Simon Sved

TRANSFORMATION OF EVENT STREAMING APPLICATION FROM PYTHON TO JAVA

Technology and Communication
2024

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Simon Sved
Title	Transformation of event streaming application from Python to Java
Year	2024
Language	English
Pages	33
Name of Supervisor	Chao Gao

This thesis focuses on converting an event streaming application from Python to Java. The client of the thesis project, has an application that utilizes event streaming to send and receive data from the SAP ERP software. This application maps the data to a more suitable format and sends it out to subscribers.

The aim of the thesis is to transform this application to use Java and the Spring framework instead of Python and the Django framework. The motivation for converting the application is to make future maintenance easier. The development team using this application is more experienced with Java and Spring and therefore, the client wanted to convert the application to simplify the maintenance.

The thesis project fulfilled its objective of transforming the application from Python to a fully working Java application utilizing Azure Service Bus for sending and receiving the data with event streaming.

Keywords Software development, Java, Spring, event streaming

CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

API Application Programming Interface

ASB Azure Service Bus

ERP Enterprise resource planning

ESB Enterprise Service Bus

HTTP Hypertext Transfer Protocol

IDoc Intermediate Document

KiB Kibibyte, 1024 bytes

SAP System Applications and Products in Data Processing – ERP software

1 INTRODUCTION

The objective of this project was to transform and improve an existing application, Customers Service Order API, for the client company. This application retrieves data from the ERP software SAP, which it then transforms into a more suitable format. The application utilizes event streaming by using the Azure Service Bus to send and receive this data.

The problem the client company faced was not that the application was not functioning well, but that the application was written in a language that the development team was less familiar with. Due to changing requirements in the process and the data, the application needed to be maintained often. Because of this, the decision was made to convert this application from Python to Java. This conversion would help the current developers, who are already experienced in Java, to maintain the application more easily in the future.

2 TECHNOLOGIES

An important aspect to consider when choosing what technologies to use in an enterprise application is the developer team's proficiency in various technologies. The choice of using Python and the Django framework at the time was not a bad one, but the developer team at the client company that are the main subscriber of the application have more experience in Java and Spring. Because of the evolving requirements of the system that requires this critical data from the API, the API had to be maintained often, and the developers felt like their lack of expertise in Python lead to decreased productivity when they had to make changes. Therefore, the choice of converting the API from Python and Django to Java and Spring was taken.

2.1 Previous Technologies

2.1.1 Python

The previous application was made using the Python programming language. The Python programming language was designed in the late 1980s and was first released in 1991. The quote below succinctly describes the Python programming language:

> "Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and

> the extensive standard library are available in source or binary form without charge for all major platforms and can be freely distributed.” (Python Software Foundation, n.d.)

2.1.2 Django Framework

The Django framework was also used in conjunction with Python. It is a high-level, free, and open-source Python web framework that can help to develop web applications faster. It handles many of the complexities associated with web development so the developer can concentrate on writing the application itself. (Django Project, n.d.)

2.1.3 Azure Functions

Azure Functions is a serverless solution from Microsoft that minimizes the amount of code required, reduces infrastructure maintenance, and lowers costs. This service allows developers to spend less time and effort on deploying and managing servers, as the cloud infrastructure automatically handles the necessary resources to keep the application running efficiently. (Microsoft, 2023)

2.2 Technologies Used by Both Applications

2.2.1 SAP

SAP is a German software company, most known for their SAP ERP software solution. The company was founded in 1972, and its first commercial product was then released one year later. ERP, or enterprise resource planning refers to a software system that can help an enterprise organize and manage their day-to-day business processes and data, whether that is in finance, supply chain or human resources. SAP is one of the most popular ERP solutions. (SAP, n.d.).

SAP ECC is the current solution that the client is using as their ERP system. It is one of the most critical and widely used systems in client, and most if not all business processes in the client company utilize the SAP system in some way.

SAP uses IDoc (Intermediate Document) as their document format to exchange business data between applications. The IDoc structure consists of three parts, or records: The control record, which contains information like the type of the IDoc, the message type and date and time of creation. The data record, as the name implies, contains the actual business data, which is stored in different segments. The status record contains information about the various processing states of the IDoc.

2.2.2 PostgreSQL

PostgreSQL is a free and open-source object-relational database management system. The PostgreSQL project was started in 1986 and is nowadays one of the more popular choices for SQL databases for both enterprises and smaller projects. PostgreSQL have many features such as tablespaces, asynchronous replication, and nested transactions. It has many extensions and a big development community thanks to it being open source. (AWS, n.d.)

