Security Enhanced Distribution Generator for a Debezium Server

Generátor distribuce Debezium serveru s rozšířenými možnostmi zabezpečení

Bc. Ondřej Babec

Master’s thesis
2024

Tomas Bata University in Zlín
Faculty of Applied Informatics

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav elektroniky a měření

Akademický rok: 2023/2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Ondřej Babec**
Osobní číslo:	**A22753**
Studijní program:	**N1032A020003 Bezpečnostní technologie, systémy a management**
Specializace:	**Bezpečnostní technologie**
Forma studia:	**Kombinovaná**
Téma práce:	**Generátor distribuce Debezium serveru rozšířenými možnostmi zabezpečení**
Téma práce anglicky:	**Security Enhanced Distribution Generator for a Debezium Server**

Zásady pro vypracování

1. Vysvětlete základní pojmy a účel Debezium serveru.
2. Navrhněte generátor distribuce.
3. Věnujte pozornost zabezpečení.
4. Vytvořte front-end aplikaci pro Rest API.
5. Proveďte unit testy i integrační testy.
6. Připravte automatizaci nasazení API a front-end do Kubernetes.
7. Zdokumentujte a popište možnosti využití v bezpečnostních systémech.

Forma zpracování diplomové práce: **tištěná/elektronická**
Jazyk zpracování: **Angličtina**

Seznam doporučené literatury:

1. MURRAY, Nathan, Felipe COURY a Ari LERNER. Ng-book: The Complete Guide to Angular. 11. Createspace Independent Publishing Platform, 2018. ISBN 978-1985170285.
2. POULTON, Nigel. The Kubernetes Book. 1. Nigel Poulton Ltd, 2023, 2023. ISBN 9781916585195.
3. Debezium. Debezium [online]. 2023 [cit. 2023-11-02]. Dostupné z: https://debezium.io/
4. OpenAPI Specification v3.1.0. LINUX FOUNDATION. OpenAPI Specification v3.1.0 [online]. 2021 [cit. 2023-11-02]. Dostupné z: https://spec.openapis.org/oas/v3.1.0.
5. TURNBULL, James. The Docker Book: Containerization is the new virtualization. 1. James Turnbull, 2014. ISBN 978-0988820234.
6. NARKHEDE, Neha, Gwen SHAPIRA a Todd PALINO. Kafka: The Definitive Guide. 1. O'Reilly Media, 2017. ISBN 9781491936160.
7. GAMMA, Erich, Richard HELM, Ralph JOHNSON a John VLISSIDES. Design Patterns: Elements of Reusable Object-Oriented Software. 1. Addison-Wesley Professional, 1994. ISBN 978-0201633610.
8. Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/info/rfc9110>.

Vedoucí diplomové práce: **Ing. Lukáš Králík, Ph.D.**
Ústav bezpečnostního inženýrství

Datum zadání diplomové práce: **20. listopadu 2023**
Termín odevzdání diplomové práce: **28. května 2024**

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
L.S.
①

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

Ing. Milan Navrátil, Ph.D. v.r.
ředitel ústavu

Ve Zlíně dne 1. prosince 2023

ABSTRAKT

Žijeme ve světě, kde jsou všechna data na dosah ruky. Otázkou je, jak tato data centralizovat a použít je tak, abychom dostali co největší přidanou hodnotu. Tato práce se pokouší přivést velmi známou technologii ve světě CDC (Debezium) do světa bezpečnosti. Tato práce obsahuje návrh a implementaci nového systému, který umožní vytvářet distribuce Debezium Serveru na míru, přesně pro potřeby daného uživatele. Nakonec práce nabídne několik možností budoucího nasazení takové distribuce pro využití v bezpečnosti.

Klíčová slova: Debezium Databáze Redis AI ML Angular Java Spring

ABSTRACT

We live in a world where we have all the data at the hand's reach. The question is how to centralize the data and use them to generate additional value. This thesis aims to bring the well-known change in data capture technology (Debezium) into the security world. This work designs and implements a new system that will allow the building of custom distribution of the Debezium Server, which will be tailored specifically for the user. Ultimately, it proposes several use cases for the Debezium Server in the security realm.

Keywords: Debezium Database Redis AI ML Angular Java Spring

Acknowledgements

I would like to thank my supervisors, Ing. Lukáš Králík, Ph.D. from UTB FAI, and Ing. Jiří Pechanec from Red Hat Czech s.r.o for guidance and providing valuable feedback. Also, I would like to thank my colleagues, Ing. Jakub Stejskal, and Bc. Miroslav Jaroš for their time during the thesis corrections.

> The optimist thinks this is the best of all possible worlds. The pessimist fears it is true.
>
> (J. Robert Oppenheimer)

TABLE OF CONTENTS

INTRODUCTION

The assignment of this thesis is to design and implement a distribution builder for the Debezium Server and describe the possible applications of such distribution in security systems. Debezium is the leading solution for change data capture (CDC) streaming, which allows changes to be captured on the database and sent into Kafka. Debezium Server is an alternative runtime for Debezium, which, among other features, provides the ability to send events to different systems like Apache Pulsar or Redis.

The main issue of the Debezium Server is that it is distributed as a complete distribution that includes all source and sink possibilities, which makes the distribution quite heavy with a relatively big memory footprint. This might be a problem for numerous applications that require archives to fit specific size restraints. Besides any restraints, it is simply an unnecessary burden for the runtime to contain everything.

This project aims to solve this problem. It should allow users to build a distribution that fits all their needs with a minimal memory footprint. Users should be provided with a Java library that allows them to create the distribution programmatically and a user interface that serves as a graphical representation of the library.

To provide a solid foundation for the implementation, the thesis begins by examining the fundamental technologies necessary to build such a complicated system. Specifically, the thesis explains the fundamentals of the Object-Oriented Paradigm, Java, Maven, User Interface technologies, and Cloud solutions. All of these together should make a solid foundation for the practical part of the thesis.

Furthermore, the thesis has to consider possible applications of the Debezium Server in security systems. These days, security experts have tons of data within the reach of a finger. However, they don't have the ability to use those data because they are located all over the Internet within different databases in various formats. The thesis has to analyze how to leverage the Debezium Server's abilities to gather data from various places and provide them with further computing that will add extra value for security experts.

In conclusion, the thesis will review all the obstacles discovered during the design and implementation process and provide a brief description of the possible future of this project.

I. THEORY

The main purpose of this thesis is to create a system for building custom distributions for Debezium Server. Debezium Server is a relatively new spin on the well-known Change Data Capture (CDC) technology Debezium. It allows continuous seeking of changes in the database and sending them to a remote messaging server.

The problem that comes with the Debezium Server is that it is currently distributed as a single archive that contains all database adapters and sinks (target message storage) adapters. This makes the distribution memory footprint extremely large, and it forces users to modify the build system themselves. This is overwhelming because the Debezium Server uses Maven as its build system, which uses several descriptor files that describe the distribution together. The result of this thesis should be a system that offers users a simplified way to assemble distributions specific to their needs.

Before the thesis can jump to design and implementation, it must go through the fundamental technologies. This Chapter chapter is divided into separate sections where each covers a specific technology for the implementation: Library (1), Debezium (2), User Interface (3), and Cloud environment (4).

1 Library

The library should allow users to create the Debezium Server distribution pragmatically. The distribution is, in this case, a Java archive for the Debezium Server that includes only a specific subset of the dependencies. To assemble such an archive, the library has to prepare a particular Maven descriptor (1.3) that contains only the desired dependencies (*source, sink connectors, and core library* (2.1) and their transitive dependencies.

