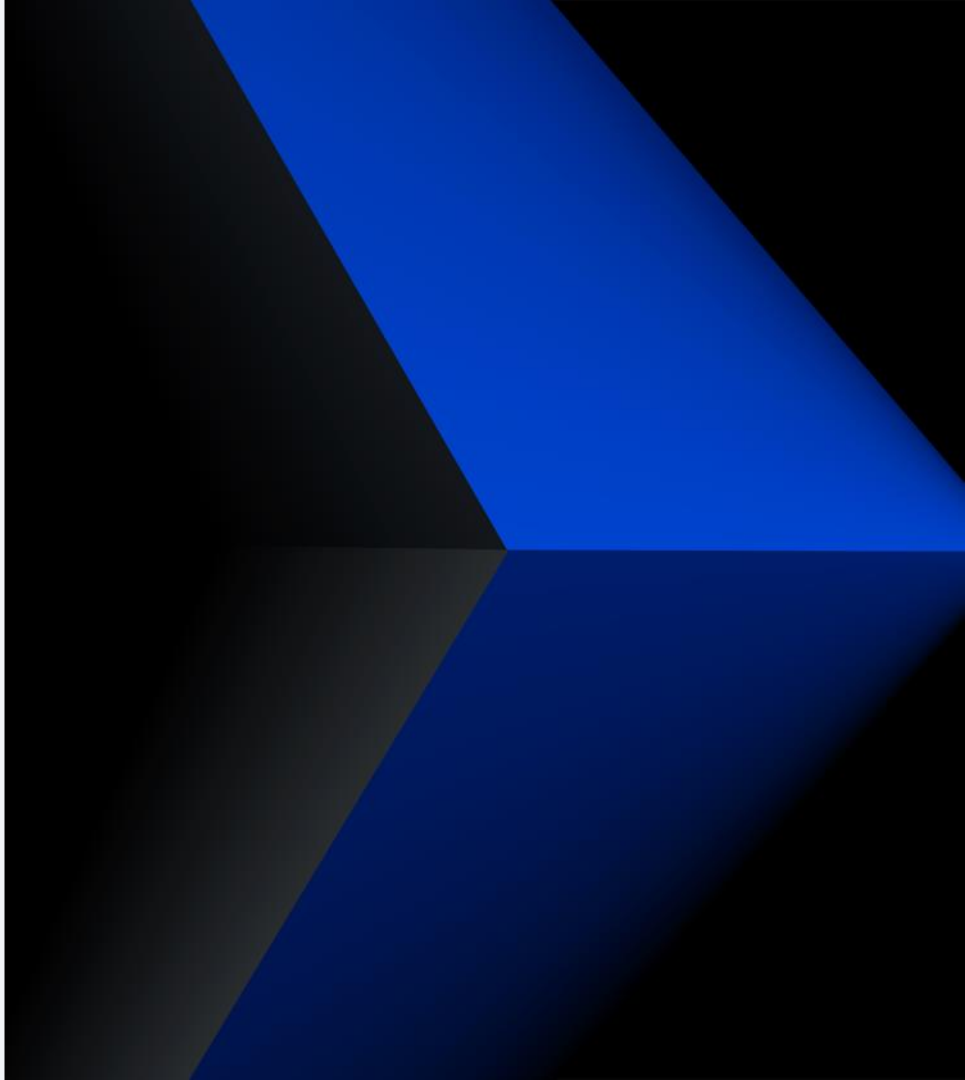
Integrating Java Programming on z/TPF

2022 TPF Users Group Conference
March 27-30, Dallas, TX
Education Session

Daniel Gritter

IBM Z

Disclaimer



Java on z/TPF – leveraging the value of Java

The IBM lab has chosen Java as a key feature to extend functionality of z/TPF. In this education, we’ll demonstrate 3 ways that you can use Java to modernize your applications and facilitate integration with your enterprise solutions.

Java Use Cases on z/TPF

There's three many categories where we believe Java can provide a tremendous amount of value:

