLARATION (2/2)

- The **logging** library is used in the Kernel and the Application projects. Consequently, the EDC and BON libraries are also part of the Application class space:

- The application build fails because this check is performed at compile time.
- To avoid that, the Kernel should provide **Kernel APIs** for **EDC** and **BON** to allow the application to access the APIs of those libraries.
- Defining **Kernel APIs** for **EDC** and **BON** will make EDC and BON belong to the kernel and will allow the access to applications:



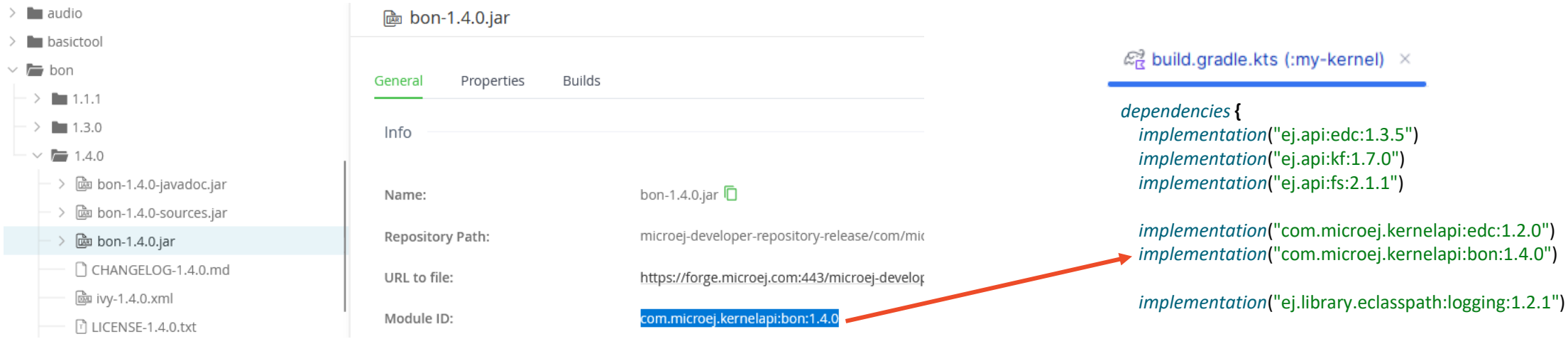
© MICROEJ 2026

ADD EDC & BON KERNEL APIS TO THE KERNEL

EDC and BON Kernel APIs should be added to the Kernel project in order to build the application project.

MicroEJ provides predefined Kernel API files for the most common libraries provided by a Kernel. These files are packaged as MicroEJ modules in the Developer Repository under the com/microej/kernelapi organization.

- Add the **EDC** and **BON Kernel APIs** in the **build.gradle.kts** file of the **Kernel** project:



- Note that the EDC Kernel API was already included. It is included by default to allow the applications to use the minimal Java standard runtime environment.

© MICROEJ 2026

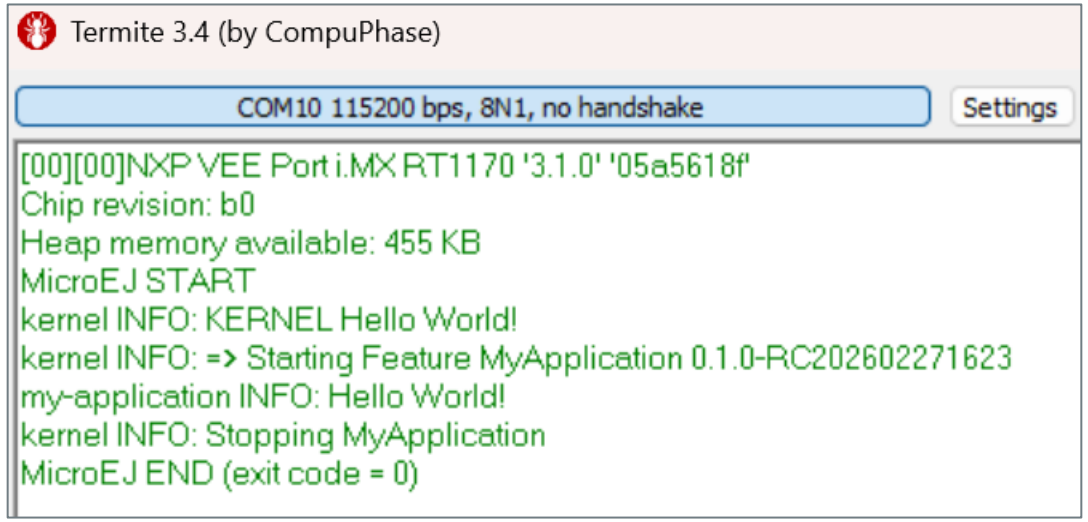
RUN ON THE DEVICE

- Run the **my-kernel > runOnDevice** task to program the updated Kernel on the device.
- Run **my-application > buildFeature** task. The build is now successful!

- Copy **application.fo** at the root of a SD Card.
- Insert the SD Card in the NXP i.MXRT1170

Get the traces:

- Open the Termite serial terminal.
- Click the **Settings** button.
- Select the NXP i.MX RT1170 EVK board COM port.
- Reset the NXP i.MX RT1170 EVK board using Reset button
- Logs are now using the Logger (traces are prefixed by the logger name / level)



© MICROEJ 2026

FOOTPRINT CONSIDERATIONS

When a type (here the Logging library) is used by both the Kernel and one or more Sandboxed Applications there are two possibilities:

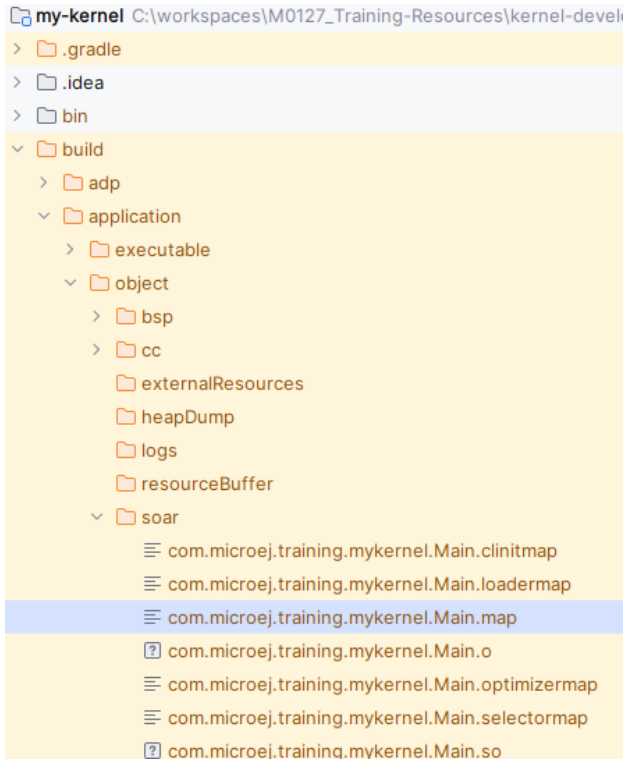
1) **The type is not declared as a Kernel API**
   In this case, the type code is duplicated in the Kernel and all the applications using it.
   This allows **the usage of different versions** of the same type in the Kernel and applications but **increases the final code size**.