Because the whole Debezium ecosystem is being developed strictly in Java, it is the only language that could be used to develop this library. Even if the situation were different, Java (1.2) would be one of the top choices for the project because of the paradigm. Besides the choice of the programming language, it is also important to decide how best to proceed with the *creation* process. Object Oriented Paradigm introduces many generally known *creational design patterns*:

Factory method introduces four participants– *Product, ConcreteProduct, Creator, ConcreteCreator.* The Creator can be implemented either as an interface or a class. The Creator declares the factory method that returns the Application interface. Besides the declaration, the Creator can still offer default implementation of the method. As the name suggests, ConcreteCreator extends/implements Creator and offers specific factory method implementation that creates ConcreteProduct [8].

Abstract factory is higher abstraction to previous *factory method.* Factory, designed in the 1990s, offers an interface for initializing specific families of objects [8]. A great example would be having a product like a table or chair. Given that abstract factory *FurnitureFactory* contains methods *createTable(), createChair().* Based on that, there could be concrete factories *SteelFurnitureFactory, WoodenFurnitureFactory.* These abstract factories would then override methods and return concrete furniture object *SteelTable, SteelChair, WoodenTable, WoodenChair.*

Builder is presumably the most popular among creational patterns. It offers a way to unify and simplify the creation of very complex objects [8]. Builders usually work on the principle that each method returns an object of the builder that already contains a modified complex object stored inside.

Once all things are complete, a method always returns a desired entity. The main advantage of the builder pattern is the possibility of nesting. If the desired entity is complicated, it is possible to nest multiple builder layers. If approached correctly, nested builders can simplify the code. On the other hand, if a programmer takes the wrong approach and uses a different naming method, code can very easily become overwhelming.

Prototype design pattern uses copying to create new objects. The original object serves as the *Prototype*. This pattern has advantages in certain languages where creating a new object can be complex and time-consuming. In Java, it is not used as commonly as in static languages such as *C++* [8].

Singleton controls the number of class instances. Specifically, it ensures that there will always be maximally one instance of the class. This can be very handy in any management process. For example, creating and deleting many resources in complicated test suites is necessary. So, there is an absolute need to have one object carrying all the mandatory resource-handling information.

Based on the description above, it can now be decided what pattern will best fit the library implementation. In this case, creating large, complicated objects is necessary because the configuration of *Debezium Server* 2.1 is quite complicated. Those objects will create a relatively big tree hierarchy.

Considering this fact, the Builder pattern is probably the best fit, as builders can be nested. Although the pattern describes the nesting, it can be overwhelming in code. Luckily, the technology growth didn't miss this part in Java, and there are projects like *Sundrio* 5 that help write such builders in a unified way.

1.1 Object Oriented Paradigm

As *Java* is the main programming language for this project, it is necessary to introduce the Object Oriented Paradigm (OOP) briefly. This paradigm brings a different look to the structures in code. It introduces new code entities called *Objects*. Objects group procedures and data should be strictly related to the specific entity. This entity can be a virtual representation of a real thing or something abstract. These objects carry the most responsibility in the OOP [2].

OOP is built on top of message dispatch and passing. You can envision all the objects as a very well-organized company. Each employee has their mailbox where he receives his job assignments. Once an employee gets a new email, he decides if this task is achievable by himself or needs somebody with higher privileges. If that is the case, he **passes** task to his supervisor. Employees can also find that for one specific sub-task, they need input from colleagues who work in other departments, so he **dispatch** email to that colleague with the details. The example above briefly describes how object cooperation works, but some other principles add primary importance and complexity to OOP. These principles are *Composition, Inheritance, Encapsulation, and Polymorphism.* These principles are crucial for code to make the most of object-oriented paradigms.

Composition is the primary concept of the paradigm. It allows objects to contain another object as their instance variable. This concept is quite important to follow the *SOLID* [1] principle. Specifically, the Single-Responsibility part of it which declares that each object should have only a specific subset of functionalities that all round up around a single area.

Inheritance allows programmers to create a new class (child class) based on an existing class (superclass). The child class inherits all methods and attributes from the superclass. This forces programmers to think about code reusability and creates hierarchical class relationships.

Inheritance is necessary for extendable software. Extension can be made just out of specific child classes without modifying superclasses. Besides that, inheritance also pushes programmers to extract duplicated common behavior out to the superclass [2]. Inheritance can be graphically represented using *UML*[2].

Figure 1.1 showcases the simple inheritance of various specific bank accounts. This UML representation may not be strictly correct based on spec because standard UML that includes inheritance does not include all parent class entries in the child class. Figure 1.1 displays inheritance with fictional classes to make the diagram easier to understand. Real use cases for class diagrams can be found in the design section 5.

Encapsulation is another key concept that is being used in object-oriented programming. It offers a way to hide an object's internal representation from the rest of the objects. The general idea behind this concept is to have attributes and certain methods visible only for specific objects (if any). These hidden attributes can then be accessed or modified via specific methods. These methods are, in most cases, *getters and setters.* Hiding internal state is often called *information hiding or data protection.* An example of encapsulation is displayed in Listing 1.

1) SOLID is an acronym for *Single-Responsibility, Open-closed, Liskov substitution, Interface segregation, and Dependency inversion principle* [21]

2) Unified Modeling Language (UML) is a specification that defines a unified graphical representation for various usages. Such as programming, data modeling, and others. The spec of current UML can be obtained from *OMG org* via `https://www.omg.org/spec/UML/2.5.1/About-UML`

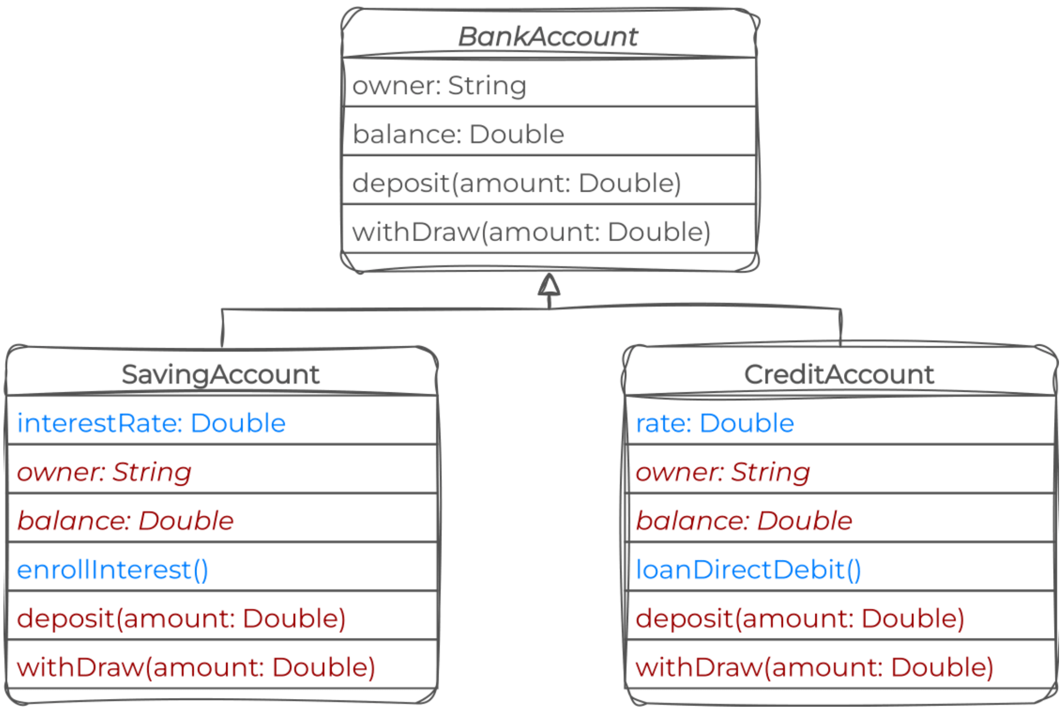


Fig. 1.1 Inheritance example displayed using UML.