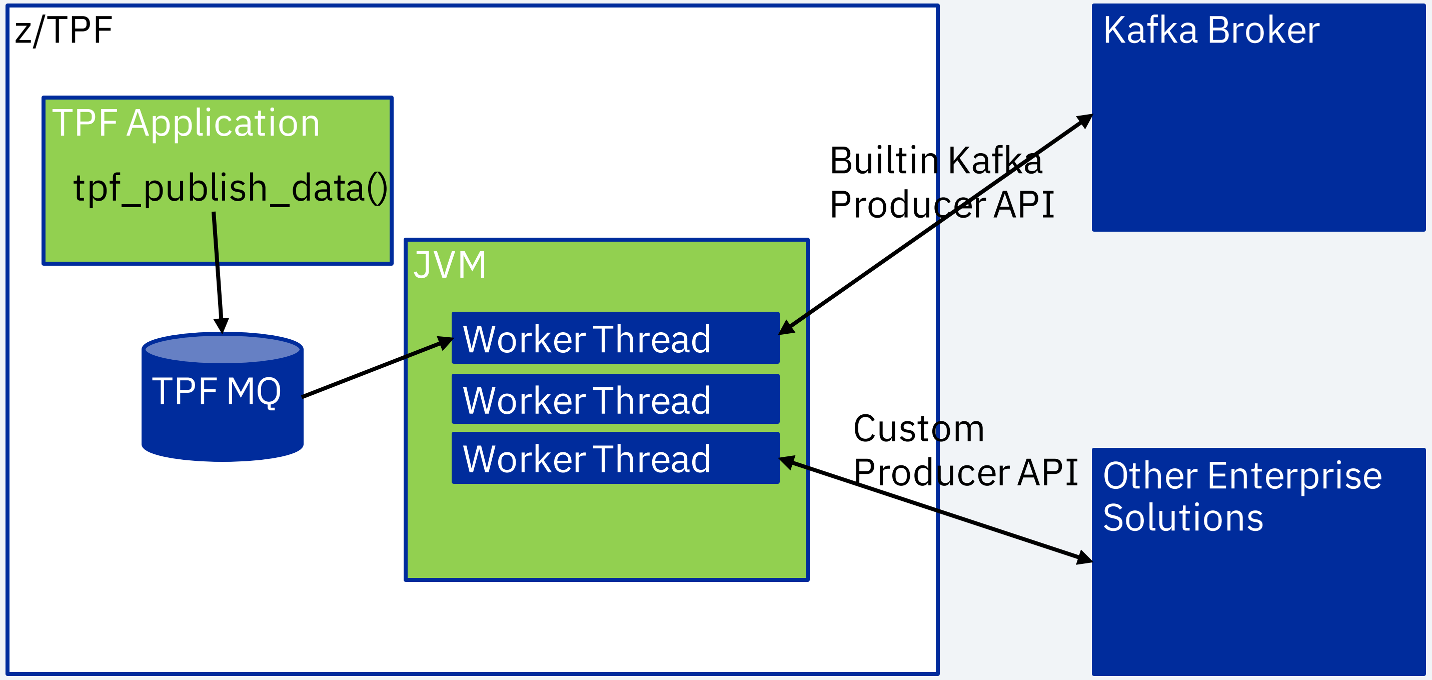
-Extending Guaranteed Delivery support

-Extending z/TPF applications through service calls

-Implementing Standard interfaces

Extending Guaranteed Delivery Support

Extending Guaranteed Delivery



Extending Guaranteed Delivery

-Either use Guaranteed Delivery right out of the box for use with Kafka or customize for additional options

-Minimal time to deployment

-Standard interface with minimal coding to accommodate different library programming models

Best Practices

Start with the “hello world” model. Build the smallest possible unit of functionality first then build around that.

Refactor during/after integration

Function first, then performance test to focus improvements

Extending Guaranteed Delivery support

By default, Guaranteed Delivery supports Apache Kafka. However, it is designed to be extensible to support multiple interfaces for use with other data publication targets.

There are two providers support by z/TPF demonstrating the two main different models:

- Synchronous model: Data is published before a thread reads another message from the input queue (DF or MQ)
- Asynchronous model: Data is published some period of time after the message is read from the queue, such that multiple messages are in flight at the same time.

Use case: Extending Guaranteed Delivery

Company X needs to publish data using Google Pub/Sub.

They have at least three options available on z/TPF:

1) REST API access
2) Use Guaranteed Delivery to write to Kafka, then redirect to Pub/Sub server using Kafka Connect
3) Implement custom Guaranteed Delivery provider.

Google Cloud Pub/Sub

Use case: Extending Guaranteed Delivery tradeoffs

1) REST API access

PRO) Simplest and most direct

CON) Potentially inefficient at scale as each message requires a separate request

CON) Potential path length / existence time impact to application or application rewrite to use asynchronous HTTP

2) Use Guaranteed Delivery to write to Kafka, then redirect to Pub/Sub server using Kafka Connect

PRO) Can use as a starter method to familiarize with Guaranteed Delivery

CON) Extra infrastructure overhead / latency / complexity

3) Implement custom Guaranteed Delivery provider.

PRO) Best throughput potential with minimal code

CON) Java programming / integration concepts may not exist yet in your TPF ecosystem

Use case: Extending Guaranteed Delivery tradeoffs vs native implementation

Using the infrastructure provided with TPF Guaranteed Delivery allows you to decouple your application processing for data publish, either as part of a custom Business Events dispatcher or directly from your application.

There’s a variety of advantages when you compare to the cost of porting standard packages:

- TPF Applications are typically designed for a short lifecycle. Many native library implementations use a complex threading model which may not be a good fit for TPF applications
- Complexity of porting effort. Many native library implementations have built-in assumptions about codepage and platform that are not consistent with TPF.
- Time to market. With an easy API and few moving parts it’s straightforward to build a solution in a matter of weeks.