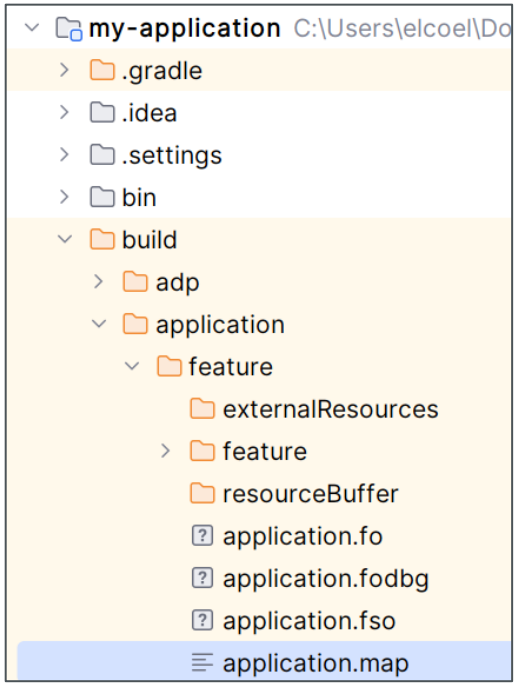
2) **The type is declared as a Kernel API**
   In this case, the type only exists in the Kernel code and will not be duplicated in applications code.
   Only one version of the type can be used but the **code size is not increased** due to the code duplication.
   Updating the type involves updating the whole Kernel.

© MICROEJ 2026

CHECK THE APPLICATION & KERNEL FOOTPRINT (1/2)

When the project Executable is built, a Memory Map file is generated.
.map files are generated for **my-kernel** and **my-application**:





These files can be visualized with the Memory Map Analyzer, an Eclipse IDE plugin. It displays the memory consumption of different features in the RAM and ROM.

© MICROEJ 2026

CHECK THE APPLICATION & KERNEL FOOTPRINT (2/2)

- Open **my-application.map** (in **my-application\build\application\feature**) with a text editor
- Add this line after the first line :

```
<lscmap name="default">
<property name="jpf.dir" value="absolute\path\to\my-kernel\build\vee"/>
```

- Install the Eclipse IDE with the required plugin, then launch it.
- In Eclipse IDE, click on **File > Open File...** to open the .map files :

my-kernel\build\application\object\soar\com.microej.training.mykernel.Main.map

com.microej.training.mykernel.Main.map

Name	Image Size	Runtime Size
All	134.0 KB	271.6 KB
ApplicationCode	690 B	0 B
ApplicationImmutables	360 B	0 B
ApplicationStrings	9.3 KB	0 B
BSP	664 B	195.2 KB
ClassNames	391 B	0 B
CoreEngine	596 B	7.5 KB
CoreEngineAllocator	0 B	68.0 KB
LibAddonBasictool	1.0 KB	0 B
LibAddonLogging	1.7 KB	0 B
LibFoundationBON	5.3 KB	0 B
LibFoundationEDC	61.9 KB	0 B
LibFoundationFS	3.8 KB	0 B
LibFoundationKF	19.3 KB	0 B
LibFoundationSNI	310 B	0 B
NativeStackMicroUI	616 B	540 B
RuntimeTables	14.5 KB	0 B
Statics	0 B	394 B
Types	13.2 KB	0 B

my-application\build\application\feature\my-application.map

application.map

Name	Image Size	Runtime Size
All	5.2 KB	99 B
ApplicationCode	171 B	64 B
ApplicationImmutables	404 B	0 B
ApplicationStrings	980 B	0 B
ClassNames	50 B	0 B
CoreEngine	20 B	0 B
LibAddonBasictool	1.0 KB	0 B
LibAddonLogging	1.7 KB	0 B
LibFoundationKF	164 B	35 B
RuntimeTables	112 B	0 B
Types	532 B	0 B

The code of the **Logging** library is duplicated in the Kernel and the Application.

ADD THE LOGGING KERNEL API

- Add the logging Kernel API in the **build.gradle.kts** of **my-kernel**:

build.gradle.kts (:my-kernel)

```
dependencies {
    implementation("ej.api:edc:1.3.5")
    implementation("ej.api:kf:1.7.0")
    implementation("ej.api:fs:2.1.1")
    implementation("ej.library.eclasspath:logging:1.2.1")

    implementation("com.microej.kernelapi:edc:1.1.0")
    implementation("com.microej.kernelapi:bon:1.4.0")
    implementation("com.microej.kernelapi:logging:1.0.0")
```

- The **logging Kernel API** contains a **kernel.api** file that defines all the types of the logging library that can be used by Features:

- Run the **my-application > buildFeature** task.
  It will rebuild the **my-kernel** and **my-application** Executables.

```
<require>
    <type name="ej.util.logging.handler.DefaultHandler"/>
    <method name="ej.util.logging.handler.DefaultHandler.DefaultHandler()void"/>
    <method name="ej.util.logging.handler.DefaultHandler.close()void"/>
    <method name="ej.util.logging.handler.DefaultHandler.flush()void"/>
    <method name="ej.util.logging.handler.DefaultHandler.publish(java.util.logging.LogRecord)void"/>

    <type name="java.util.logging.Handler"/>
    <method name="java.util.logging.Handler.Handler()void"/>
    <method name="java.util.logging.Handler.close()void"/>
    <method name="java.util.logging.Handler.flush()void"/>
    <method name="java.util.logging.Handler.publish(java.util.logging.LogRecord)void"/>

    <type name="java.util.logging.Level"/>
    <field name="java.util.logging.Level.ALL"/>
    <field name="java.util.logging.Level.CONFIG"/>
    <field name="java.util.logging.Level.FINE"/>
    <field name="java.util.logging.Level.FINER"/>
    <field name="java.util.logging.Level.FINEST"/>
    <field name="java.util.logging.Level.INFO"/>
    <field name="java.util.logging.Level.OFF"/>
    <field name="java.util.logging.Level.SEVERE"/>
    <field name="java.util.logging.Level.WARNING"/>
    <method name="java.util.logging.Level.Level(java.lang.String,int)void"/>
    <method name="java.util.logging.Level.getName()java.lang.String"/>
    <method name="java.util.logging.Level.intValue()int"/>
    <method name="java.util.logging.Level.toString()java.lang.String"/>

    <type name="java.util.logging.LogManager"/>
    <method name="java.util.logging.LogManager.LogManager()void"/>
    <method name="java.util.logging.LogManager.addLogger(java.util.logging.Logger)boolean"/>
    <method name="java.util.logging.LogManager.getLogManager()java.util.logging.LogManager"/>
    <method name="java.util.logging.LogManager.getLogger(java.lang.String)java.util.logging.Logger"/>
    <method name="java.util.logging.LogManager.getLoggerNames()java.util.Enumeration"/>
```