Polymorphism is the hardest but probably the most important concept of all. Fundamental idea behind Polymorphism is an object that can take many forms. Specifically, it brings an ability that allows child classes to override the behavior of superclasses or interfaces without breaking any relationship created by Inheritance 1.1. Polymorphism is key for writing reusable and adaptable code. There are two types of polymorphism:

- **Compile-time** or, in another name, method overload. Method overload means a class can have multiple methods with the same name but different parameters. The method that will be executed is selected during compilation. This type of polymorphism is not strictly bound to OOP.

```java
public class Person {
    private String name;

    public String getName() {
        return name;
    }
    public void setName(String newName) {
        name = newName;
    }
}
```

Listing 1 Example of embracing Encapsulation principle in Java.

- **Run-time** polymorphism brings the override ability. Methods have the same signature, and execution is determined during run time. This concept is usually different for each language, but the overriding method needs to be annotated with specific keywords like `@Override`. In Listing 2, there is a depiction of Run-time polymorphism in Java.

```java
public interface Car {
    String getManufacturer();
}
public class Audi implements Car {
    @Override
    public String getManufacturer() {
        return "Audi";
    }
}
public class Ford implements Car {
    @Override
    public String getManufacturer() {
        return "Ford";
    }
}
```

Listing 2 Polymorphism in Java.

These principles are key to good software. There are many reasons, but the major reasons are the software's extensibility and maintenance. In the past, some projects were written once, and nobody touched them for ten or fifteen years. This is no longer possible, especially when discussing user-facing software.

1.2 Java

Java is a general-purpose, object-oriented programming language. The first release of Java took place in 1995. The current last release is version 21. The difference between the first and last versions is certainly not negligible. Although many new features are introduced in Java 21, Java 17 is a better choice for the purpose of this thesis as the whole Debezium ecosystem is currently based on this version. Java 21 and 17 are versions with Long Term Support (LTS), which assures that it won't be necessary to upgrade soon.

For this project, the main advantage of using Java 17 instead of, for example, old Java 11 is better *Null-Pointer* exception feedback. This exception can occur very often during serialization and parsing. Debugging it, on the other hand, takes time, and previous Java versions weren't exactly helpful during this process. From version 17, once an exception is thrown, the program will return the line on which the exception occurred and the name of the method and object that was *null* [11].

One question that may arise is if there is any other reason why to use Java instead of some other programming languages like C++ or Rust, rather than just sticking with the organization standard. The main benefit is the fact that programs run in a completely independent virtual machine – *Java Virtual Machine (JVM).*

Running in the *VM* brings many upsides and downsides. The most significant advantage is that code is, in theory, completely portable. It can run on any HW or operating system that has *Java Runtime Environment (JRE)* installed. But as Oscar Wilde said: "No good deed goes unpunished.". JVM is no different, and the downsides are not negligible.

There are two main disadvantages with JVM:

- **Performance** – JVM has low performance. This information is popularly used, but is it true? It depends on what language we are comparing. If we compare Java in default JVM with an application written in C++ and compiled for the specific architecture with exact CPU optimization, Java would be slower in most cases.

  When comparing Java to generically compiled C++ binary, Java might sometimes be slower and strictly depends on the problems with which performance is measured. That is because Java uses Just In Time (JIT) compiler, which can optimize to the platform where the program is executed. Although these optimizations may not be as strong as specific CPU optimization instructions, it happens on each platform that the program is started without the need for the particular binaries for each distinct platform.

- **Memory usage** – the biggest downside of the JVM. JRE takes more than a reasonable amount of memory. This led to many new initiatives like *GraalVM*[3] that can reduce memory footprint by almost 50%.

An essential part of this library is essentially dependency manipulation, so it is necessary to explain how dependencies in Java work. Dependencies are essentially external

[3]GraalVM is a new type of JDK developed by Oracle. More information about GraalVM can be obtained at `https://www.graalvm.org/latest/docs/`

libraries[4]. As mentioned, Java runs in JVM, meaning all necessary libraries must be accessible on the classpath or added as arguments during the program start.

With bigger projects, the amount of dependencies can be outstanding because writing things like logging, file manipulation, or any other common functionality from scratch does not make sense. This quantity of dependencies cannot be managed manually since it leads to mistakes. Gladly, some solutions make things much easier for programmers. Currently, there are two main competitors:

Gradle is a newer alternative to Maven. It allows building and managing dependencies for projects written in many languages, including C++, JavaScript, or Java. Gradle uses Groovy programming language for project description, making it more natural for some developers. The big advantage of Gradle is its performance. It can be over 70% faster than Maven in certain scenarios [10].

Maven has been and still is the most used build automation and dependency management tool. Even if Gradle is newer, there are still some reasons why Maven may still be a better choice. The first of them is simplicity. Maven uses *XML (Extensible Markdown Language)* format, so the whole description of the project is declarative by nature.

The second advantage of Maven is integration. As Maven has been in the community for over 20 years, almost every tool supports Maven as a build system. The last benefit of this build system is the significant number of build-lifecycle steps that allow for great versatility during the build.

Even if there are good reasons why to use Gradle, you will not find it in any projects included in this thesis. The reason for that is simple *Debezium Server* uses Maven as a build system, so the library will have to work with its XML declaration, so it doesn't make any sense from the community point of view to bring a new build system into this environment.

1.3 Maven

As Maven is this assignment's primary project management tool, it is necessary to go through all Maven fundamentals and specific parts that must be manipulated to create a functional custom distribution of the Debezium Server. We must understand Maven project descriptor `pom.xml`, some of the lifecycles, and *assembly descriptors*.

[4] Java code that serves a specific purpose and can be externally added to the current project

Without any objections, `pom.xml` file is the most important thing for any Maven project. This file (later only *POM (Parent Object Model)* contains information about the project and configuration for each life-cycle step. POM uses XML format, and it has rigorous syntax. Each POM's main part is the identification of the current project/module. This occurs in the `<project>` closure and contains artifact id, group id, and version. It may look like on the Listing 3. That description sets what the project's naming will look like once it is released. It would be released under `io.debezium` organization with `distribution-builder` name, under `0.0.1-SNAPSHOT` version. The

```
<project>
    <groupId>io.debezium</groupId>
    <artifactId>distribution-builder</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</project>
```

Listing 3 POM project description.

next section of the POM usually contains information about external dependencies or project modules. The dependencies section starts with `dependencies` closure. That closure contains a list of dependencies, where each dependency is described in a separate `dependency` closure. Such a list is showcased on the Listing 4. With such definitions of dependencies, Maven will download and include an external dependency on the Debezium Operator API.

The modules section allows programmers to divide projects into separate parts (modules). Each project module has its own POM with `<parent>` directive aiming to the relative path of parent POM. The parent is effectively another Maven project that can define things every day for all the modules, like dependencies, plugins, information about developers, and others.

```
    <dependency>
        <groupId>io.debezium</groupId>
        <artifactId>debezium-operator-api</artifactId>
        <version>2.6.0-SNAPSHOT</version>
    </dependency>
</dependencies>
```

Listing 4 POM dependency description.

Dependencies in parent must contain full definition (`artifactId`, `groupId`, `version`), but modules that can use just simplified one (`artifactId and groupId`) and inherit version from the parent project. This way, consistency in dependencies is guaranteed.