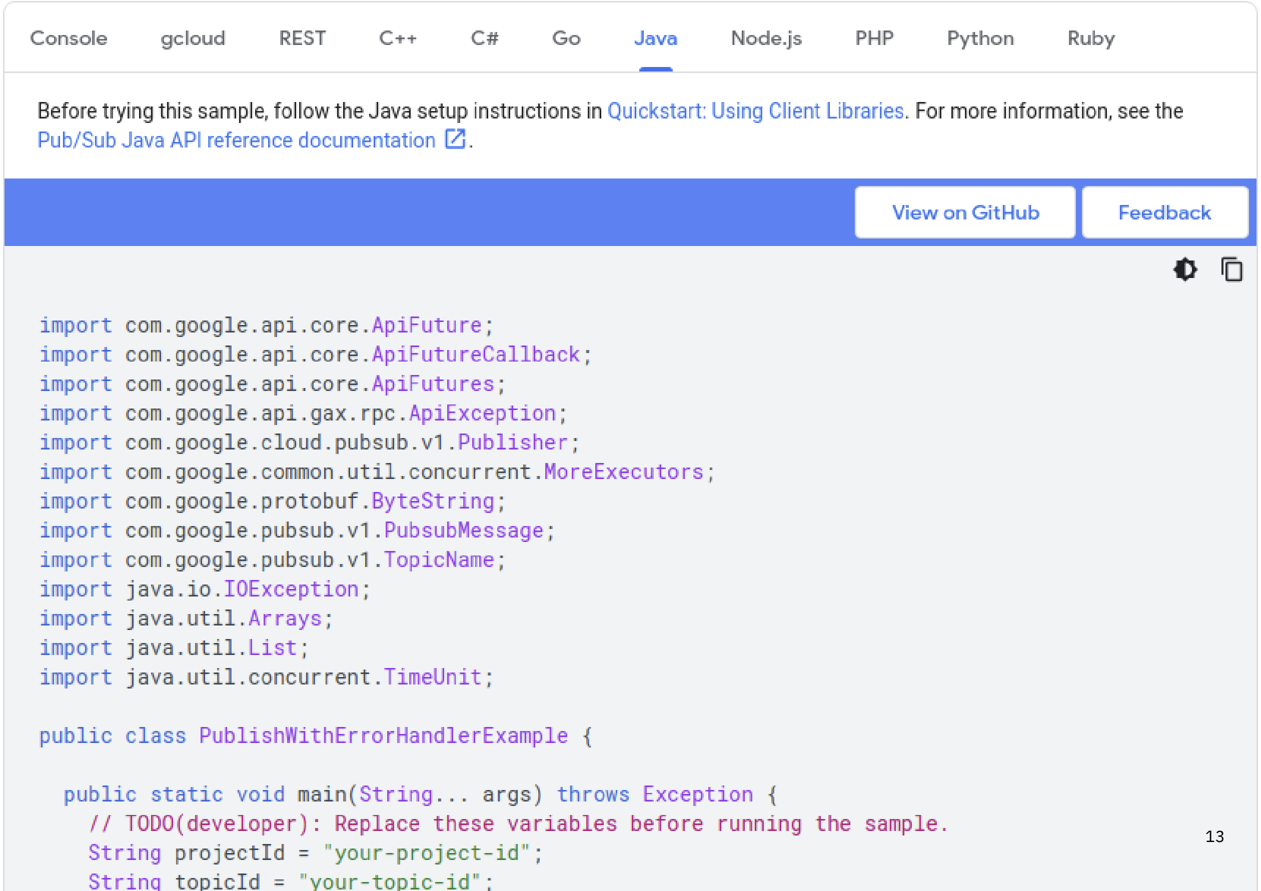
Use case: Google Pub/Sub

For the first step in Java programming, I like to build in terms of smallest functional unit.

Distill the problem into the smallest piece that works ("hello world") then extend.

For this use case we're going to investigate the Google Pub/Sub library:

https://cloud.google.com/pubsub/docs/publisher



Before trying this sample, follow the Java setup instructions in Quickstart: Using Client Libraries. For more information, see the Pub/Sub Java API reference documentation.

```java
import com.google.api.core.ApiFuture;
import com.google.api.core.ApiFutureCallback;
import com.google.api.core.ApiFutures;
import com.google.api.gax.rpc.ApiException;
import com.google.cloud.pubsub.v1.Publisher;
import com.google.common.util.concurrent.MoreExecutors;
import com.google.protobuf.ByteString;
import com.google.pubsub.v1.PubsubMessage;
import com.google.pubsub.v1.TopicName;
import java.io.IOException;
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.TimeUnit;

public class PublishWithErrorHandlerExample {

  public static void main(String... args) throws Exception {
    // TODO(developer): Replace these variables before running the sample.
    String projectId = "your-project-id";
    String topicId = "your-topic-id";
```

Use case: Google Pub/Sub Investigation

As the investigation into the online document unfolds, you will find there’s a wealth of online resources with code snippets and guidance and samples, in this case included directly in the documentation:

For pub/sub, there’s a “Publisher” class that you can instantiate an instance of:

```
Publisher publisher = null;

// Create a publisher instance with default settings bound to the topic

publisher = Publisher.newBuilder(topicName).build();
```

Use case: Google Pub/Sub Investigation

There's some other classes that you use to build a message, and then publish it:

```
ByteString data = ByteString.copyFromUtf8(message);

PubsubMessage pubsubMessage =
 PubsubMessage.newBuilder().setData(data).build();


// Once published, returns a server-assigned message id (unique within the topic)

ApiFuture<String> future = publisher.publish(pubsubMessage);
```

Use case: Google Pub/Sub Investigation

The ApiFuture class that's returned can be used to perform actions on result of publication:

```
ApiFuture<String> future = publisher.publish(pubsubMessage);

ApiFutures.addCallback(future,

    new ApiFutureCallback<String>() {

                    public void onFailure(Throwable throwable) {}

                    public void onSuccess(String messageId) {}

    },

    MoreExecutors.directExecutor());
```

Use case: Google Pub/Sub Investigation

Finally, there are methods used to cleanup resources upon completion of publishing messages:

```
publisher.shutdown();

publisher.awaitTermination(1, TimeUnit.MINUTES);
```

This is a fairly simple model but is the best place to start, copying the sample or tweaking it to make it your own. Best part of this is you can implement it directly in a workstation in a Java IDE to get it working quickly and to interactively understand the flow.

Use case: Google Pub/Sub Investigation

In order to implement a custom delivery class, it must extend the **com.ibm.tpf.PublishInterface** class. This class contains one method:

```
    java.util.concurrent.Future publish(request request);
```

The return value is a "Future" object, which is only used with DF queue type to identify when the operation has completed if processing occurs in synchronous mode. The Future.get() method is called for synchronization.