Extract of Logging kernel.api

© MICROEJ 2026

CHECK THE FOOTPRINT IMPACT

- Edit **my-application.map** file with the previously mentioned property
- Check the .map files of **my-kernel** and **my-application**

my-kernel\build\application\object\soar\com.microej.training.mykernel.Main.map

com.microej.training.mykernel.Main.map

Name	Image Size	Runtime Size
All	136.3 KB	271.6 KB
ApplicationCode	654 B	0 B
ApplicationImmutables	360 B	0 B
ApplicationStrings	9.4 KB	0 B
BSP	664 B	195.2 KB
ClassNames	391 B	0 B
CoreEngine	596 B	7.5 KB
CoreEngineAllocator	0 B	68.0 KB
LibAddonBasictool	1.8 KB	0 B
LibAddonLogging	2.2 KB	0 B
LibFoundationBON	5.3 KB	0 B
LibFoundationEDC	61.9 KB	0 B
LibFoundationFS	3.8 KB	0 B
LibFoundationKF	19.6 KB	0 B
LibFoundationSNI	310 B	0 B
NativeStackMicroUI	616 B	540 B
RuntimeTables	15.0 KB	0 B
Statics	0 B	394 B
Types	13.4 KB	0 B

my-application\build\application\feature\my-application.map

application.map

Name	Image Size	Runtime Size
All	1.2 KB	51 B
ApplicationCode	123 B	16 B
ApplicationImmutables	404 B	0 B
ApplicationStrings	456 B	0 B
CoreEngine	20 B	0 B
LibFoundationKF	164 B	35 B
RuntimeTables	48 B	0 B
Types	84 B	0 B

The code of the **Logging** library is only embedded in the Kernel code.

Note that the size of the library has increased: **2.2kB instead of 1.7kB**. That's because **all the types** of the Logging library are **embedded in the Logging library Kernel API**.
A custom kernel.api file could be used instead of the Logging Kernel API provided by MicroEJ, to only embed the required Logging library types used in your Kernel / Application.

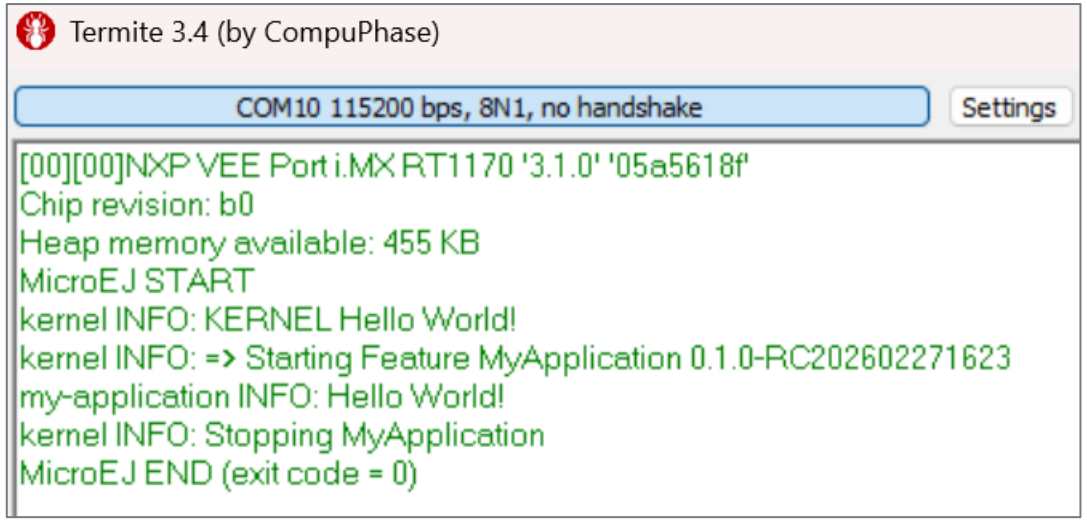
© MICROEJ 2026

RUN ON THE DEVICE

- Run the **my-kernel > runOnDevice** task to program the updated Kernel on the device.
- Run **my-application > buildFeature** task.
- Copy **application.fo** at the root of a SD Card.
- Insert the SD Card in the NXP i.MXRT1170

Get the traces:

- Open the Termite serial terminal.
- Click the **Settings** button.
- Select the NXP i.MX RT1170 EVK board COM port.
- Reset the NXP i.MX RT1170 EVK board using Reset button
- **The logs are identical.**