2.2.3 Azure Service Bus

Azure Service Bus is a fully managed enterprise message broker that offers message queues and publish-subscribe topics. It serves the purpose of decoupling applications and services from each other, providing several advantages in the process such as load balancing, safely routing and transferring data and coordinating transactional work that requires a high degree of reliability. (Microsoft, 2024a)

The Azure Service Bus transfers so-called messages. These messages can contain data of any kind, such as JSON, XML, or plain text. The service supports two primary messaging structures: queues and topics.

Queues offer a one-to-one, First In, First Out (FIFO) message delivery. A producer sends a message, and the consumer receives it. Topics, on the other hand, offer one-to-many message delivery using a so-called publish and subscribe pattern.

This architecture allows a single message published to a topic to be received by multiple subscribers. In our case the topic-based architecture was used.

(Microsoft, 2024b)

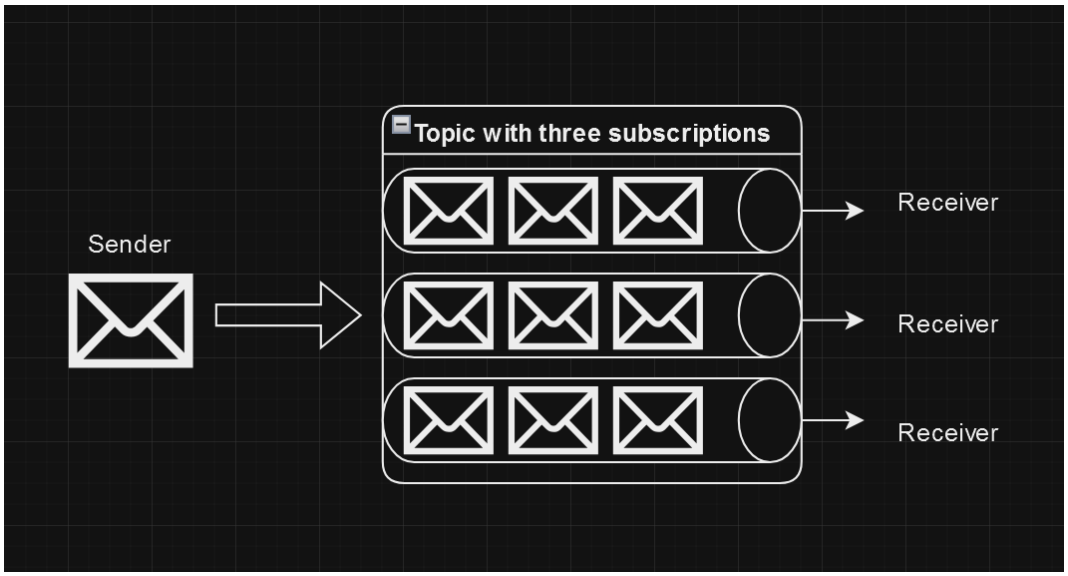

Figure 1. Topic with three subscriptions

2.2.4 SonarQube

SonarQube is a self-managing and automatic tool that reviews the code to ensure that it is so-called clean code. Clean code is code that is easily understood, portable, secure and without bugs or code smells. Sonar can be configured to fit the development team's standard of code to ensure consistency across the whole project.

SonarQube can analyse over thirty different programming languages and can integrate into the Continuous Integration (CI) DevOps pipeline. SonarLint is an advanced linter for the IDE that checks the code as it is being written, making it possible to find and fix issues before committing.

SonarQube uses so-called quality gates, which are a way to enforce code quality. It informs the developer if the code is clean and meets the project's standard and is ready to be merged. (SonarSource, n.d.)

2.3 New Technologies

2.3.1 Java

Java is a popular object-oriented programming language. It is used for many kinds of applications, including mobile applications and web applications. It was first invented in 1995 as a part of Sun's Java platform and the first public version was released in 1996. Java can be written once and be run on any platform. This is because the Java code being written is compiled into an intermediate language called bytecode. This bytecode can then be run on any system with a Java Virtual Machine (JVM) installed. (Microsoft, n.d.)

2.3.2 Spring Boot

The Spring Framework is one of the most popular frameworks in Java. It is an open-source framework that can help to create stand-alone, production-grade applications. The Spring Framework is used because it removes many of the complexities that are associated with application development in Java. It offers built-in support for many common tasks that an application needs, such as data binding, type conversion and exception handling.

Spring boot is a module that is an extension of the Spring Framework. Spring Boot helps you develop applications even faster than with just the normal Spring Framework. Applications are initialized with pre-set dependencies by autoconfiguration, saving the trouble of manually configuring them. Spring Boot also takes a more opinionated approach when it comes to the dependencies to help to get started with development faster. This approach is less flexible than the unopinionated, but also helps to avoid errors since it is based on best practices. One more thing that Spring Boot helps you with is to create standalone Spring applications that can run immediately without extra annotations or configuration. (IBM, n.d.)