Use case: Google Pub/Sub Investigation

The majority of the work is expected to be resolved through use of the com.ibm.tpf.PublishRequest class. This class contains several methods expected to be used by a data publish provider:

```
public String getTopicName();
public String getKey();
public byte[] getData();

public void async();
public void commit();
public void rollback();
public void retry(long errcode, String message);
public void error(long errcode, String message);
```

Use case: Google Pub/Sub Investigation

The three getters provide access to the data passed on the tpf_publishData() api from a TPF application or business events dispatch adapter:

```
public String getTopicName();
public String getKey();
public byte[] getData();
```

Use case: Google Pub/Sub Investigation

The other methods are used to manage the state of the request.

public void async();

The async() method is used to indicate to the framework that this is an asynchronous request and additional messages may be read from the input queue while processing the request.

public void commit();

This method should be called when confirmation is received that processing has been completed successfully.

public void rollback();

This method should be called when the message should be reverted to it's original condition. This is different than a "retry" which indicates an attempt was made to deliver and failed.

Use case: Google Pub/Sub Investigation

The other methods are used to manage the state of the request.

```
public void retry(long errcode, String message);
```

The retry() method should be called when a “recoverable” error was encountered, and the operation may be successful at some future time with the current configuration.

```
public void error(long errcode, String message);
```

The error() method should be called when a “unrecoverable” error was encountered, and the operation would never be successful given the current configuration.

Use case: Google Pub/Sub Implementation

Based on the investigation done, we need to merge the two pieces together into a new producer class. *(this is simplified, does not include required try / catch logic)

```
class GooglePubSubProvider implements PublishInterface {
		java.util.concurrent.Future publish(request request) {
				// mark this as an asynchronous request
				request.async();
				String topic = request.getTopic();
				// locate/create the publisher based on topic name
				Publisher publisher = publishers.get(topic);
				ApiFuture<String> future = publisher.publish(pubsubMessage);

		… <next page>
				return future;
	}

}
```

Use case: Google Pub/Sub Implementation

The callback is setup to handle success or failure after completion rather than blocking

```
ApiFutures.addCallback(future, new ApiFutureCallback<String>() {
public void onFailure(Throwable throwable) {
  if (throwable instanceof ApiException) {
    ApiException apiException = ((ApiException) throwable);
    // details on the API exception
    if (apiException.isRetryable()) {
     request.retry(apiException.getStatusCode().getCode()),apiException.getMessage()
    } else {
     request.error(apiException.getStatusCode().getCode()),apiException.getMessage()
    }
  } else {
    request.error(GENERIC_THROWABLE_CODE,throwable.getMessage();
  }
}
public void onSuccess(String messageid) {
  request.commit();
}
},
MoreExecutors.directExecutor());
```

Use case: Google Pub/Sub Implementation

Once the class is packaged, we update the configuration to point to the new class in the jam.xml configuration:

```
<tns:TopicGroupList>
    <tns:TopicGroup>
      <tns:TargetName>  stats  </tns:TargetName>

      <tns:QueueName> GPS.INPUT.QUEUE </tns:QueueName>
      <tns:ErrorQueue> GPS.ERROR.QUEUE </tns:ErrorQueue>
      <tns:RetryQueue> GPS.RETRY.QUEUE </tns:RetryQueue>

      <tns:TopicHandler> com.company.GooglePubSubProvider</tns:TopicHandler>

      <tns:RetryInterval> 100 </tns:RetryInterval>
      <tns:RetryCount> 5 </tns:RetryCount>
      <tns:NumberWorkerThreads> 5 </tns:NumberWorkerThreads>
      <tns:NumberOfInFlightRequests> 10 </tns:NumberOfInFlightRequests>

    </tns:TopicGroup>
  </tns:TopicGroupList>
```

Use case: Google Pub/Sub Test/Deployment

Configuration and tuning of the deployment may vary based on a number of factors.

Each library will provide its own set of parameters and configuration to tune for a desired outcome.

In general, multiple JVMs for scalability should not be necessary until you reach a throughput limit for a single JVM, either based on the maximum number of threads or simply bottlenecks due to locking. However, it is recommended that you use a minimum of 2 JVMs for redundancy.

Extending z/TPF applications and utilities

Extending z/TPF applications and utilities

Support is based on REST, using OpenAPI descriptors

Define desired interface first, then generate code for implementation of producer / consumer ends

