Service Integration: How to Build an Application
Build the Application

Configure Datatypes
- Determine needed types, describe them
- Configure basic service attributes
- Configure logging

Configure Services
- Configure basic service attributes
- Define the service mesh
- Create the templates

Configure Application
- Define the application
- Define the service attributes
- Configure logging

Build the Application
- Implement the service functionality
- Install the services
- Create the templates
- Build the application

Test the Services
- Create (unit) tests
- Use generated tests

Deploy the Application
- Add application
- Start up a platform
- Run the application
Table of Contents

- Prerequisites
- Building an Application
Prerequisites

• Required:
  • Installed the platform and its dependencies or the development container
  • Installed the IDE for IIP-Ecosphere Platform (provided Eclipse Version)
  • How to configure datatypes
  • How to configure services
  • How to configure an application

• Optional:
  • Introduction to code generation
Table of Contents

• Prerequisites
• Building an Application
Building an Application (1)

- The templates for the .ivml files should enable you to now build a application right away
  - If you followed the videos chronologically this is already done

  ![Diagram showing the directory structure and commands]

- Rename the “impl.model” as needed/desired
- Use cmd to run “mvn -U generate-sources -Dunpack.force=true” in the “impl.model” directory
To view the generated classes go to `impl.model/gen/py/ApplicationInterfaces/

- You should NOT edit these classes, if they do not behave as you expect double check if the corresponding `.ivml` file is correct.
- IF you have a python service you will find `ApplicationInterfaces/python` as well as `ApplicationInterfaces/java`
Building an Application (3)

- Templates is the location where you can find your main service classes
  - Here you can quickly confirm that the generation looks like you intended
  - The “gen/...” location will be overwritten if you re-generate the templates to change i.e. datatypes
  - To create functionality import the .zip file
Import the Implementation Template (1)

- Import the template into Eclipse to work on them
  - Create a new, empty Maven project (preferably in the same workspace)
  - Select “Create a simple project”
  - Fill in the fields (only “Artifact Id” will remain and be the project name in Eclipse)
  - Press finish to create the project
• Import the templates to work on them
• Choose import → General → Archive file
  • Select the template .zip located in “/gen/py/templates/eclipse/”
• Select the new Maven project
• Tick to overwrite existing files
• Run Maven Update on the project
  • Right click → Maven → Update Project…
• Go to your imported project
  • The path is dependent on your naming choices!
  • Edit the `produce<YourInputDataName>()` method to return a new instance of your datatype

```java
@Override
public InData produceInData() {
    InData result = new InDataImpl();
    result.setIntExample(1);
    result.setFloatExample(1);
    result.setStringExample("1");
    result.setDoubleExample(1);
    // TODO add your code here
    return result;
}
```

• Input data is named `InData`
• In this example we just use 1 as a value for each datapoint
Building an Application (5)

- Go to your imported project
  - Edit the method receiving your datatype in a way that it passes on an instance of the output datatype

```java
@Override
public void processInData(InData data) {
    OutData out = new OutDataImpl();
    out.setResult(2);
    out.setStringExample("2");
    ingestOutData(out);
}
```

- Create an instance of your output datatype
- Call `ingest<outputName>(<instance>)`
Building an Application (6)

- Go to your imported project
  - Edit the method receiving your output datatype to print the values to the console

```java
@override
public void processOutData(OutData data) {
    System.out.println(data.getStringExample());
    System.out.println(data.getResult());
}
```

- This is the end of the application, utilise your results as needed
• The complete source class as created in the videos.

```java
import java.io.*;

/**
 * Service implementation for net node 'Source'.
 * Generated by: EASY-Producer.
 */

public class TestSource extends SourceImpl {
    public TestSource() {
        super(ServiceKind.SOURCE_SERVICE);
    }

    public TestSource(String serviceId, InputStream ymlFile) {
        super(serviceId, ymlFile);
    }

    @Override
    public InData produceInData() {
        InData result = new InDataImpl();
        result.setIntExample(1);
        result.setFloatExample(1);
        result.setDoubleExample(1);
        result.setStringExample("1");
        return result;
    }
}
```
The complete transformer class as created in the videos.

Example Transformer File

```java
package de.iip_ecosphere.platform.impl.shop;

import java.io.*;

/**
 * Service implementation for net node 'PyService'.
 * Generated by: EASY-Producer.
 */

public class TestTransformer extends PyServiceImpl {

    /**
     * Fallback constructor, also used for testing main program.
     */
    public TestTransformer() {
        super(ServiceKind.TRANSFORMATION_SERVICE);
    }

    /**
     * Creates a service instance from a service id and a YAML artifact.
     *
     * @param serviceId the service id
     * @param ymlFile the YAML file containing the YAML artifact with the service descriptor
     */
    public TestTransformer(String serviceId, InputStream ymlFile) {
        super(serviceId, ymlFile);
    }

    @Override
    public void processInData(InData data) {
        OutData out = new OutDataImpl();
        out.setStringExample("Out");
        out.setResult(2);
        ingestOutData(out);
    }
```
• The complete sink class as created in the videos.

```java
package de.iip_ecosphere.platform.impl.shop;

import java.io.*;

/**
 * Service implementation for net node 'Sink'.
 * Generated by: EASY-Producer.
 */
public class TestSink extends SinkImpl {

    /**
     * Fallback constructor, also used for testing main program.
     */
    public TestSink() {
        super(ServiceKind.SINK_SERVICE);
    }

    /**
     * Creates a service instance from a service id and a YAML artifact.
     *
     * @param serviceId the service id
     * @param ymlFile the YML file containing the YAML artifact with the service descriptor
     */
    public TestSink(String serviceId, InputStream ymlFile) {
        super(serviceId, ymlFile);
    }

    @Override
    public void processData(OutData data) {
        System.out.println(data.getStringExample());
        System.out.println(data.getResult());
    }
}
```
• In your imported project run “mvn install” to make the services available to your main application through maven
• After successfully installing the templates run “mvn install” in `impl.model` again to finish creating the application.
• Your finished App will be in `impl.model/gen/py/<Appname>/target/<Appname><Version>-SNAPSHOT-bin.jar`
Summary

• What we learned
  • How to add functionality to the services
  • How to build an application from the template project
• How to go on
  • How to test an application
  • Running the application in the platform