© MICROEJ 2026

Kernel and Features Communication

© MICROEJ 2026

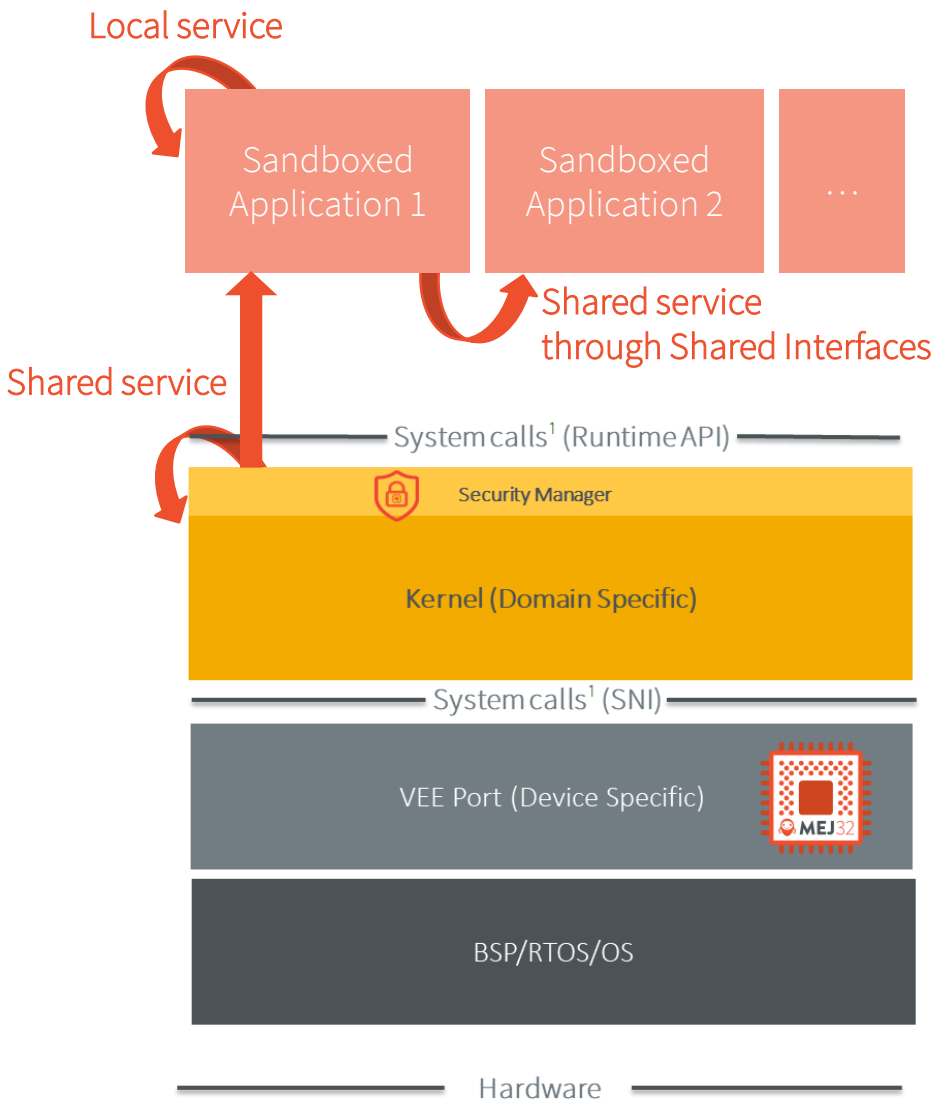
SERVICES

WHAT IS A SERVICE?

A service is a functionality made available by a software component to perform a particular task defined by a Java interface.

There are 3 types of services available:

1. **Local services**: provided and used by the application/kernel itself (local implementation)
2. **Shared services**: provided by the Kernel and used by applications and the kernel itself.
3. **Shared services through Shared Interfaces**: provided by an application and used by another. It allows inter app communication.
   Shared interfaces are not demonstrated in this training (see sandboxed applications training)



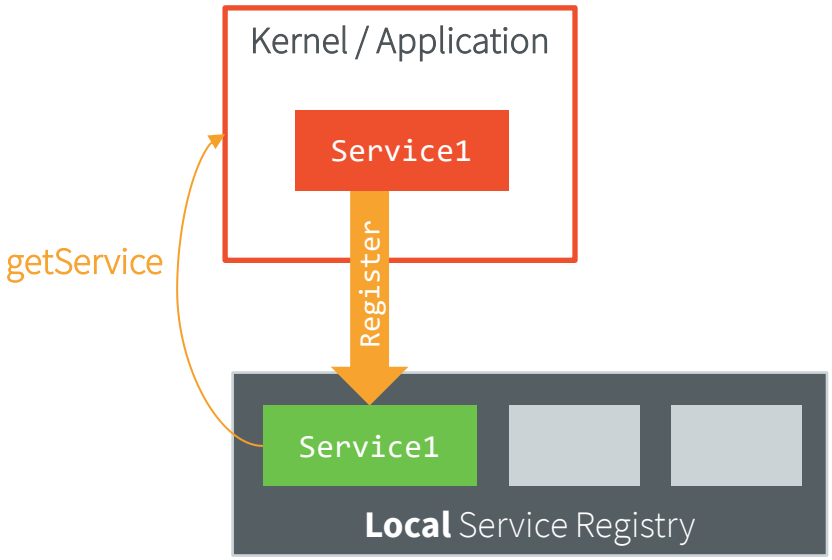
© MICROEJ 2026

LOCAL AND SHARED SERVICES

PROVIDE AND RETRIEVE A SERVICE

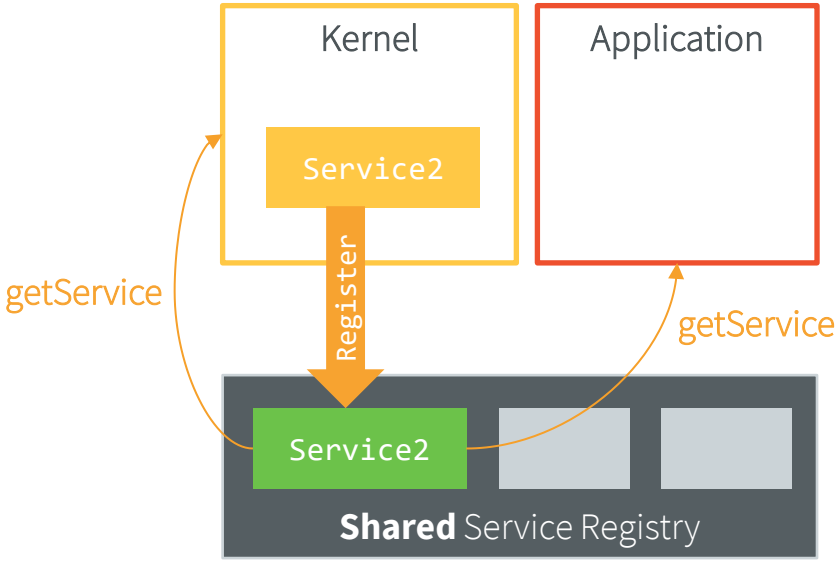
Local services can be provided by the Kernel and Applications within their own context.

- A Service instance provided the Kernel is only accessible by the Kernel.
- A Service instance provided by the Application is only accessible by the Application.



Shared Services are provided by the Kernel.

The Applications and the Kernel itself can use Shared Services.



© MICROEJ 2026

Hands-on

Provide a Kernel Timer (Shared Service) to Applications

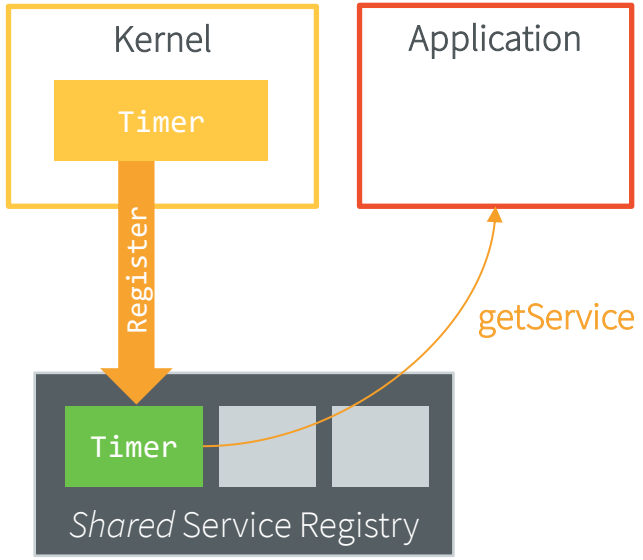
© MICROEJ 2026

HANDS ON SUMMARY

Hands on summary:

- Create a new Timer instance in the Kernel
- Update the Kernel to provide this Timer instance as a Shared Service
- Use this Timer service in the Application to print repeatedly a message in the console

Using a single Timer instance across the Kernel and Applications is beneficial to the system.
It avoids extra-thread creation when scheduling time-based; thus, reducing the required Runtime Heap.