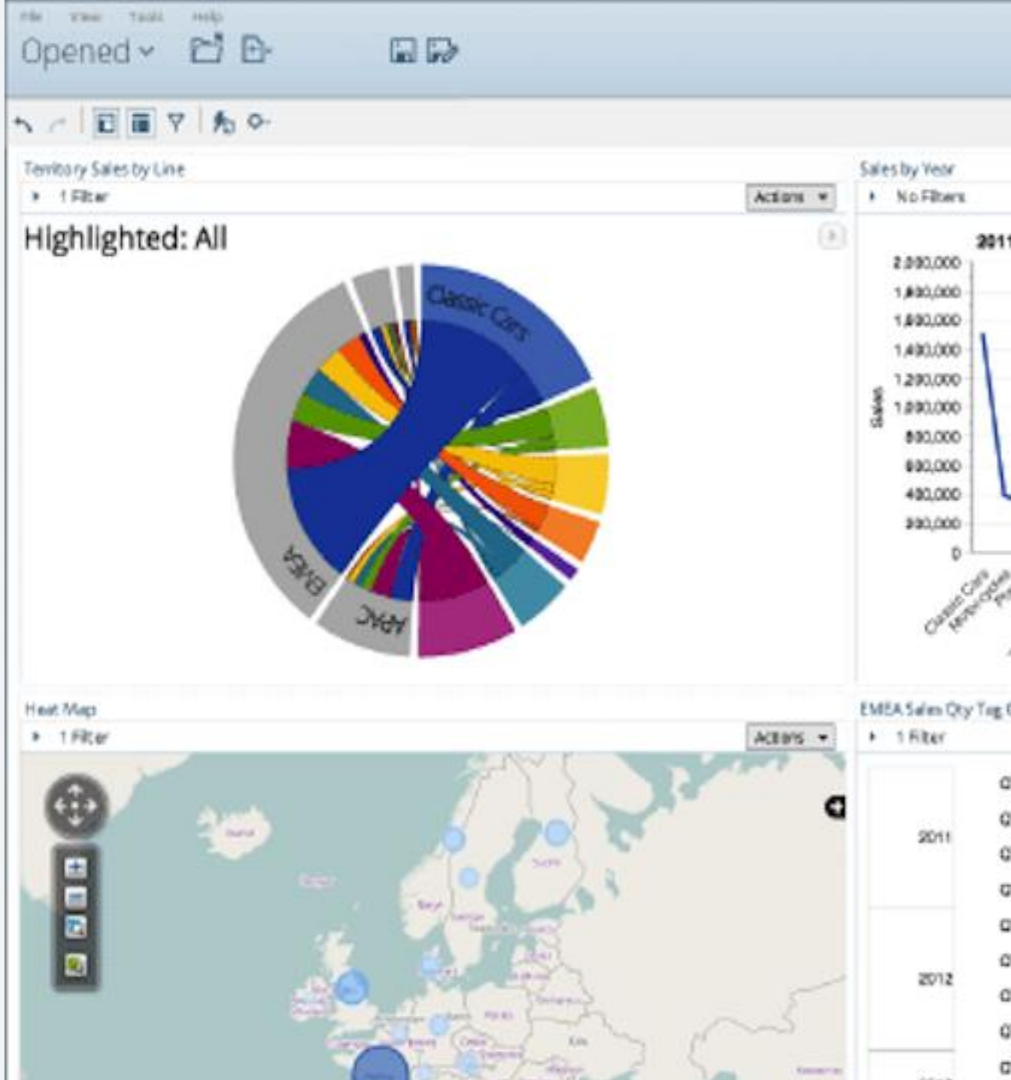
Error handling for scenarios when the JAM is not available

Use case: Calling analytics

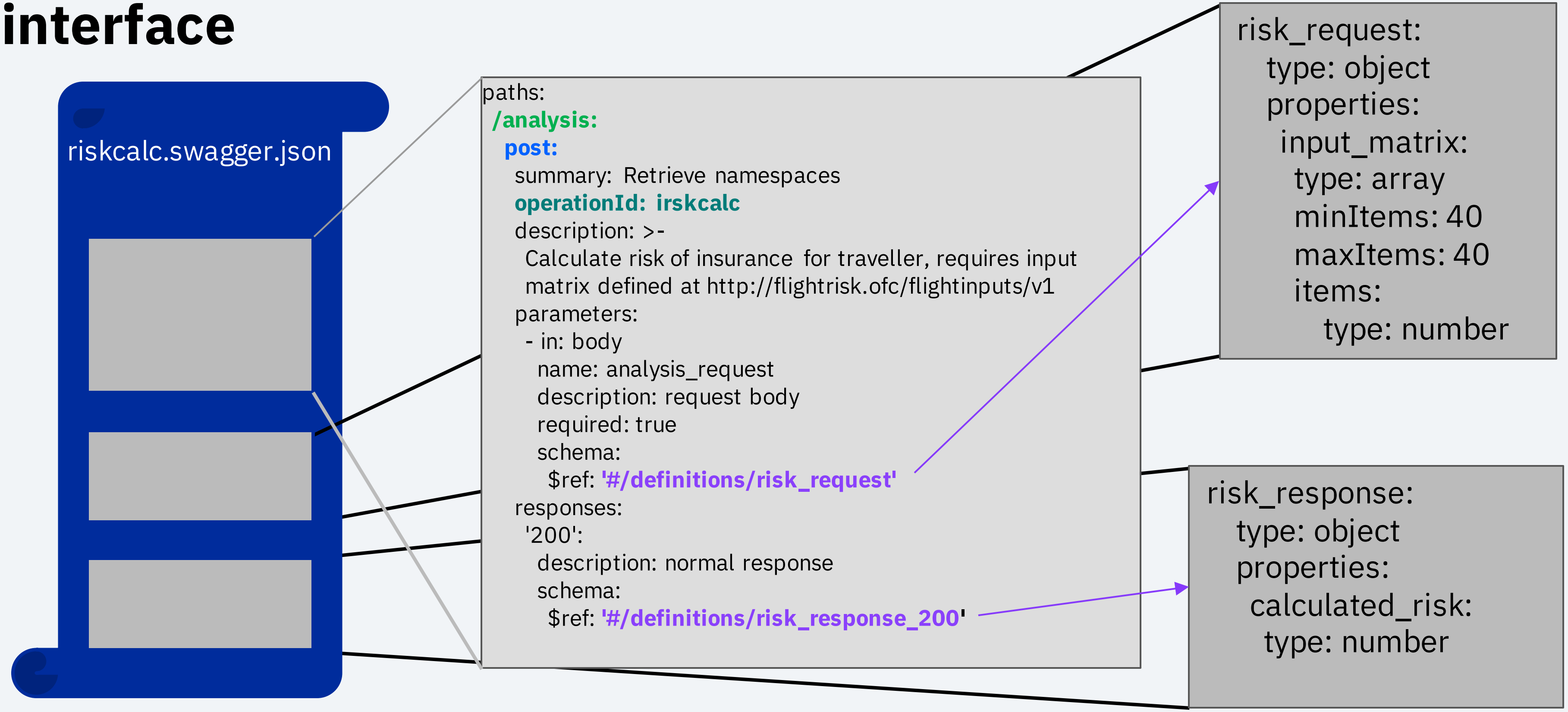
Company R needs to extend their application to include AI model processing into their transaction for fraud detection

The application currently has embedded rule processing hard-coded in C and wants to replace this with a more dynamic approach. They decided to implement this using a commercial product only available in the Java programming language.