Parent directive creates a DLL (Double Linked List) between POM files. Illustration usage of this functionality is displayed on Listing 5. The last part of the POM file

```xml
<!--Parent POM-->
<project>
    <groupId>io.debezium</groupId>
    <artifactId>debezium-server-dist-builder-parent</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <!-- Project modules -->
    <modules>
        <module>dist-builder-library</module>
        <module>dist-builder-example</module>
    </modules>
</project>
<!--Module POM-->
<project>
    <artifactId>dist-builder-library</artifactId>
    <!-- Versions can be different -->
    <version>0.0.1-SNAPSHOT</version>
    <parent>
        <artifactId>debezium-server-dist-builder-parent</artifactId>
        <groupId>io.debezium</groupId>
        <version>0.0.1-SNAPSHOT</version>
        <!-- Relative path to parent POM file -->
        <relativePath>../pom.xml</relativePath>
    </parent>
</project>
```

Listing 5 POM modules example.

usually contains the configuration of project life cycles. These life cycles often differ based on the current or specifics the programmer wants to achieve. Profile definition introduces POM in POM schematics. And it is the profile that affects the result of Maven build the most. It starts with `profile` closure.

This closure contains profile ID, activation schema, dependency list, and build directives. The build directive is significant because it contains the configuration for the project build process. In this thesis, the only important part of this closure is the plugins section, specifically `Maven-assembly-plugin`. Listing 6 illustrates such a profile section. The plugin allows developers to combine project output into a single archive that can be distributed to any destination. The result of this plugin can be any archive like *.zip, .tar, tar.gz*, and others. This archive is called *Assembly*. Assembly contains all files that are necessary for project execution and are defined in the assembly descriptor. The descriptor is composed of specific filters. It is important to remember that the assembly plugin cannot add anything that has not been downloaded. There

```xml
<profiles>
    <profile>
        <id>assembly</id>
        <activation>
            <!-- If true profile will be started in each Maven package
                phase, otherwise only with -Passembly argument -->
            <activeByDefault>false</activeByDefault>
        </activation>
        <!-- Part of the profile is skipped -->
        <build>
        <plugins>
            <plugin>
                <groupId>org.apache.Maven.plugins</groupId>
                <artifactId>Maven-assembly-plugin</artifactId>
                <!-- This variable points to central properties
                   section -->
                <version>${version.assembly.plugin}</version>
                <executions>
                    <execution>
                        <id>default</id>
                        <phase>package</phase>
                        <goals>
                            <goal>single</goal>
                        </goals>
                        <configuration>
                        <descriptors>
                            <descriptor>descriptor.xml</descriptor>
                        </descriptors>
                        </configuration>
                    </execution>
<!-- Rest of the profile is skipped -->
```

Listing 6 POM assembly profile example.

are three groups of filters *FileSets, DependencySets, ModuleSets* [16]. Each of these groups targets a specific subset of entities. In those filters, developers can either explicitly include or exclude the entities. An obvious question might arise: "Why do not include all files, modules, and dependencies?". Not all those files, dependencies (including transitive dependencies), and modules are needed for specific assembly, and all of those unnecessary entities make the resulting archive bigger.

Some of these filters are pretty easy to understand. Those are *FileSets and ModuleSets*. These add or exclude files based on exact file/module names or based on wildcards. The dependency set is the most problematic because it is easy to exclude some dependency that is not the original intention. This often happens because developers exclude top-level dependencies, but that filter also excludes all the children of that dependency. With hierarchy from Listing 7 and dependency set filter from Listing 8, Maven assembly will exclude not only `org.external.dependency` but also `org.external.transitive-dependency` which will result in exception `No Class named XYZ found on classpath`. This Section went through all the principles and

```
io.example.parent
+-io.example.first-child
| +--org.external.dependency
| | +---org.external.transitive-dependency
+-io.example.second-child
  +--org.external.transitive-dependency
```

Listing 7 Example dependency hierarchy

```
<dependencySets>
  <dependencySet>
      <outputDirectory>${project.parent.artifactId}/lib</
         outputDirectory>
      <unpack>false</unpack>
      <scope>runtime</scope>
      <useProjectArtifact>false</useProjectArtifact>
      <useTransitiveFiltering>true</useTransitiveFiltering>
      <excludes>
        <exclude>org.external.dependency:*</exclude>
      </excludes>
  </dependencySet>
</dependencySets>
```

Listing 8 Example dependencySet part of description

fundamentals of Creational design patterns, OOP, Java, and Maven. These create an absolute foundation for the main library implementation for this thesis. The knowledge gathered through this section will be used to develop a Distribution Builder and debug and track issues that could occur in the Debezium Server or any other dependency along the way.

2 Debezium

Debezium is the leading solution for Change Data Capture (CDC) streaming. CDC is essentially a set of design patterns. These patterns declare how to determine and track database entries that changed their deltas. Delta indicates that database entry changed in any way (creation, deletion, update). These entries are gathered and delivered to the next system. There are many ways how to track these deltas. Below, you can find some examples:

- **Additional timestamp on rows** – this timestamp would change accordingly after each change of the entry. In this way, it is very tricky to track deleted events.
- **Specific triggers on tables (if there is support for that in the database)** – trigger is usually an SQL statement that is executed once a specific event happens on the target table. In this case, the trigger could, for example, append flags to the *status* table after each specific change on the target table. This way, developers can effectively create a very simplified and less accurate version of the transaction log.
- **Status flag on rows (custom specific column)** – similar to additional timestamp. Each row gets one additional flag column, which can then be used to identify what type of event (if any) happened on that row. Status flags are best in combination with the timestamp approach.

Even if these simple solutions may sound promising, mostly they are not. With very heavy traffic, these methods start to lose some of the changes, and things get messy. A much better solution is living directly in the heart of the database – the Transaction log. The transaction log stores all the transactions that happen in the database. Log entry format can differ from database to database, but at least in all traditional databases, log is there. This makes things really interesting because it seems like any system based on these transaction logs cannot miss anything. Yes, that would be an ideal world, but as you might already think, there is a catch.

The transaction log usually behaves like a circular tape, so at some point, it will start rewriting the earliest changes. The size of this log is mainly defined by the user, so any CDC solution has to play around with this problem and handle it. Gladly, some solutions allow CDC applications to recover from this situation and mitigate the risks.

A possible mitigation for such a problem is a snapshot. The snapshot is a process during which Debezium scans the whole database or the subset of it and sends this whole state into Kafka. This process can mitigate the problem mentioned earlier very much. Debezium snapshot process can be configured to mode `when_needed` that will trigger a snapshot whenever the committed offset was not defined before in the transaction log.

Besides, this snapshot can also be used in much more straightforward situations, such as database migration. In this situation, Debezium snapshot mode is set to `initial`. Once the connector starts, it scans the whole database and sends all the records to Kafka. Then, another connector, like the JDBC sink connector, can take those events and insert them back into another SQL database.

Some database manufacturers offer CDC solutions on their own. An example of this approach would be Oracle, which provides this solution under a product named *Golden Gate*. This solution is strictly for the Oracle database. It streams all the changes in an Oracle-specific format to one of the supported storages. This solution might be sufficient for some users, but the major downside is that it is expensive and may not apply to smaller companies. The second downside of a proprietary CDC solution is that it is always bound to only one database. This might be a problem for some companies because more prominent companies usually use a more diversified technology stack.

Debezium is a trendy project not only because of the issues mentioned above. The significant advantage of Debezium is the considerable support matrix. There are seven database connectors (connector term will be described later in this section) directly supported by Debezium maintainers and a couple of others in the incubating process[1]. Some other database connectors are actively developed by the community or are led by different companies like Google[2].