2.3.3 Docker

Docker is a platform that allows developers to distribute and run applications. It does this by letting you package and run the application in an environment called a container, which is a lightweight, standalone package of all the elements you required to run your application, so you can run it anywhere. This includes the code itself, the runtime, libraries, and system tools the application needs to function. When the application is tested and ready, it can be packaged into a container and be deployed to the production environment to get the application running. (Docker, n.d.)

3 BACKGROUND

3.1 Description of Previous Application

Customer service orders contain data such as what engine the service order is meant for, required operations, designated workshop, customer details, and scheduled start and end dates. These orders are created and stored in the SAP system, where they are linked with other objects.

The data that these orders contain are required by an internal application. This application serves as a digital system for the workshops that repairs and maintains marine engines. Not all data from the orders is required by the internal application, so the data is filtered to meet the system's needs. Although this filtering could be integrated into the internal application, it was decided to keep it separate. This decision was made to accommodate potential future applications and systems that might require the same or differently formatted data from these orders, ensuring flexibility and decoupling of systems.

The order data that is produced in SAP is transferred to ESB, an internal enterprise service bus, which then transfers the data as a message to a topic in the Azure service bus which the Customer Service Order API is subscribed to. The application, which is deployed to Azure Functions, consumes the message, and then transforms the data inside the message to the required format. The transformed message is then sent back to the same topic in the Azure Service Bus. The internal application is also subscribed to this topic and only consumes the already transformed messages.

3.2 Motivations for Converting the Application

The primary reason for transitioning the application from Python and the Django framework to Java and the Spring framework was to facilitate easier future maintenance. The internal application which is the main consumer of the data, the, was developed using Java and Spring. The development team for the internal application has extensive experience with this technology stack and therefore it was reasoned that it would be easier for them to maintain the Customer Service Orders API if it were developed using Java rather than Python. Consequently, they would not have to rely on a separate Python developer maintaining the application if more complex changes had to be made.

Another reason behind converting the application was the use of Azure Function application for hosting, even if this did not have to do with the programming language choice. The previous application used an Azure Function app to reduce the costs of hosting the application. It was reasoned that using a Kubernetes cluster hosting solution would be more manageable than having Azure manage the application lifecycle. Although this would be at a slightly higher cost, the im-portance of the application meant the cost was justifiable. By using the Kuber-netes cluster, the event streaming would also be more easily monitored.

4 PROJECT REQUIREMENTS

The primary objective of the project was to ensure that the new application replicated the functionality of the original application. This meant that both applications would produce identical outputs given identical inputs. This was crucial to avoid any modifications to the internal system consuming the formatted data.

One important requirement was regarding the usage of the programming language and framework to implement the application. While the previous application utilized Python and Django, the decision was made to switch to Java and Spring Boot to facilitate easier maintenance.

Another requirement was to continue using Azure Service Bus and PostgreSQL. The application should maintain its use of Azure Service Bus for handling the messages sent from SAP through ESB. This would also apply to sending data, where Azure Service Bus would be used to publish the formatted data to the topic subscribed to by the consuming application.

The application would also continue to use the existing PostgreSQL database, which already has a suitable model with only a few exceptions. The database contains data from previous events saved during the usage of the previous application, thus eliminating the need for data migration.

5 IMPLEMENTATION

5.1 Analysis of Previous Application

Before beginning the implementation of the new application, a thorough analysis of the previous application and the entire system surrounding it was done. This was done to identify the core features of the application and how it worked in conjunction with the event streaming platform to receive and send the messages via the Azure Service Bus.

The analysis process began with taking a comprehensive overview of the entire application and the workflow the system supported. This included examining the application architecture, its components, modules and their interactions with each other and other systems. Additionally, the dependencies and libraries used in the application were also analysed. This was carefully considered as to identify if they could be replaced with existing Java-based dependencies and libraries, or if additional development time would be required to create custom implementations to replicate the features of the previously used libraries.

During the rewrite process, the application was critically evaluated to identify potential improvements. Both the data objects and the application's functionality were scrutinized, leading to minor enhancements. For instance, improvements were made in the relationships between data entities and in the functionality related to event production.

5.2 Event Streaming Flow

The workflow starts with a user creating a customer service order inside the SAP ERP software. The content inside the customer service order is then sent out from SAP in the form of an IDoc to the Enterprise Service Bus (ESB). ESB is an internal platform which is used for integrating various internal and external applications, systems, and partners to different businesses internally. In this scenario, it acts as an intermediary between SAP and the Azure Service Bus.