To invoke the processing, they need to pass 40 different data points into the application, then return a risk analysis from the application.



Analytics: Defining the interface



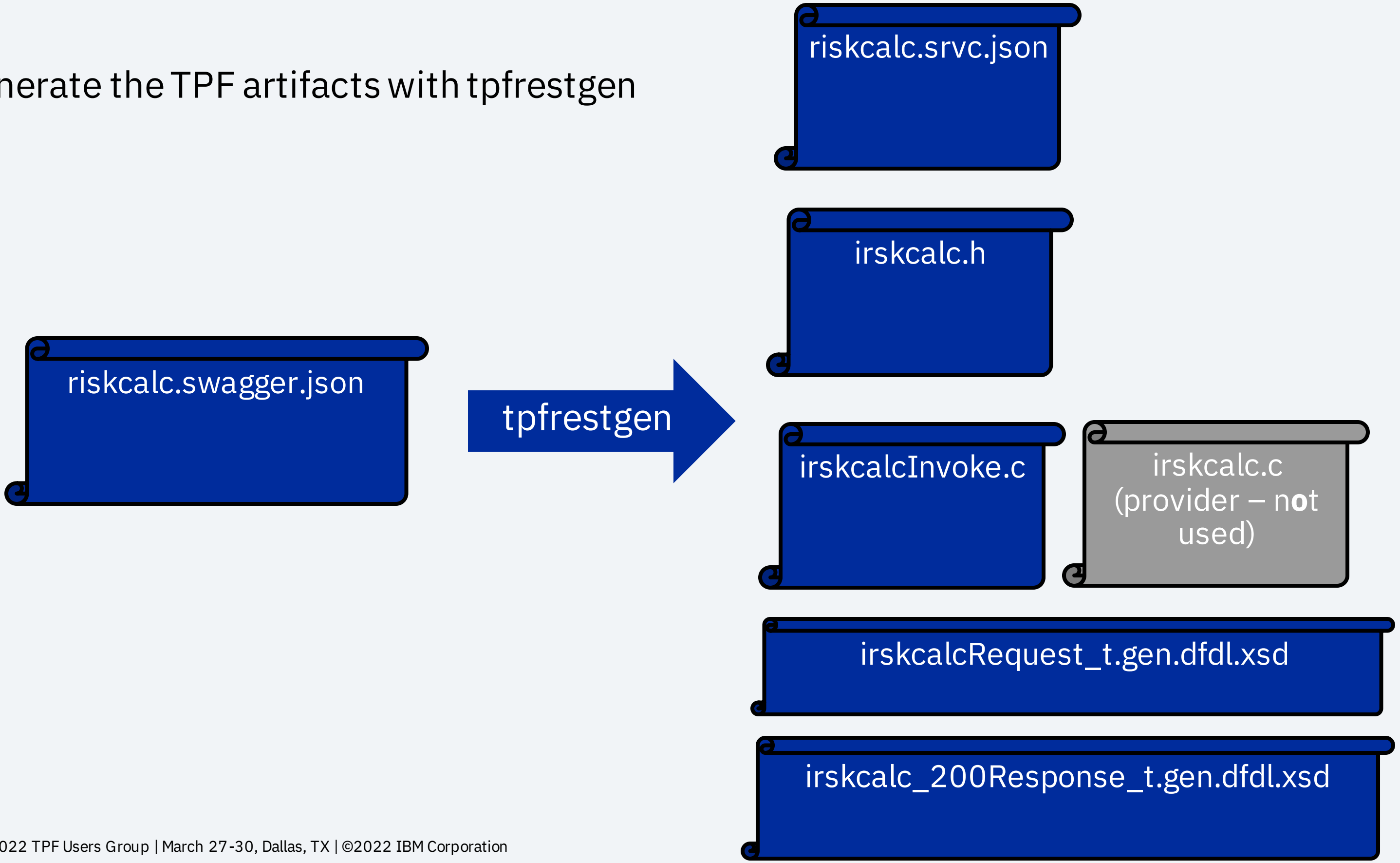
```
paths:
 /analysis:
  post:
   summary: Retrieve namespaces
   operationId: irskcalc
   description: >-
    Calculate risk of insurance for traveller, requires input
    matrix defined at http://flightrisk.ofc/flightinputs/v1
   parameters:
    - in: body
      name: analysis_request
      description: request body
      required: true
      schema:
        $ref: '#/definitions/risk_request'
   responses:
    '200':
      description: normal response
      schema:
        $ref: '#/definitions/risk_response_200'
```

```
risk_request:
  type: object
  properties:
   input_matrix:
    type: array
    minItems: 40
    maxItems: 40
    items:
      type: number
```

```
risk_response:
  type: object
  properties:
   calculated_risk:
    type: number
```