So far, this Section has gone through the basics of the CDC. Now, it is time to look deeply at Debezium. The CDC part of this project is based on the transaction log approach. The problem is that this approach is different for each type of database because each transaction log is entirely different, and more importantly, each database works with this log differently. Because of this, Debezium is divided into separate connectors, one per database. However, this is still only about the CDC. What about streaming? Apache Kafka and Kafka Connect ensure the streaming part of Debezium.

[1] Incubating process in Debezium indicates that connector contains main functionality but it is not yet thoroughly tested and proofed by community.

[2] Google is actively developing Spanner connector.

Apache Kafka is an open-source platform for event processing. Event in this context can be any message in Kafka event format [20]. The very simplified architecture of Kafka is displayed in Figure 2.1

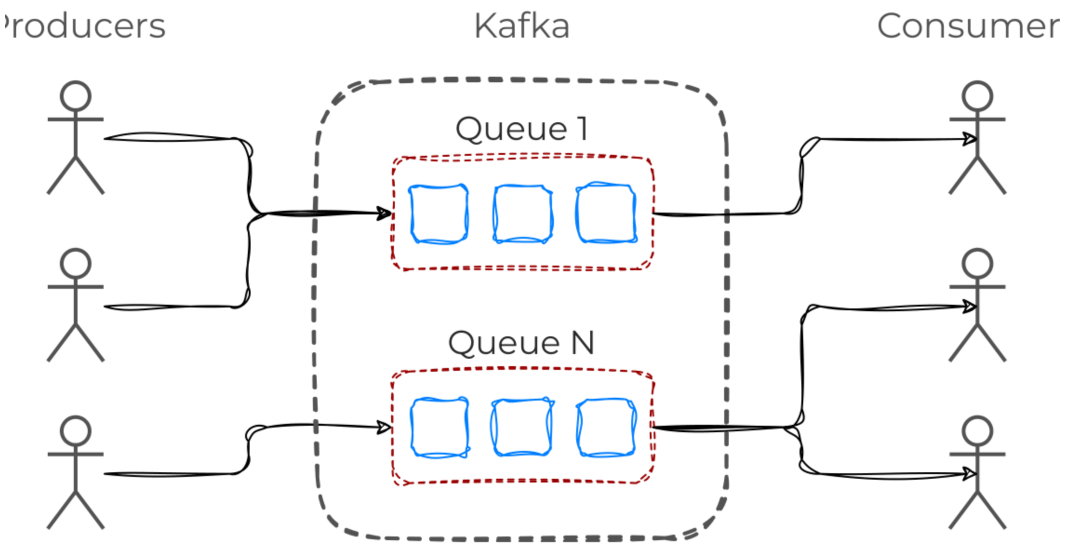


Fig. 2.1 Simplified Kafka architecture

As Figure 2.1 shows, each Kafka broker is composed of event queues. Each of these queues should contain messages about the same subject. These queues are named Kafka topics. Both producers are connected to a specific topic. Consumers can form groups. Kafka stores offset for each consumer group, so it assures (for the single group) at least once delivery, which is very important for Debezium. Besides that, Kafka offers many other exciting technology solutions like scaling, partitioning, or fault-tolerance, which are unnecessary for this thesis.

Kafka Connect is a component of Apache Kafka that provides seamless integration between third-party systems and Kafka brokers. Kafka Connect (KC) is a distributed process that can run across multiple machines and is comprised of a series of connectors. The main process of KC handles all the communication with the Kafka Broker. KC pulls the changes from its internal queue that is being filled by all connectors. There are essentially two types of connectors:

- **Source** – Fills Kafka with data gathered from some source system
- **Sink** – Allows exporting data from Kafka to another system

Manipulation with the connectors inside the Connect process can be achieved via REST API. This API allows all of the CRUD operations to be performed on the connectors. Configuration of the connectors is stored in the JSON format. This format is displayed in the Listing 9.

```
{
    "connector.class": "io.debezium.connector.mysql.MySqlConnector",
    "value.converter": "io.confluent.connect.json.JsonSchemaConverter"
        ,
}
```

Listing 9 Properties format example

Besides capturing changes from the database and delivering them to Kafka, Debezium also offers several implementations of Kafka Connect transformation that were demanded mainly by the community. These transformations are called Single Message Transformations (SMT). As the name suggests, transformation is always applied to a single message. The list below contains examples of some useful transformations:

- **Content-Based Routing** – transformation looks into the record, checks the record's content, and selects the best topic based on the content.
- **Message Filtering** – based on the content of the record transformation class, select if the record suits the filter. If yes, the record is sent to Kafka.
- **Timezone Converter** – one of the most popular transformations offered by Debezium. This SMT converts the value of the target field to the desired time zone.

Besides the prefabricated transformations, Kafka Connect also supports creating custom transformations. Users implement a specific Interface that is provided. Build custom transformation into `jar` and put this archive next to the Debezium connector in Kafka Connect plugin dir.

As mentioned earlier, Debezium offers a bundle of Kafka Connect connectors to users. These connectors are distributed as assembly 1.3 archives that contain all necessary libraries and Java executables. Those archives are then added to the KC *plugin path*[3]. In the case of Kubernetes deployment, users have to add those assemblies to the specific path and then build *Docker Image* 4.1. This image is later pushed to the reachable image registry, and its reference is then used to deploy the KC container into Kubernetes.

Although the Kafka and KC combination is a powerful technology stack, Kafka's technology specifics might not suit everyone. Debezium maintainers have often seen

[3]Plugin path is a directory which contains all Kafka Connect plugins, stored as Java Archives.

questions about supporting different sink types like Apache Pulsar, Google PubSub, etc. These questions have led Debezium developers to development of the Debezium Server 2.1.

2.1 Debezium Server

Debezium Server is a standalone application that allows running Debezium source (database) connectors and sink modules. This application is built on top of *Quarkus*[4) framework. As Section 2 noted, the main runtime for connectors is Kafka Connect. As there were many requests from the community to allow Debezium to run on top of a runtime different from Kafka Connect, Debezium developers came up with the Debezium Engine.

Debezium Engine mimics the behavior of Kafka Connect. The engine uses the same API as Kafka Connect and provides the same services to connectors as Kafka, including offset storage. Offset storage stores the index of the last message received, so Debezium can offer at least one delivery.

Until recently, the Debezium Server based on the old engine didn't provide any improvements compared to what KC provides. But release *2.6.0.Alpha2* brought major improvements that promoted the server to another level.

This release brings parallelization to the playground. So far, everything in Debezium has worked as a single thread. Everything is in the same thread from the source through processing into the sink, as Figure 2.2 showcases. The only parallelization

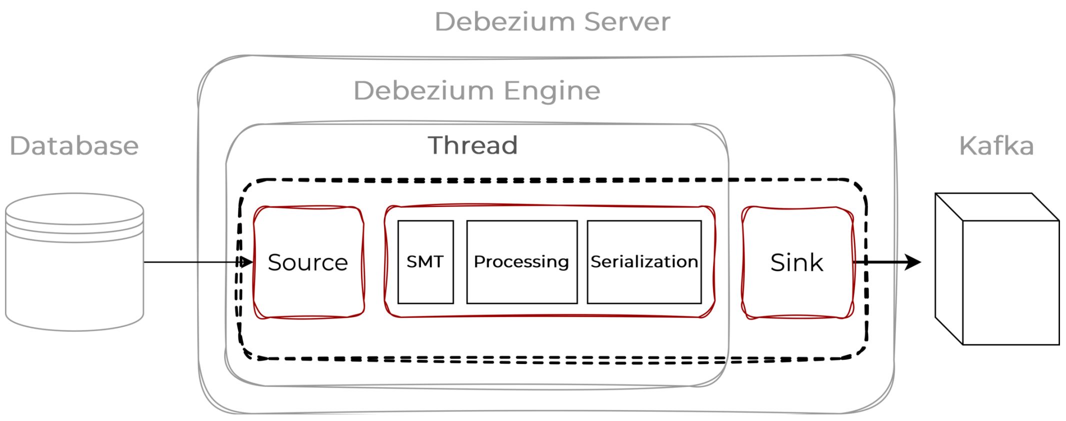


Fig. 2.2 Diagram of Debezium Server and Debezium Engine composition, including thread schematics.