Once ESB receives the event from SAP, it processes the event before sending it to the Azure Service Bus. ESB does not directly forward the event to the Azure Service bus; instead, the event first passes through an internal API called the Event Producer API. However, before ESB can access this API, it must be authorized within Microsoft Entra ID using a service principal. A service principal is an identity created for applications to securely access Azure resources, acting as a user identity for authentication and resource access. The service principal must be authorized with OAuth2.0 to gain access to the API. If the service principal has the correct role for the topic it wants to produce to, the Event Producer API will then produce the event to the Azure Service Bus to the designated topic. The event is then distributed to the subscribers of that topic.

For the events to be received, consumers follow a similar flow. With a service principal that has the correct roles for consuming events within Microsoft Entra ID, the consumer can get authorized to call another internal API, the Event Consumer API. The Event Consumer API provides controlled access to the consuming side of the event streaming as the name suggests and has an endpoint for a HTTP GET call to consume the events in the topic you are subscribed to. The API checks the event consumer's role and verifies if it is allowed to access the target topic.

5.2.1 Message Data Model

The messages that are sent and received on the Azure Service Bus follow a specific structure. There are two message types, standard and chunked messages. Both types contain three main components which are shown in figure 2:

1. The payload, which holds the data content of the message
2. The metadata field automatically populated by the ASB that contains information describing the payload.
3. Custom properties, which is used for internal purposes, such as filtering topic messages to subscriptions based on certain rules

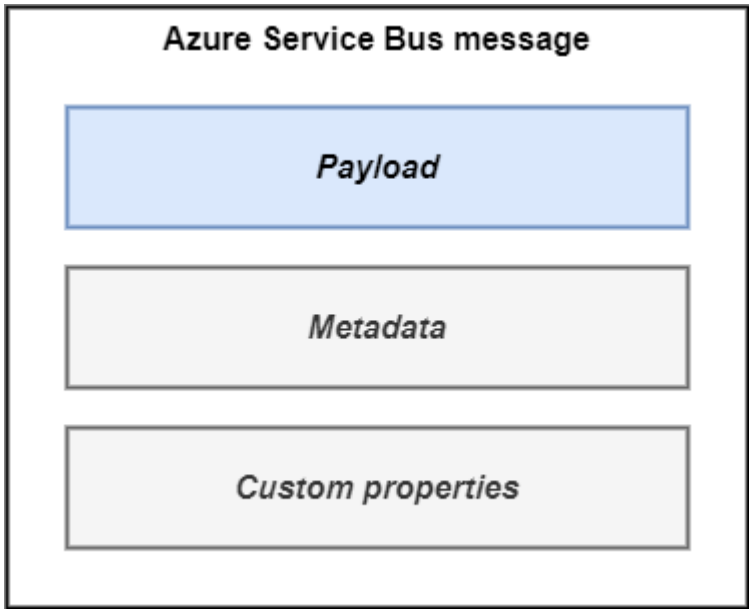

Figure 2. High-level overview of message object's structure

The standard message structure allows a maximum message structure of 256KiB. If the application were to use the standard message structure, and the message would exceed 256KiB, the messages would be rejected. That is why the chunked message type is the one that is produced from, in case of the rare event that the message would exceed the maximum amount.