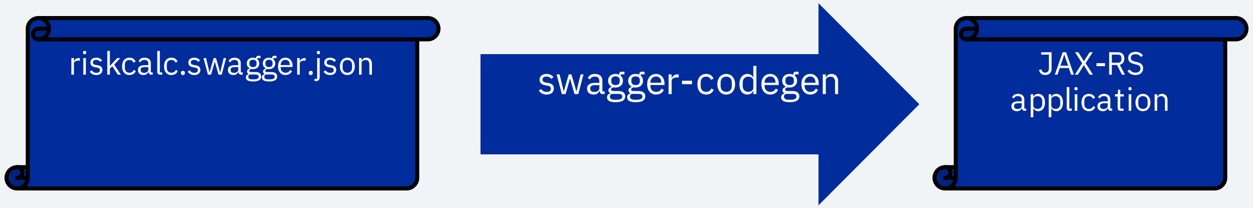
Analytics: Generating consumer

Generate the TPF artifacts with tpfrestgen


riskcalc.swagger.json
tpfrestgen
riskcalc.srvc.json
irskcalc.h
irskcalcInvoke.c
irskcalc.c (provider – not used)
irskcalcRequest_t.gen.dfdl.xsd
irskcalc_200Response_t.gen.dfdl.xsd

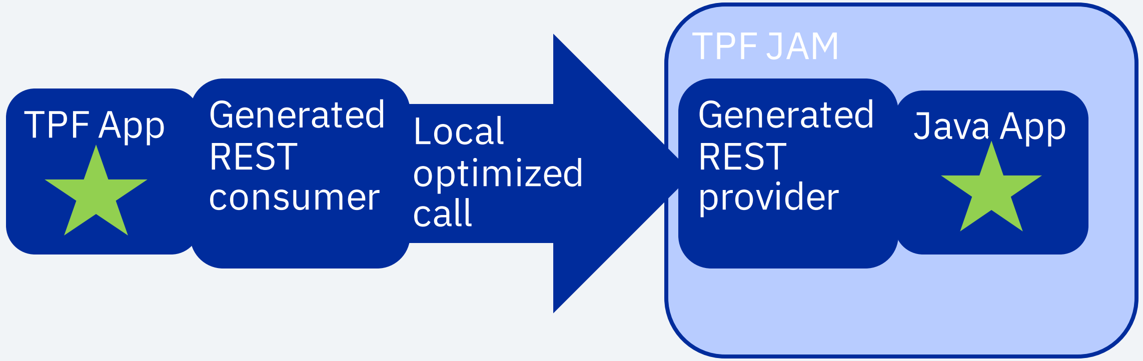
Analytics: Generating producer

Generate the Java artifacts with swagger-codegen or swagger editor to generate the Java producer



Analytics: Generating stubs

Update code paths to include consumer calls and implement producer logic:



Analytics: Deployment steps

- If not already done configure system to at least minimum requirements for Java https://www.ibm.com/docs/en/ztpf/2022?topic=java-configuring-your-ztpf-system
- Define file system storage and mount points required for Java
- Define jam.xml including Java Application in classpath https://www.ibm.com/docs/en/ztpf/2022?topic=descriptors-jam-descriptor
- Enable system monitoring through OtherCommandLineOptions parameter https://www.ibm.com/docs/en/ztpf/2022?topic=guide-monitor-java-applications-across-ztpf-system
- Deploy JAM

Analytics: Testing steps

- Unit testing can generally be done off TPF for Java
- Integration testing on VM
- Initial performance testing / validation on VM with Apache JMeter or load driver
- Configuration testing on “pre-prod”

Analytics: Performance steps

- Using Apache JMeter or TPF application load driver drive sustained workload
- Connect IBM Healthcenter client (available from eclipse marketplace) to JAM
- Use Method Profiling and Garbage collection screens to identify hotspots and garbage collection issues
- Use System monitoring to view JVM performance combined with CDC metrics in a single view

Analytics: Production

- Define number / size of JVMs based on analysis of throughput constrainsts in test
- Configure / adopt JVM system monitoring to support coverage
- Considerations for AutoRecycle YES vs NO
  - AutoRecycle currently restarts on every activation change
  - Frequently activating multiple loadsets is currently better supported through process (manual recycle vs auto recycle)

Implementing Standard Interfaces

Implementing Standard Interface

-Java listeners available for any number of interfaces

-Complementary to Guaranteed Delivery support, except for data subscription instead of publication

-Can easily be integrated into a JAM for consumer model, standard single port interfaces not compatible with current recycle model (for example would need every JVM to have their own unique port)

Use case: Kafka consumer

Just as z/TPF integration with the enterprise is facilitate by data publication, z/TPF may have a requirement to ingest data from the enterprise. Presently there are 2 main capabilities that exist natively on z/TPF: MQ and REST.

Often the enterprise solution can require connectors or special configuration to adopt these as ingress connectors into z/TPF. Java on z/TPF can help you natively read from these enterprise solutions without extra external configuration.

Use case: Kafka consumer

Implementing a Kafka consumer can be done as part of a JAM workload.

While you may wish to implement some REST services to configure or control the client, they are not required to be externalized.

```
/**
 * Entry point into the application
 */
public class KafkaClientApp extends Application {

        static {
                 KafkaClientEngine engine =
KafkaClientEngine.initialize();
                engine.begin();
        }

        @Override
        public Set<Class<?>> getClasses() {
                Set<Class<?>> classes = new HashSet<>();
                classes.add(KafkaClientConfiguration.class);
                return classes;
        }
}
```

Use case: Kafka consumer

In our example, we're going to start with a simple Kafka Consumer in a single thread for simplicity, but you can scale this out dynamically or just via configuration.