© MICROEJ 2026

KERNEL: PROVIDE A TIMER SHARED SERVICE

In the **my-kernel** project:

- Add the following dependency lines to the **build.gradle.kts** from each Javadoc**:**

```
dependencies {
...
   implementation ("ej.api:bon:1.4.4")
   implementation("ej.library.runtime:service:1.2.0")
   implementation("com.microej.library.util:kf-util:3.3.0")
   implementation("com.microej.kernelapi:service:1.3.1")
```

Provides the Timer API (Javadoc)
Provides Service APIs (Javadoc)
Provides APIs to access the Shared Service Registry (Javadoc)
Provides Kernel API service

- Reload the **my-kernel** Gradle project.
- Add the following code in the **main()** method and import the necessary classes:

```
public static void main(String[] args) {
   String kernelName = Kernel.getInstance().getName();
   LOGGER.info(kernelName + " Hello World!");

   // Create a new ej.bon.Timer instance
   Timer myTimer = new Timer();
   // Get the Shared Registry instance
   ServiceRegistryKF serviceRegistryKF =(ServiceRegistryKF) ServiceFactory.getServiceRegistry();
   // Register the myTimer instance as a Shared Service
   serviceRegistryKF.register(Timer.class, myTimer, false);
```

⚠ Make sure to import the Timer class from BON and not java.util

~~import java.util.Timer;~~

import ej.bon.Timer;

© MICROEJ 2026

APPLICATION: RETRIEVE THE TIMER SERVICE

In the **my-application** project:

- Add the following dependency lines to the **build.gradle.kts:**

```
dependencies {
   ...
   implementation ("ej.api:bon:1.4.4")
   implementation("ej.library.runtime:service:1.2.0")
```

- Reload the **my-application** Gradle project.
- Add the following code in the **main()** method and import the necessary classes:

```
public static void main(String[] args) {
   LOGGER.info("Hello World!"); //$NON-NLS-1$

   // Get the Timer Service shared by the Kernel
   Timer timer = ServiceFactory.getService(Timer.class);
   // Check that the service has been properly retrieved
   if(timer != null){
      // Create a TimerTask to print a message periodically
      TimerTask timerTask = new TimerTask() {
         @Override
         public void run() {
            LOGGER.info("Hello from TimerTask!");
         }
      };
      // Schedule the TimerTask using the Timer provided by the Kernel
      timer.schedule(timerTask, 0, 500);
   }
```

⚠ Make sure to import the Timer and TimerTask classes from BON and not java.util

```
~~import java.util.Timer;~~
import ej.bon.Timer;
~~import java.util.TimerTask;~~
import ej.bon.TimerTask;
```

© MICROEJ 2026

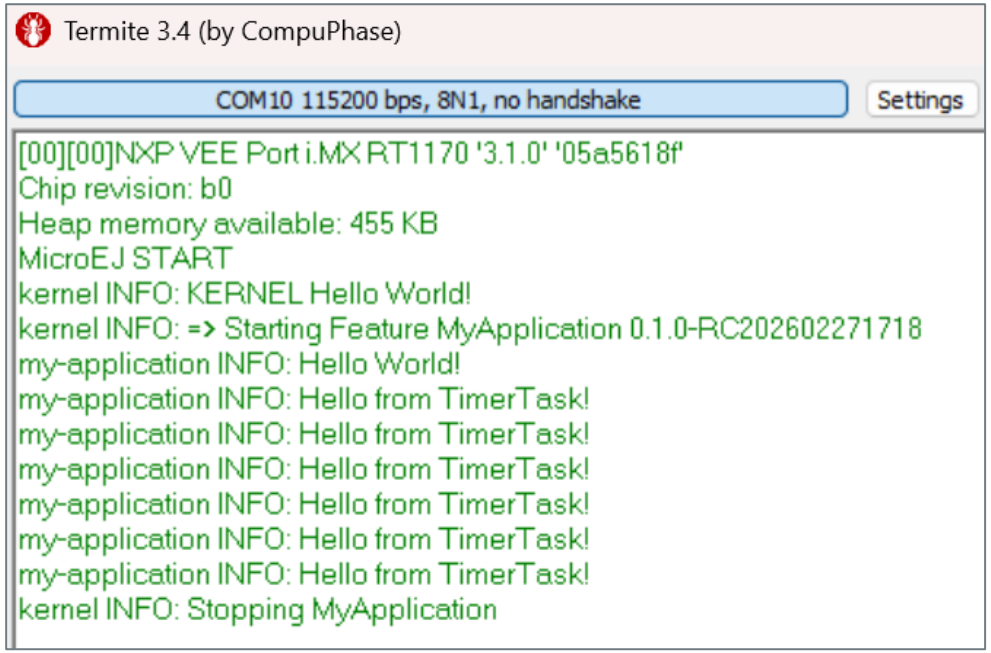
RUN ON THE DEVICE

- Run the **my-kernel > runOnDevice** task to program the updated Kernel on the device.
- Run **my-application > buildFeature** task.

- Copy **application.fo** at the root of a SD Card.
- Insert the SD Card in the NXP i.MXRT1170

Get the traces:

- Open the Termite serial terminal.
- Click the **Settings** button.
- Select the NXP i.MX RT1170 EVK board COM port.
- Reset the NXP i.MX RT1170 EVK board using Reset button
- **The “Hello from TimerTask!” message is printed periodically using the shared Timer service!**



© MICROEJ 2026

KEY TAKEAWAYS

- Services can be shared by means of the ej.Service library.

- Each Application owns a local registry in which it can register and get services within its own context; services registered in a local context cannot be retrieved by the Kernel nor any other Feature.
- The Kernel also has a local registry in which it can register services that can be used within its own context but not from the context of Features.
- Finally, there exists a unique shared service registry that contains all the registered shared services; this registry is available to all Features and to the Kernel as well.

- For more information about Kernel Services, check the Kernel and Features Communication documentation.