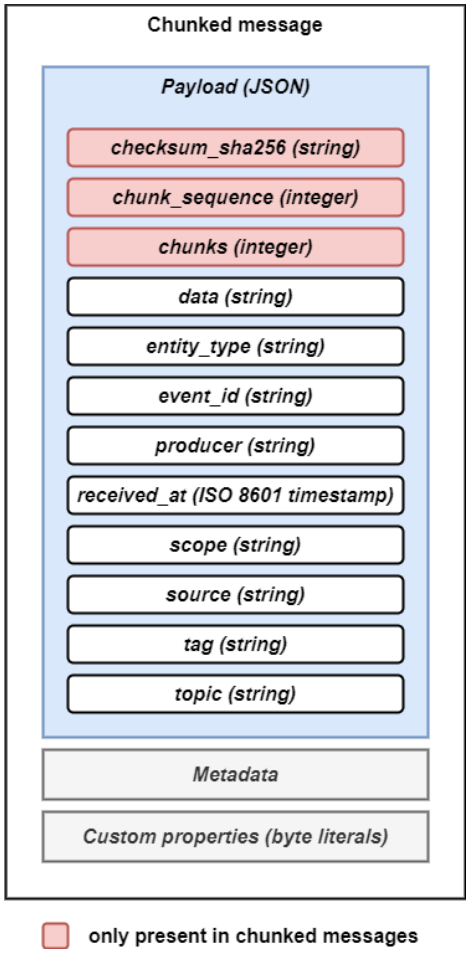

Figure 3. Message structure of a chunked message

The chunked message feature is a custom in-house abstraction that is built on top of the standard ASB features. The chunking feature works by first base64 encoding the payload, then splitting it into one or more chunks with a maximum size of 200KiB. Each chunk is encapsulated into an ASB message, with one chunk equating to one message. The checksum_sha256 field groups the chunked messages together, ensuring all chunks belonging to the same message have the same unique checksum. In figure 3 you can see the message structure of the chunked message, specifically the payload content.

5.2.2 Event Streaming Overview

Figure 4 provides a high-level overview of the entire process. It illustrates how all the various systems interact with the Event Producer and Event Consumer APIs, and how events are sent and received from the ASB until finally reaching the subscribing application.