[4)] Quarkus is a Java framework tailored explicitly for Kubernetes. More information about Quarkus can be obtained at `https://quarkus.io` offered by Kafka Connect is to run source connectors in multiple KC worker processes. The Old Debezium Engine cannot provide even this and always runs only one task in a single process. However, this feature is only enabled for specific connectors that can ensure all functionality. These connectors are currently MongoDB and Microsoft SQL Server. The problem is that the processing part is the most significant performance bottleneck of the whole pipeline. This processing includes operations that cost the most CPU time, like serialization and SMTs (Single Message Transformations). The new version of the Debezium Engine gladly solves this issue. Figure 2.3 displays how the new version changes the thread schema.

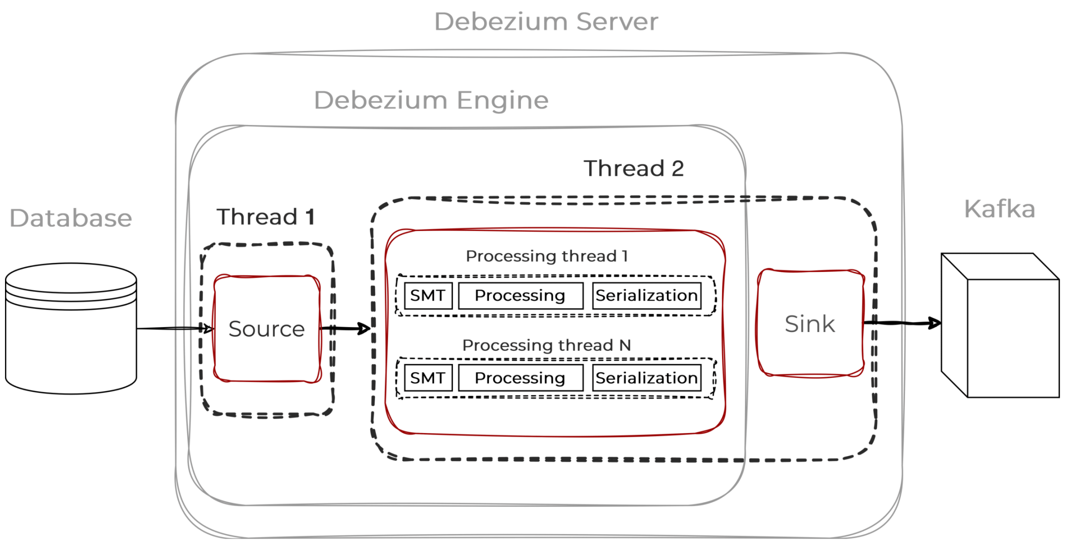


Fig. 2.3 Diagram of Debezium Server and the new version of Debezium Engine, featuring multi-thread processing.

As Figure 2.3 shows, the engine now separates the pipeline into two main threads. The first thread is strictly for source connectors. However, the possibility of running specific connectors (noted above) in multiple threads is still maintained. The second thread is reserved for managing threads running records processing and sink. The processing threads are, in reality, called from the *Thread 2.* And this is precisely the part where the new engine comes into play. This thread workflow is essentially separated into two parts. The first part is all about already noted processing.

As soon as the source connector exceeds the threshold of internal queue processing, the sink thread takes up all the messages and distributes them across the processing threads. These threads do all the work in parallel. Once all the serialization and SMT processing are done, everything is then handed over to the sink. There is also the possibility to avoid waiting for all processing to be done, but if users allow this option, ordering cannot be assured. Once messages are handed to the parent process, the sink sends them to the selected destination system.

The user can provide the sink implementation or use some of the sinks that Debezium maintainers provide. Sinks provided by Debezium cover the most used and community-promoted systems. Some of the connectors are listed below.

Apache Pulsar is a hybrid system that combines traditional messaging brokers like RabbitMQ and more modern systems that use publish-subscribe a model like Kafka. Pulsar separates compute and storage parts to different compute tenants, making it ideal for cloud environments like Kubernetes 4.2. One of the main features of Apache Pulsar is a built-in schema registry, geo-replication, and IO connectors.

RabbitMQ , open-source message broker that was originally based on AMQP protocol. Later, RabbitMQ was extended with MQTT, STOMP, and HTTP gateways during the later development phases. It supports both failover and clustering mechanisms.

Redis is arguably the most popular sink in the Debezium Server community. This sink system is an in-memory key-value database. Key-value databases are prevalent for feeding Machine Learning (ML) models. ML can be very useful even in Security systems. Many modern security systems use machine learning for face recognition, behavioral detection, and other use cases. Redis is such a popular solution because it is effortless to use and can store a wide variety of data.

This thesis already mentioned all the new features that were included in the last release of Debezium Server. But there is still one surprise in the box. Debezium Server now offers easy deployment to Kubernetes (4.2) via Debezium Operator. Operator is a designation for service applications that manage other application deployments. In this case, managing the operator scales the application, handles configuration reload, and other necessary operations to keep the application running.

The operator brings significant security advantages, as it allows deploying Debezium Server to Kubernetes, where users can leverage all Kubernetes security features, like secrets. These secrets behave like encrypted storage, where users can store certificates and other sensitive information. These secrets can then pass credentials to the operator, who can deploy the Server with already decrypted and correctly inserted sensitive configuration fields.

So far, this part of the Section has gone through how the Debezium Server and Debezium Engine work, where these technologies can be leveraged, and why they are popular. For more clarification, the list below summarizes the biggest advantages of the Debezium Server over Debezium on KC.

- **Kafka independent** – Debezium Server is not bound to the Kafka messaging system. This means the Debezium Server can produce change records for various messaging systems like Redis, RabbitMQ, etc.
- **Expandability** – This twist on Debezium offers users an easy way how to extend the Debezium Server by any custom Sink connector. These custom implementations can implement Server API and then be injected into distribution libraries.
- **Performance** – Debezium Server offers parallelization for the processing pipeline. This brings big performance gains for the users as serialization still is and always was the biggest performance bottleneck in almost every application.

We have worked our way through the leading technologies in CDC, Debezium, and Debezium Server. These are two pivotal components in CDC and event-streaming. Before ending this Section, it would be best to synthesize all the findings and underscore the key takeaways.

Debezium is the leading solution for CDC Streaming. CDC (Change Data Capture) is a pattern that provides possible ways how to capture changes from the database. Debezium leverages one of these patterns, specifically the one based on the transaction log.

Running in Kafka Connect, Debezium can ensure that all changes will be captured and streamed to Kafka with very low delays. The biggest advantage is that Debezium supports many various databases so that it can be plugged into any technology stack with little effort.

Debezium Server, new runtime for Debezium connectors. This runtime brings many new possibilities because it supports various messaging systems as destinations. Besides variability, the Debezium Server also significantly improves performance, offering even lower delays than Debezium on KC, even if it is running specific SMTs.