© MICROEJ 2026

Security Manager

Control API Invocation

© MICROEJ 2026

OVERVIEW

- The security manager defines the security policy for the System.
- This policy specifies actions that are sensitive or unsafe and requires additional control at runtime.
- Any action not allowed by the security policy throws a **SecurityException**.

- Usually, a Feature runs with a security manager provided by the Kernel.
  The security manager checks the Features's permission to execute a specific action as defined by the security policy.

- To obtain a reference to the Security Manager, invoke:

```
SecurityManager sm = System.getSecurityManager();
```

- To specify a Security Manager implementation, Invoke:

```
System.setSecurityManager(new MySecurityManager());
```

© MICROEJ 2026

Hands-on

Use the Security Manager to control the actions performed by Features on the File System.

© MICROEJ 2026

USE CASE OVERVIEW

- In this example, the Security Manager will be used to control the actions performed by Features on the File System.

- The Security Manager will restrict the ability of an application to perform File System actions outside of a predefined path.

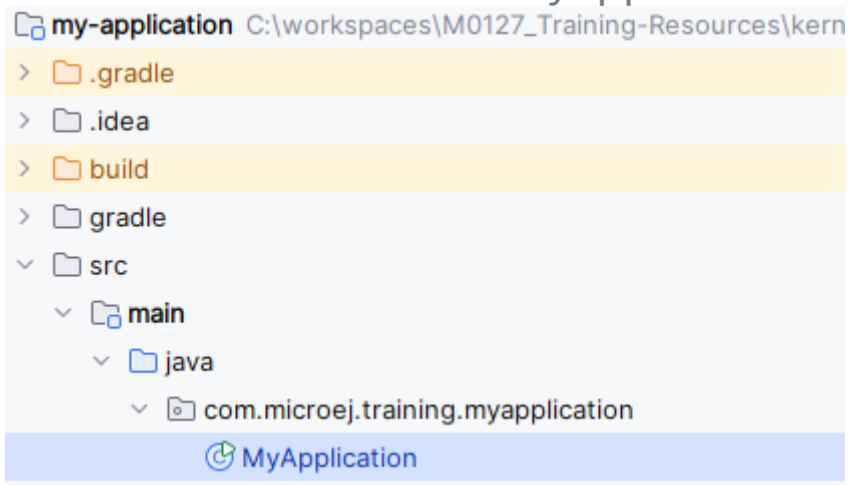
- The following rule will be implemented:
    - Features can only perform file actions under the file system path **/app/<APP_NAME>/**

© MICROEJ 2026

UPDATE MY-APPLICATION FEATURE NAME

For the purpose of this hands-on, update the feature name of the **my-application** project:

1. Rename the Main class to MyApplication:



2. Update the **applicationEntryPoint** in the **build.gradle.kts** file of **my-application**:

build.gradle.kts (my-application)

```
microej {
    applicationEntryPoint = "com.microej.training.myapplication.MyApplication"
}
```

© MICROEJ 2026

UPDATE MY-KERNEL PROJECT

The **my-kernel** project needs to be updated to expose the File System Kernel API to **my-application**:

- Add the following line to **my-kernel** project in the **build.gradle.kts** file:

```
dependencies {
    implementation("ej.api:edc:1.3.7")
    implementation("ej.api:kf:1.7.0")
    implementation("ej.api:fs:2.1.1")

    implementation("com.microej.kernelapi:edc:1.1.0")
    implementation("com.microej.kernelapi:bon:1.4.0")
    implementation("com.microej.kernelapi:logging:1.0.0")
    implementation("com.microej.kernelapi:fs:1.1.0")
```

- Reload the **my-kernel** Gradle project.
- Build the Kernel project (**my-kernel > buildExecutable**).

© MICROEJ 2026

CREATE A FILE ON THE FILESYSTEM

Update **my-application** to create a file in the FileSystem:

- Add the fs API to the **build.gradle.kts** file of **my-application**:

build.gradle.kts (my-application)

```
dependencies {
    implementation("ej.api:edc:1.3.7")
    implementation("ej.api:fs:2.1.1")
    implementation("ej.library.eclasspath:logging:1.2.1")
```

- Reload the **my-application** Gradle project.
- Update the **my-application main()** method as follows:

```
public static void main(String[] args) {
    LOGGER.info("Hello World!"); //$NON-NLS-1$

    final File file = new File("/app.txt");
    try (FileOutputStream fos = new FileOutputStream(file)) {
        fos.write("Hello from MyApplication!".getBytes());
    } catch (IOException e) {
        LOGGER.severe("ERROR: " + e.getMessage());
    }
    ...
```

© MICROEJ 2026

RUN ON THE DEVICE

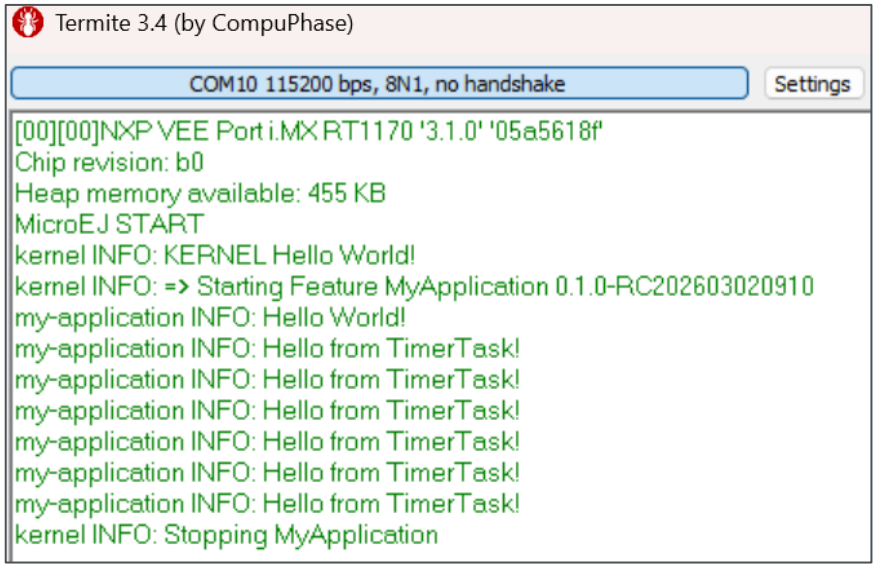
- Run the **my-kernel > runOnDevice** task to program the updated Kernel on the device.
- Run **my-application > buildFeature** task.