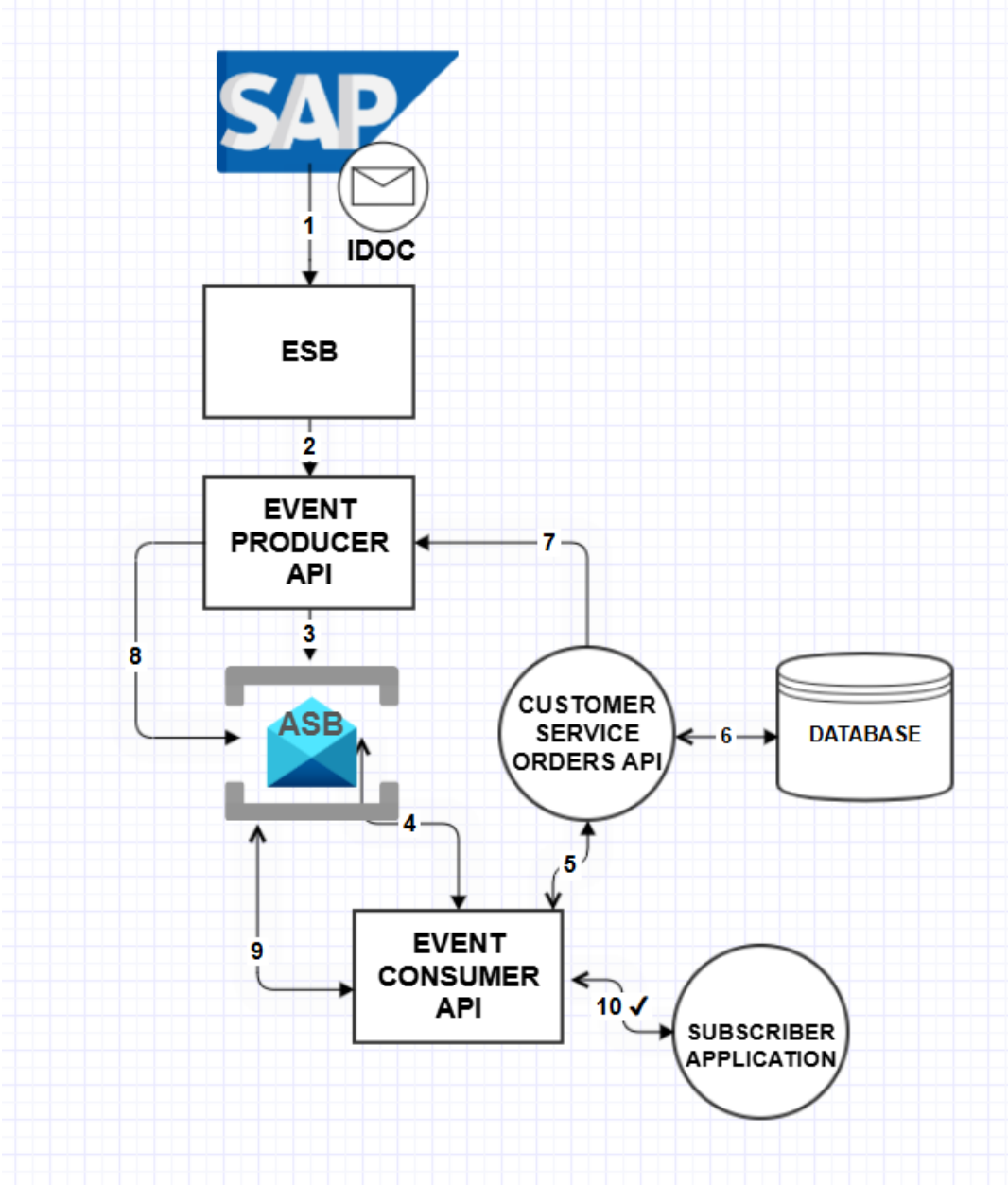

Figure 4. Flowchart of event streaming process

5.3 Data Models

To begin the implementation process of the application in Java and Spring Boot, the data entities were defined in the application. Since the database and its data model were to be reused, this was a logical starting point. By examining both the database schema and the Django models in the Python application, which represent tables in a schema, one could easily understand the data model. With the help of Spring Data JPA, the tables could be represented by Java objects known as entities in JPA, analogous to Django's models. Within these entities, fields represented the table columns along with their respective data types. Figure 5 illustrates a segment of a JPA entity, with two columns represented by fields. This is in comparison with figure 6 which shows how part of a Django model looks like, which is representing the same table as the JPA entity in figure 5.

```
@Getter
@Setter
@Entity
@SequenceGenerator(
    name="customerserviceordersubscriber_id_seq",
    sequenceName="customer_service_order_api_subscriber_id_seq",
    allocationSize=1)
@Table(name = "customer_service_order_api_subscriber")
public class Subscriber {

    @Id
    @GeneratedValue(
        strategy = GenerationType.SEQUENCE,
        generator="customerserviceordersubscriber_id_seq"
    )
    @Column(name = "id", nullable = false)
    private Integer id;

    @NotNull
    @Column(name = "email")
    private String email;
```

Figure 5. Code snippet of JPA entity

```python
class Subscriber(models.Model):
    """Information about users subscribing to events and what they want."""

    email = models.EmailField(
        null=False,
        blank=False,
        max_length=255,
        help_text="Email address of the subscriber requesting support or update.",
    )
    name = models.CharField(null=False, blank=False, max_length=255, help_text="Subscriber name.")
    can_receive = models.BooleanField(
        null=False,
        blank=False,
        help_text="Set to true / tick this box if the subscriber can receive events from Customer
    )
    registered_at = models.DateTimeField(
```

Figure 6. Code snippet of Django model

The database schema comprises five tables containing the actual customer service order data from the IDoc, which includes data related to the order itself, the order's coordinator, operations, and the ship-to and sold-to address data.

Additionally, there are three tables related to the event streaming. The first is the event table, which stores details of consumed events, including the base64 encoded payload, the event producer, the event tag, and the time of receiving the event. The second table, forwards, records produced events, with information such as the forwarding time and the event data content. The third table is the subscriber table, which lists the topic subscribers and their related information. There were certain tables associated with Django that did not require recreation as entities during the refactoring process.

5.4 IDOC Processing

The IDoc transmitted from SAP is initially in a non-human-readable format before processing. The primary function of the application is to transform this IDoc into a human-readable format suitable for the actual consumers. To facilitate the deserialization of the IDoc's JSON data into Java objects for database storage and forwarding, the Jackson JSON library was employed.

The Jackson library was used to traverse through the JSON structure and extract the necessary values from the IDOC data. These values were then mapped to specific fields within our entity classes representing the database tables Custom deserialization classes were created for each entity class to handle this process. These entity classes were then annotated with Jackson annotations to ensure that they were deserialized using their corresponding custom deserialization classes whenever the IDOC data was processed.

Figure 7 shows an example of how the IDoc data looks like before and after processing it. Some fields that were not needed were removed and the other ones were renamed to accurately describe what values they hold.

Figure 7. IDoc JSON formatting example

5.5 Event Streaming Implementation

One vital part of the application functionality was the ability to consume and produce events to and from the Azure Service Bus. To be able to communicate with the Azure Service Bus, an internal library was used. This library utilizes the Spring Cloud Azure for Java to communicate with ASB within the internal ecosystem.

The library implements the builder pattern to construct “AzureAsyncSasReceiver”, which is a class in the library that manages subscription credentials and registers a handler that we define within our application to process received messages. The handler creates a receiver client that extracts and processes messages from the Azure Service Bus efficiently.

When a message is sent to the Azure Service Bus and processed through the library, the event handler implementation receives this message. Initially, the message is deserialized into an object of the event class. The event is then validated by ensuring that the checksum is not blank and that the source and producer of the event match the specified criteria. Once validated, the event is saved to the database and processed.

The processing of the event involves decoding the base64 encoded data, which is then deserialized into five distinct entities representing the order itself, the order’s coordinator, operations, and the ship-to and sold-to address data. These entities are saved to the database, and the entire order is then forwarded.