Machine Learning is not a strict Debezium or Debezium Server feature. With the new streaming options of the Debezium Server, there come new possibilities for using those Change Stream events. One of those possibilities is Machine Learning.

Some of the messaging and event solutions that Debezium Server supports are ideal for feeding Machine Learning, which can be very useful in modern security solutions, as will be later described in Section 7.

3 User Interface

So far, this thesis has been based on object-oriented programming, Java, and Debezium. These are all very important information, but just the Java library might not be enough for most of the community. Because of that, we have to look into web technologies. This section covers both *backend* and *frontend* technologies that could possibly be used for the implementation.

The first part of the Section goes through the *backend* technologies. Specifically, it wraps up the bases of the *Java* frameworks – *Micronaut* and *Spring Boot*. The second part covers information about *JavaScript* frameworks – *React* and *Angular*. In the end, the Section summarizes all gathered information and forms the final selection of the technologies used in the implementation.

3.1 Backend

Firstly, it is wise to define what a backend is. In software engineering, the web environment is logically separated into independent pieces, and each of these pieces covers different functionality. The backend is one of those pieces that should cover all data manipulation. The backend should not be directly accessed by the users. It should be accessible only through the front end. Backends are, in most cases, realized via *REST API* [5], which essentially accepts data, processes them in some way, and then returns the data to the front end.

Micronaut

It is one of the new Java frameworks that is currently rising. The whole framework is based on modularity, lightness, and speed. Micronaut is more or less aiming to allow developers to build *microservices* easily. Microservice architecture is an architectural style that structures an application as a collection of small, independent services [14]. This architecture allows companies to delegate work to multiple teams, where each team develops a specific service [18].

As microservices are the main aim of the Micronaut, it needs to work around the performance because the microservices are known to be fast and lightweight. This framework delivers this at a very high standard. The performance is assured with a very specific approach – as soon as the application is compiled, it processes all metadata and generates a set of classes that represent an already configured application (this process is named Ahead of Time compilation (AoT)).

Besides the performance, Micronaut offers a wide variety of specific features [17]:

Cloud-native support is arguably the main reason why a lot of users decide to give Micronaut a chance. Cloud-native means that the framework provides tools that can be used to develop applications that run in the cloud environment. These tools might be service discovery, distributed configuration, and distributed tracing. All these are necessary to create modern cloud-native applications [3].

Reactive programming is definitely one of the main points of interest in web development. Reactive programming allows the development of non-blocking servers that can handle a large number of concurrent connections with a relatively small amount of threads occupied. Micronaut provides support for both *RXJava and Project Reactor*1)

Data access with AoT significantly reduces runtime overhead during database access. This is assured because Micronaut precomputes all queries for both *JPA and SQL modeled repositories*2). Besides that, it also provides native support for MongoDB.

Overall, Micronaut is a solid choice for the development of applications that are mainly aimed to run in the cloud environment. It has many beneficial attributes, the most valuable of which is performance. Besides just the performance benefits, Micronaut also significantly reduces the memory footprint of applications, which is, as already mentioned in previous Section 1.2, probably the most significant disadvantage of Java.

Spring

Spring is an alternative to the original enterprise Java (Java EE), nowadays more known as *Jakarta Enterprise Edition*3). Spring build on top of the *Model View Controller*

1)These libraries bring types like Observable or CompletableFuture. More information about those libraries can be obtained from `https://github.com/ReactiveX/RxJava` and `https://projectreactor.io/`.

2)Java Persistent API (JPA) is an abstraction for a relational database that allows database management via code, more information available from `https://www.oracle.com/java/technologies/persistence-jsp.html`.

3)Java Enterprise Edition (JEE) is an enterprise software platform that is mostly used to build web systems. More information is available from: `https://www.oracle.com/cz/java/technologies/java-ee-glance.html`

(MVC) architecture. MVC essentially divides the application into three logical units:

- **Model** – part of the MVC responsible for handling all the logic for the application. Logic, in this case, handles all the computation, database queries, etc. The model is completely separated from the users. It does not know anything about the input source or what the response of the application looks like.
- **View** – only unit visible to the user. It is responsible for presenting data to the users and allowing them to interact with the system. The view is usually but not exclusively via programmable HTML templates.
- **Controller** – serves the role of the middle man between the Model and View. The controller takes data served from View, parses them into the Objects, and hands them over to the Model. Once the Model finishes all the processing, it returns data to the Controller, which transforms them into the correct form and passes them to the view.

Spring provides many features for developers that ease their lives during development. Many of these features are common in frameworks like Spring, and some are distinctive to this framework. Spring framework is separated into specific modules, where each module serves a specific feature. This part of the section goes through the most exciting modules [24].

Spring Data Access is an integral facet of the Spring Framework and bridges the gap between applications and underlying data storage. It encompasses transaction management, JDBC-based approaches, and more contemporary paradigms like Reactive Relational Database Connectivity (R2DBC). Additionally, Spring supports *Object-Relational Mapping (ORM)*, marshaling data formats, and seamless integration with diverse technologies. ORM is a technique that allows developers to manipulate databases using an Object-Oriented Paradigm. Its significance lies in ensuring efficient and reliable data interaction for modern software systems. Spring Data Access provides this functionality to the most used persistence– JDBC, JPA, and MongoDB.

Spring Boot arguably the most important of the Spring extension. Spring Boot is a practical solution to the challenges involved in setting up Spring applications. It simplifies development by eliminating the need for complex configurations. Instead, it follows a "convention over configuration" approach, automatically managing dependencies, embedded servers, and application startup processes. Spring Boot brings many advantages but also disadvantages:

- **Productivity Boost** – Spring Boot's opinionated defaults and autoconfiguration accelerate development, reducing cognitive load.
- **Embedded Servers** – The inclusion of embedded servers (e.g., Tomcat, Jetty) simplifies deployment, eliminating external server setup
- **Standalone Executables** – Spring Boot generates standalone JAR files, enabling straightforward execution without external dependencies.
- Health Monitoring – Built-in health indicators facilitate application monitoring and management.

Advantages

- **Easy to use** – Once the developer gets familiar with Spring Boot, it provides a lot of techniques that can speed up the development process, like specific annotations.
- **Embedded servers** – Unlike the core Spring, Spring Boot provides already built-in *servlet*[4] container (by default Tomcat). Spring core does not include this and expects that all the configuration and preparation of the servlet container will be done prior by the developer.
- **Built-in health minitoring** – Spring Boot by default includes some basic *JMX metrics*[5]. These metrics often include metrics like the number of failed and successful requests.

Disadvantages

- **High abstraction** – Spring Boot abstracts a lot of the functionality, and that might be a problem for some developers because, at some point, it might be hard to debug the issue if the developer does not have a clue about internal implementation.
- **Autoconfiguration trade-off** – As the framework configures the application based on the dependencies, it might clash with some specific requirements.
- **Overwhelming learning curve** – There are a lot of features and techniques bundled directly in the Spring Boot alone. This might make things hard for the developer during the first project based on Spring Boot.

4) Java application that runs within the web server like Tomcat, Netty, etc.

5) Java Management Extension (JMX), standard mechanism for monitoring Java applications. More information is available from: `https://docs.spring.io/spring-boot/docs/2.1.1.RELEASE/reference/html/production-ready-jmx.html`

Spring Security encompasses all layers of security within an application, making it a comprehensive solution. Notably, it seamlessly integrates with various authentication models, including database-based authentication, OpenID connect, form-based authentication, X.509 certificate authentication, and Lightweight Directory Access Protocol (LDAP). Its adaptability ensures that security provisions evolve alongside changing requirements, making it a preferred choice for securing Spring-based applications. Spring Security is not included in Spring Boot, but it is added for developers who want to embrace the security.