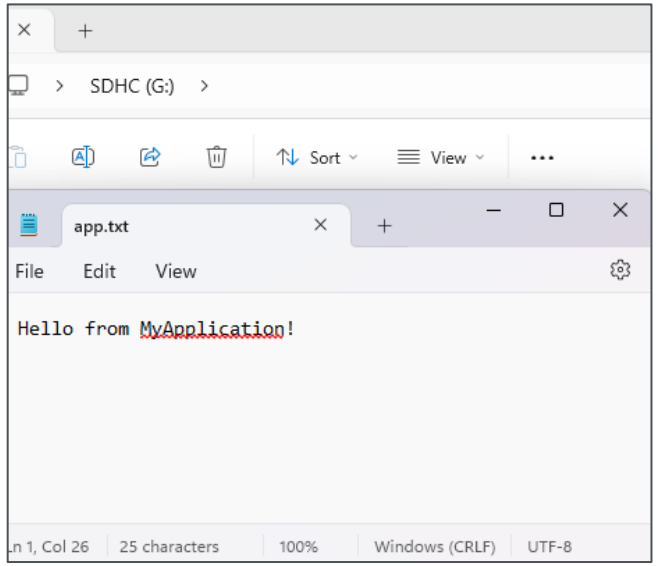
- Copy **application.fo** at the root of a SD Card.
- Insert the SD Card in the NXP i.MXRT1170

Get the traces:

- Open the Termite serial terminal.
- Click the **Settings** button.
- Select the NXP i.MX RT1170 EVK board COM port.
- Reset the NXP i.MX RT1170 EVK board using Reset button

- Remove micro-SD card from the board
- **Check the micro-SD card content, and notice that the file was created successfully**





© MICROEJ 2026

ADD A SECURITY MANAGER TO THE KERNEL

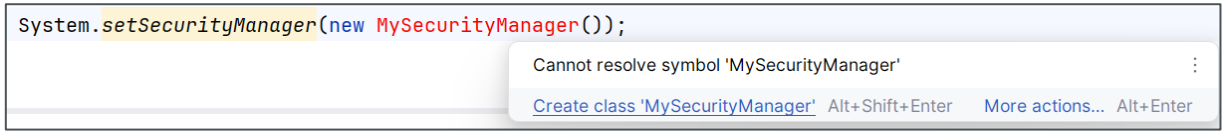
Add a Security Manager to the Kernel in order to control the actions performed by applications on files.

Update the **Main** class of the Kernel project:

- Add the line at the beginning of the **main()** method:

```
public static void main(String[] args) {
   String kernelName = Kernel.getInstance().getName();
   LOGGER.info(kernelName + " Hello World!");
   for (Feature feature : Kernel.getAllLoadedFeatures()) {
      LOGGER.info("=> Starting Feature " + feature.getName());
      feature.start();
   }

   System.setSecurityManager(new MySecurityManager());
   ...
```

- Hover the class “MySecurityManager” and click on “Create class ‘MySecurityManager’”



- Create this new class in the same folder as Main file
- Create the class **MySecurityManager** with the following content:

```
public class MySecurityManager extends SecurityManager {
   @Override
   public void checkPermission(Permission perm) {
      // implement security policy (see next slide)
   }
}
```

© MICROEJ 2026

CHECKPERMISSION IMPLEMENTATION (1/2)

```
public void checkPermission(Permission perm) {
   // When in Kernel mode, we allow all actions
   if (Kernel.isInKernelMode()) {
      return;
   }
   // Here we are in Feature Mode
   // Let's get the Feature Object
   // then, let's check that the Feature is allowed to perform the requested action.
   Feature feature = (Feature) Kernel.getContextOwner();
   // Check that the permission is a File permission
   // If so, let's ensure that the Feature comply with the following rule:
   // Applications can only perform file actions under the file system path /app/<APP_NAME>/
   if (perm instanceof FilePermission) {
      // Get the feature name
      String featureName = feature.getName();

      // Safely cast perm to FilePermission to have access to the File permission data
      FilePermission filePerm = (FilePermission) perm;
      String path = filePerm.getName(); // the name in a FilePermission contains the file path

. . .
```

© MICROEJ 2026

CHECKPERMISSION IMPLEMENTATION (2/2)

```
. . .
// Perform all the following FS operations under the Kernel Context
Kernel.enter();
// Create the Feature Root Directory if not exists
File featureRootPath = new File("/app/" + featureName + "/");
if (!featureRootPath.exists()) {
   if (!featureRootPath.mkdirs()) {
      throw new SecurityException("Error While creating Feature root path: " + featureRootPath);
   }
}
// Check that the path is allowed
// We use canonical path to avoid path traversal issues that can be caused by “..” in path
try {
   File file = new File(path);
   if (!file.getCanonicalPath().startsWith(featureRootPath.getCanonicalPath())) {
      throw new SecurityException("This feature is only allowed to read/write files under /app/" + featureName + "/");
   }
} catch (IOException e) {
   throw new SecurityException("Can't get canonical path");
}
// exit Kernel Context after all FS operations
Kernel.exit();
} // end of if (perm instanceof FilePermission)
} // end of checkPermission()
```

Features will not be able to create their Feature root directory by themself.

Features are only allowed to operate on files within their Feature path.