The implementation in Python uses much of the same variables, with some different naming conventions. It implements an event consumer client which is defined in the library with the necessary configuration. It can then be used to listen to the events with its own event handler implementation, much like in the Java version.

The forwarding process is similar to the consumption process. Subscribers to whom the data will be sent are first retrieved from the database. The order is serialized into the format expected by the subscriber using the Jackson library. The serialized data is then sent to a HTTP POST endpoint in the Event Producer API, which will create and send an event containing this data to our ASB topic. The configuration used to send events to the Event Producer API is much like the receiver configuration. The message structure that is used is like the one shown in Figure 4, except it is not chunked in this case. Once the message is sent to the Event Producer API, the Customer Service Order API's role is complete, and the subscriber can consume the message from the ASB.

5.6 DevOps

To be able to easily develop and deploy the application various systems were used. For version control the distributed version control system Git was used. This allowed working on different parts of the application at the same time by using so-called feature branches. Bitbucket was the tool used to host the actual code. Bamboo was used alongside it and integrated seamlessly with Bitbucket. Bamboo is a continuous integration and deployment tool that compiles the code, runs tests, and packages the code. By creating a Bamboo specification file in our project, we can define what we want Bamboo to do and for which branches.

The application itself was containerized using Docker. This allowed us as previously stated to have a single container with everything we need for running the actual application. With a dockerfile inside our project we can specify what base image we want for the application as well as the commands for building the image.

Kubernetes was used to manage the containerized application. Kubernetes takes care of the lifecycle of the containers, including deployment and scaling. To easily manage Kubernetes an internal Helm chart was used. This chart is a configuration of Kubernetes that can easily be reused. To manage and deploy the Kubernetes

clusters, a tool called Rancher was used. This tool lets us easily manage the clusters with a user-friendly dashboard among many different environments.

5.7 Testing of Application

Testing an application is a vital part of the software development lifecycle and provides many important benefits. It ensures that your application is functioning properly and according to requirements. By writing proper tests you can identify defects if you were to add or modify something in the application. It also helps you reduce unnecessary risks when it comes to the software quality and security. (IEEE Computer Society, n.d.)

The application was tested using unit tests, integration tests as well as manual tests. The Java testing framework used for the unit tests was JUnit 5. To mock external dependencies and their functionalities during unit tests, the mocking framework Mockito was used. The unit tests focused on specific methods without the use of a database or interactions from any other modules, following a white box testing approach. In the example shown in figure 8, you can see how Mockito's ArgumentCaptor is used to capture the value that is passed to the save method of the CustomerServiceOrderRepository. By using the AssertJ library and its assertions we then verify that the captured object has the expected values. In figure 9 instead we can see how a similar test is done in the original Python application. This test also tests that the mapping of the order data is correctly done.

```java
@Test
void when_updateCustomerServiceOrder_withValidOrder_then_saveOrder() throws IOException {

    String testOrder = readFileToString(
        Paths.get( first: "src/test/resources/files/testOrders.json").toFile(), UTF_8);

    customerServiceOrderService.updateCustomerServiceOrder(testOrder);

    verify(customerServiceOrderRepository, times( wantedNumberOfInvocations: 1))
        .save(customerServiceOrderArgumentCaptor.capture());

    CustomerServiceOrder order = customerServiceOrderArgumentCaptor.getValue();
    assertThat(order.getPlantCode()).isEqualTo( expected: "PLANT");
    assertThat(order.getWorkCenter()).isEqualTo( expected: "WORK-CENTER");
}
```

Figure 8. Unit test example in Java

```python
class MappingTest(TestCase):
    def test_mapping(self):
        with (open(os.path.join(Path(__file__).parent.parent,
                "fixtures", "someOrders.json"),
                encoding="utf8")
              as some_orders_file,
              open(os.path.join(Path(__file__).parent.parent,
                "fixtures", "expectedOrderFromEventForwarder.json"),
                encoding="utf8")
              as expected_order_file):
            some_orders = json.load(some_orders_file)
            reference_obj = json.load(expected_order_file)
            self.compare_idoc_output_vs_reference(some_orders[0], reference_obj[0])
```

Figure 9. Unit test example in Python

For the integration tests, the spring-boot-starter-test module was used to provide us with the necessary libraries for the tests, including the JUnit, Spring Test, Spring Boot Test and Hamcrest libraries. Since the application interacts with the Azure Service Bus for event consumption and production, a dedicated test controller was created to test the entire system, including the database.