There's very few lines of Java code necessary to interact with the Kafka Broker.

```
Properties config = new Properties();
config.put("client.id", InetAddress.getLocalHost().getHostName());
config.put("group.id", "foo");
config.put("bootstrap.servers", "host1:9092,host2:9092");
KafkaConsumer consumer = new KafkaConsumer<K, V>(config);

while (running) {
  ConsumerRecords<K, V> records =
    consumer.poll(Long.MAX_VALUE);
  process(records); // application-specific
  consumer.commitSync();
}
```

Use case: Kafka consumer

There’s a few choices on how to interact with z/TPF.

For a safe design, we want to place the message into MQ for guaranteed receipt.

This gives us two options:

1) Standard Java MQ client
2) Custom REST service

Use case: Kafka consumer

1) Standard Java MQ client
   a. Least code
   b. Network connected to TPF
2) Custom REST service
   a. Best performance
   b. Minor code
   c. Option to use Long-running ECB (stateful services)

Use case: Kafka consumer

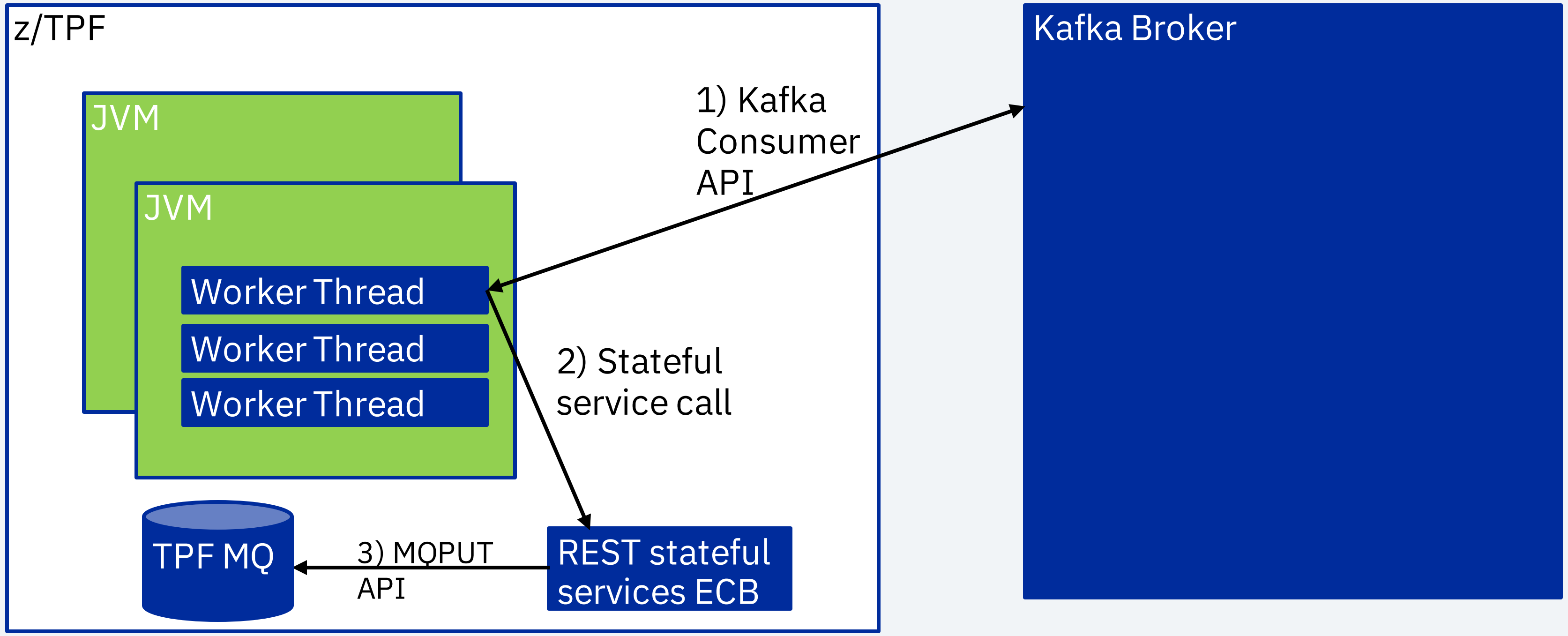
Questions to consider when deciding implementation:

1) What’s the expected usage
2) Do I care about latency?
3) Do I care about CPU overhead
4) Do I care about scalability
5) How quickly do I need to deploy this

Use case: Kafka consumer

1) Standard Java MQ client
   a. Least code
   b. Network connected to TPF
2) Custom REST service
   a. Best performance
   b. Minor code
   c. Option to use stateful services ECB

Technical Details



Use case: Kafka consumer

Creating a stateful service:

This is just a REST provider with a providerType set to StatefulProgram in the service descriptor. This changes the behavior so that calls from the same Java thread will all be processed from a common ECB instead of creating a separate ECB for each request.

The advantages is decreased overhead of MQ initialization as well as ECB creation as a single Java thread is expected to make a multitude of REST calls (potentially one per message)

Use case: Kafka consumer

Making an optimized REST service call from a Java application

https://www.ibm.com/docs/en/ztpf/2022?topic=java-calling-local-ztpf-application-services-from

Using the CXF client generated from swagger-codegen you can use the same procedure from earlier:

1) Define interface
2) Implement service
3) Invoke service from application

Conclusion

Leveraging the capabilities of Java on the z/TPF platform, you can provide enterprise integration capabilities using built-in as well as custom solutions with minimal coding and effort.

Thank you