© MICROEJ 2026

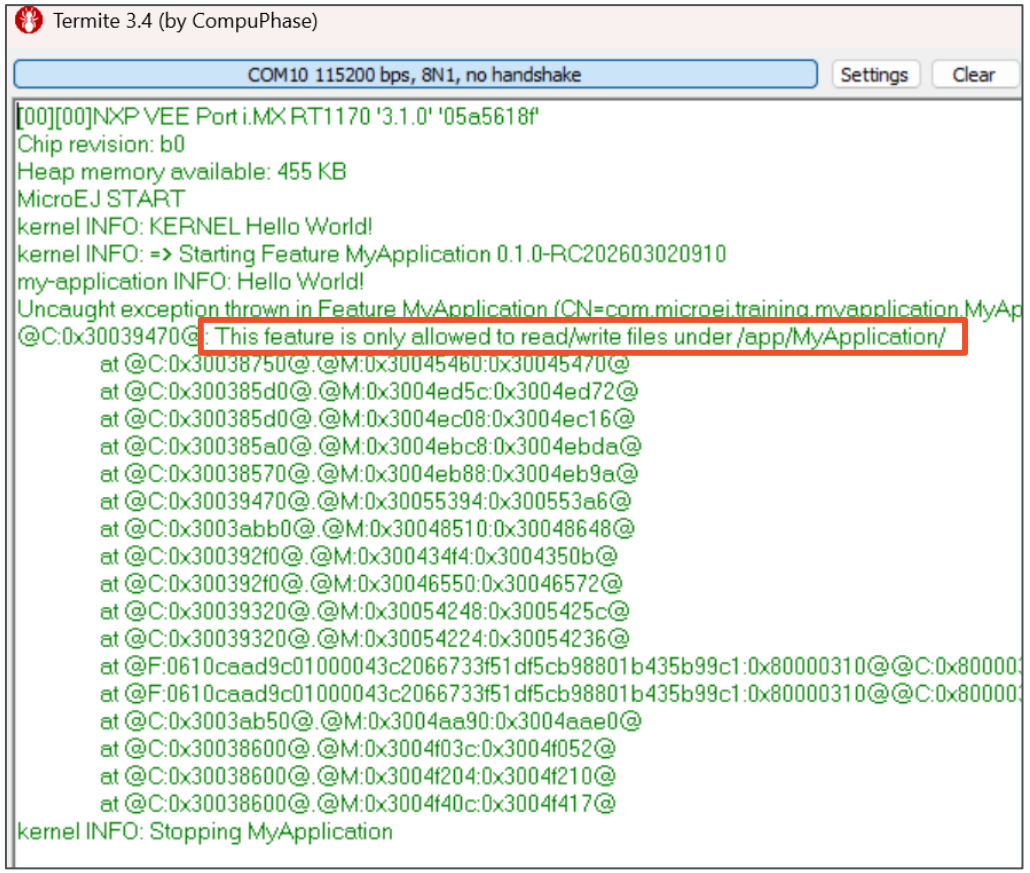
RUN ON THE DEVICE

- Run **my-kernel > runOnDevice** task.
- Run **my-application > buildFeature** task.

- Copy **application.fo** at the root of a SD Card.
- Insert the SD Card in the NXP i.MXRT1170

Get the traces:

- Open the Termite serial terminal.
- Click the **Settings** button.
- Select the NXP i.MX RT1170 EVK board COM port.
- Reset the NXP i.MX RT1170 EVK board using Reset button
- **A SecurityException is thrown. The Application is not allowed to manipulate files outside of the authorized path.**



© MICROEJ 2026

UPDATE THE APPLICATION CODE

Update the **my-application main()** method to comply with the Security policy ; write the file under the **MyApplication** folder:

```
public static void main(String[] args) {

   LOGGER.info("Hello World!"); //$NON-NLS-1$

   final File file = new File("/app/MyApplication/app.txt");
   try (FileOutputStream fos = new FileOutputStream(file)) {
      fos.write("Hello from MyApplication!".getBytes());
   } catch (IOException e) {
      LOGGER.severe("ERROR: " + e.getMessage());
   }
   …
```

© MICROEJ 2026

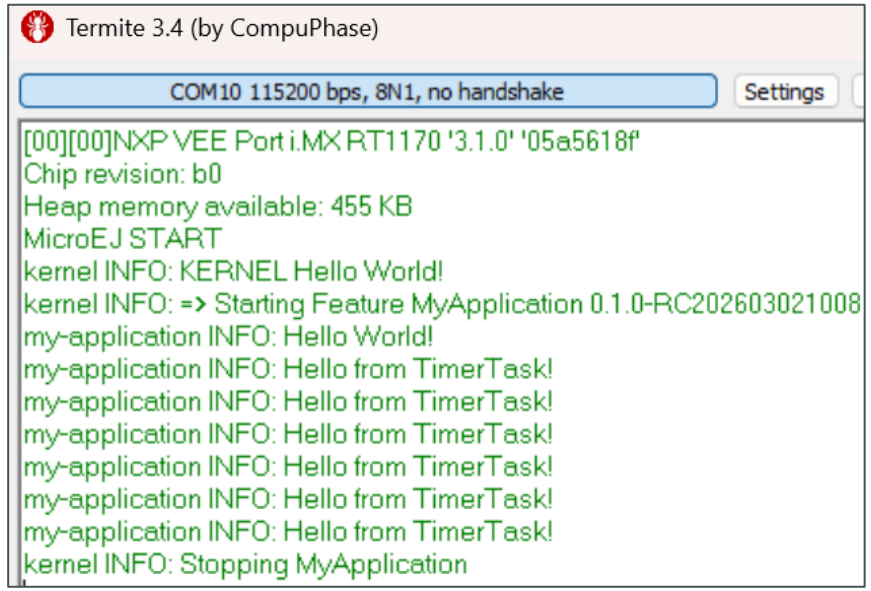
RUN ON THE DEVICE

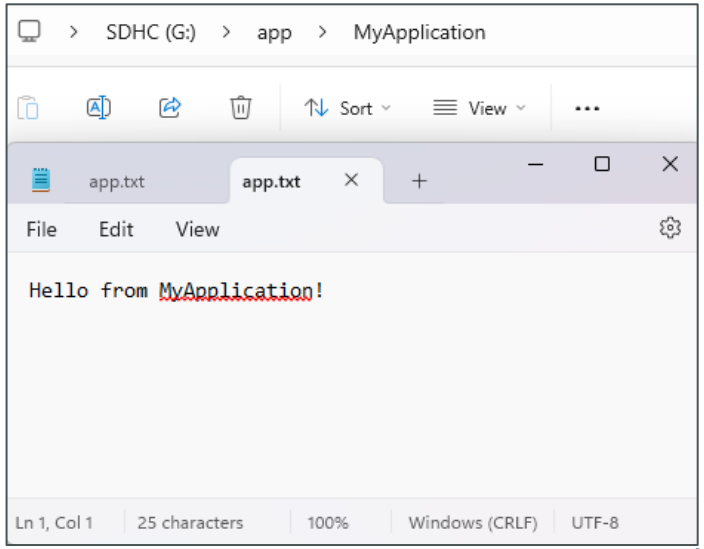
- Run the **my-kernel > runOnDevice** task to program the updated Kernel on the device.
- Run **my-application > buildFeature** task.

- Copy **application.fo** at the root of a SD Card.
- Insert the SD Card in the NXP i.MXRT1170

Get the traces:

- Open the Termite serial terminal.
- Click the **Settings** button.
- Select the NXP i.MX RT1170 EVK board COM port.
- Reset the NXP i.MX RT1170 EVK board using Reset button
- **Check the micro-SD card content. The file is now created under the /app/MyApplication folder.**





© MICROEJ 2026

KEY TAKEWAYS

- A security policy allows the Kernel to prevent an Application from accessing resources or calling specific APIs. Whereas Kernel APIs allow to control at build-time, the security policy here controls the APIs called at runtime.
- Security manager can be used to implement a security policy.
- Security manager can control all the sensitive and unsafe code sections.
- Security manager can control which Feature is allowed to call which method or perform a specific action in the system.
- Custom Permissions can be created for custom use cases.

The Security Manager also provides a ready-to-use implementation based on permissions declared by the Application itself. It assumes the application and its permissions file have been approved beforehand by a moderator.

Check the Define a Security Policy documentation for more information.

© MICROEJ 2026

THANK YOU

for your attention !

MICROEJ®