To use databases within the integration tests, the Testcontainers library was used. This is an open-source library that provides you with a way of running real database instances inside Docker containers. By using Testcontainers, there is no need to rely on mocking or in-memory databases that might not function the same way as your real database. In this case, Testcontainers provided a Postgres database identical to the real one. By using Spring Data JPA the test database can automatically be initialized with your database schema according to your entities. Once initialized, the database is populated with data using SQL files, which can be specified for each test. In the test method, it is possible to assert that your tables contain the expected data. Figure 10 illustrates a simple example of an integration test used in the application. This test mimics the scenario of an event being consumed from the topic using the test endpoint. This approach ensures that the integration between the database and the event consumption mechanism is functioning as expected.

```
@Test
@Sql(scripts = {
    "classpath:files/sql/clean.sql",
    "classpath:files/sql/insertIntoEvent.sql",
    "classpath:files/sql/insertIntoSubscriber.sql"},
    executionPhase = Sql.ExecutionPhase.BEFORE_TEST_METHOD)
void testUpdate() throws Exception {

    mockMvc.perform(post( urlTemplate: "/api/test-update")
            .contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk());

    assertThat(countRowsInTableWhere(jdbcTemplate, TBL_CUSTOMER_SERVICE_ORDER,
            String.format("plant_code='%s'", "plant_code")),
        equalTo( operand: 1));
}
```

Figure 10. Example of integration test

To thoroughly test event consumption from the Azure Service Bus, manual tests were also conducted. Mocking or integrating the ASB functionality posed significant challenges, and therefore manual tests were done instead. To achieve this an internal tool called Event Client was used, which is a graphical user interface that

lets the user easily consume and produce events to and from topics. This tool utilizes the Event Producer API and Event Consumer API, allowing the simulation of ESB sending an event to the ASB. The application was then run to verify if the event was saved, formatted, and sent back to the ASB topic according to the requirements.

6 CONCLUSIONS

Transforming an application from one programming language and framework to another depends on the complexity of the original application. The implementation process began with a thorough analysis of the original application and its components, with a focus on maintaining identical functionality whilst developing the new application.

In this project, the main functionality of the application was mapping the customer service order received from the SAP ERP system to a more suitable format. This part of the application was not too difficult, although there were careful considerations on what library to use to implement this.

The more challenging part of this transformation was the understanding and implementation of event streaming to send and receive the data. As a developer, I was not familiar with event streaming or Azure Service Bus prior to this project. This project has provided valuable insights into how event streaming works in practice and how the Azure Service Bus functions as a message broker.

The transformation of the application was a success and will replace the original Python application. This will, as previously stated, help the development team maintain it in the future. The other changes related to its deployment should also improve the management and logging of the application.

In conclusion, the successful transformation of the application not only ensures continuity in functionality but also enhances the maintainability and operational efficiency of the system. The experience gained from this project has been invaluable in understanding advanced concepts such as event streaming and transformation of existing applications.

REFERENCES

AWS. (n.d.). *What is PostgreSQL?* Retrieved May 11, 2024, from AWS: https://aws.amazon.com/rds/postgresql/what-is-postgresql/

Django Project. (n.d.). *Why django?* Retrieved October 25, 2024, from Django Project: https://www.djangoproject.com/start/overview/

Docker. (n.d.). *Docker overview*. Retrieved May 11, 2024, from Docker: https://docs.docker.com/get-started/overview/

IBM. (n.d.). *What is Java Spring Boot?* Retrieved May 13, 2024, from IBM: https://www.ibm.com/topics/java-spring-boot

IEEE Computer Society. (n.d.). *The Importance of Software Testing*. Retrieved October 27, 2024, from IEEE Computer Society: https://www.computer.org/resources/importance-of-software-testing

Microsoft. (2023, May 24). *Azure Functions overview*. Retrieved May 11, 2024, from Microsoft: https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview

Microsoft. (2024a, December 5). *Service Bus queues, topics, and subscriptions*, A. Retrieved December 8, 2024, from Microsoft: https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-queues-topics-subscriptions

Microsoft. (2024b, February 24). *What is Azure Service Bus?* Retrieved May 11, 2024, from Microsoft: https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview

Microsoft. (n.d.). *Azure*. Retrieved May 12, 2024, from What is Java?: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-java-programming-language

Python Software Foundation. (n.d.). *What is Python? Executive Summary*. Retrieved May 7, 2024, from Python: https://www.python.org/doc/essays/blurb/

SAP. (n.d.). *What is SAP?* Retrieved May 9, 2024, from SAP: https://www.sap.com/about/what-is-sap.html

SonarSource. (n.d.). *SonarSource*. Retrieved May 11, 2024, from SonarQube Documentation: https://docs.sonarsource.com/sonarqube/latest/