Both Spring and Micronaut are solid choices for building microservices or web applications. It is very hard to say which of these frameworks is a better choice. It very well depends on the project's requirements. If the developer seeks performance, Micronaut is the better choice. On the other hand, Spring / Spring Boot offers a wide variety of specific modules that could help the developer achieve specific functionality. In other words, the decision between these two frameworks lies only on the developer because they provide almost the same features. This project will be implemented using Spring Boot because the author has experience with this framework from past projects.

3.2 Frontend

These days, what users see may be more important than how the application actually works. This might be sad, but it is a cruel reality. That means the choice of front-end technology is more important than ever. Users want a nice-looking, responsive User Interface (UI). Because of this, *JavaScript (JS)* is definitely the right tool for the job.

JavaScript is a high-level, interpreted programming language that is primarily used for building web-based applications. It is a multi-paradigm language that supports event-driven, functional, and imperative programming styles. It allows developers to implement complex features on web pages, such as interactive forms, games, scrolling abilities, and page navigation. Based on the name, you might think that there is some connection between Java and JavaScript based on the name. There is no connection between those languages, and they can't be more different.

Although JavaScript offers way how to implement all the components that should be necessary to build UI for the Distribution Builder, it would be wise to explore all frameworks that could be of use. There are many frameworks that are available but two of them definitely stand out – *React and Angular*.

React

React is an open-source library/framework primarily for building single-page web user interfaces[6]. Both React and Angular are handling the View layer (described with MVC 3.1). It allows design views for each specific state of the application and efficiently updates affected components when data changes. All the React views are described in a declarative way. This way, React provides a great way to debug those views once an issue occurs.

React features component-based architecture. This architecture allows developers to build encapsulated components that only have their own state and then compose all of those components to complex user interfaces. Each component has its own lifecycle, which the developer can monitor and manipulate during three lifecycle phases:

- **Mounting** – Steps during which element is put into virtual *DOM*[7].
- **Updating** – Changing or adding new data to the component, then rendering changes.
- **Unmounting** – Removal element from DOM.

The inclusion of virtual DOM significantly improves performance compared to using regular DOM. React includes an extension to JavaScript, named *JSX (JavaScript Syntax Extention)*. JSX is a combination of HTML and JavaScript that allows JavaScript to be embedded directly into the HTML components, which makes code much simpler and more readable. Although React is open-source, it is mainly maintained by Facebook or, nowadays, Meta [23].

React is a very powerful framework, but it has some disadvantages. The framework's documentation is very poor. The second disadvantage is that React does not provide any management component that would help keep the whole component tree in good shape. This might be especially problematic for large-scale applications, but it can be mitigated by additional tooling.

[6] Single-page website dynamically displays all information on the single page. More information is available at `https://www.bloomreach.com/en/blog/2018/what-is-a-single-page-application`

[7] Virtual Document Object Model, is a virtual description of the HTML document.

Angular

Angular is an open-source web application framework mainly developed by Google. Like React, it is mainly used to create modern single-page web applications. Angular is not written in JavaScript but in TypeScript.

TypeScript is an open-source programming language. Initially developed by Microsoft. It is essentially a superset of JavaScript, which means that it contains all the functions that JavaScript provides and adds some more functionality. The first difference is that TypeScript introduces static typing. This was introduced mainly to increase runtime performance and also mitigate risks of runtime exceptions. An even bigger change from JS is that TypeScript is Object-Oriented, whereas JS is Prototype-based [27].

This framework does not include HTML in the code. Because of that, Angular provides specific HTML templates that can, under specific circumstances, include TypeScript code. This framework also supplies a wide variety of libraries for almost every purpose, such as routing and forms. These libraries are especially useful, and that might be the very reason why Angular is popular. Especially the Reactive Forms library is arguably what makes Angular stand out from other frameworks.

This library allows developers to create forms dynamically in TypeScript code. Developers define metadata objects for each form input. These metadata objects can be either `FormGroup, FormArray, or FormControl`. Each of these objects serves a different purpose:

- `FormGroup` – Folds form objects into one bigger. This metadata object essentially represents `<form>` tag in HTML. Every entry in this group must have a unique identification. Thanks to this directive, it is possible to nest the forms with one output.
- `FormArray` – An alternative to the `FormGroup`. The main difference is that entries in the array do not need a unique ID. It can be imagined as a simple object array.
- `FormControl` – This object represents actual input field. It is composed of name and value.

These components shape a double-link tree. That way, Angular assures that a change of any element results in a change of the whole form. Angular also allows developers to set

`FormValidator` for inputs. Validators contain specific rules that the input field must comply with. However, the metadata object still must have a valid HTML template that will exactly match the metadata [19].

Angular is an ideal choice for projects that aim to provide dynamic and responsive UI. With the OOP, it can ensure much better code maintainability and readability. Besides that, Angular has great documentation, which includes all possible examples and provides excellent descriptions for all classes and functions.

This part of the text explored the technologies that are utilized in the creation of contemporary and responsive web applications. The backend portion of the thesis examined two frameworks, *Micronaut and Spring*. As stated in the backend conclusion, these two frameworks provide nearly identical features; hence, the choice of framework largely depends on the developer's preference. This project will be implemented using *Spring Boot*. The reason is that the author has already worked with it before, so the learning curve will be much faster, if any.

In terms of UI, this section has evaluated two potential technologies – React and Angular. The contrast between these two is significantly larger than that between backend frameworks. React is JavaScript-based, while Angular utilizes TypeScript. TypeScript is a definite plus for Angular as it facilitates Object-Oriented Programming. Another major benefit of Angular is its ability to construct reactive forms dynamically. This is a substantial advantage, as the majority of the Distribution Builder UI revolves around user configuration, which implies forms. This capability would enable the UI to operate independently from the backend. For these reasons, Angular will be the primary technology used in the UI.

4 Cloud environment

In this Section, we will go through the fascinating world of the cloud environment. Although the cloud includes many interesting technologies, this Section will cover only some of them – *Docker, Kubernetes, and GitOps.* Docker provides the ability to build and run applications loosely isolated from the infrastructure by using containerization.

Next, we will explore Kubernetes. Kubernetes bundles the functionality of the Docker and creates a vast orchestration system for containerized applications. Kubernetes allows developers to deploy, scale, and manage applications. Finally, this section will cover *GitOps.* GitOps is effective automation that allows keeping applications in the cloud up to date with the code stored in the *Version Control System (VCS)*[1] By the end of this Section, you will have a solid understanding of these technologies and why it is beneficial to build your application with them.

4.1 Docker

Docker allows the running, building, and shipping of containerized applications. In order to explain all the docker features, it is necessary to define what is *container* [15].

Container is essentially an alternative to Virtual Machines, although it works on a completely different basis. The container packs up the code and dependencies and creates an independent unit. Containers create abstraction at the application layer and share the Operating System Kernel, whereas VMs are an abstraction on the physical layer. The architectural difference between VMs and Containers is showcased in the Figure below 4.1. Containers, like VMs, are started from images. A container image is essentially an executable pack of software that includes everything necessary to start the application. Images can be reused, and developers can use the same image for multiple containers.

Docker builds the whole platform around the container architecture. It uses a client-server architecture. The server part is the container daemon (`containerd`), and the client is essentially the tooling that allows creating and managing containers running on top of the daemon [26].

[1]VCS is a software tool that allows versioning software sources. The most popular VCS tool is *GIT*.

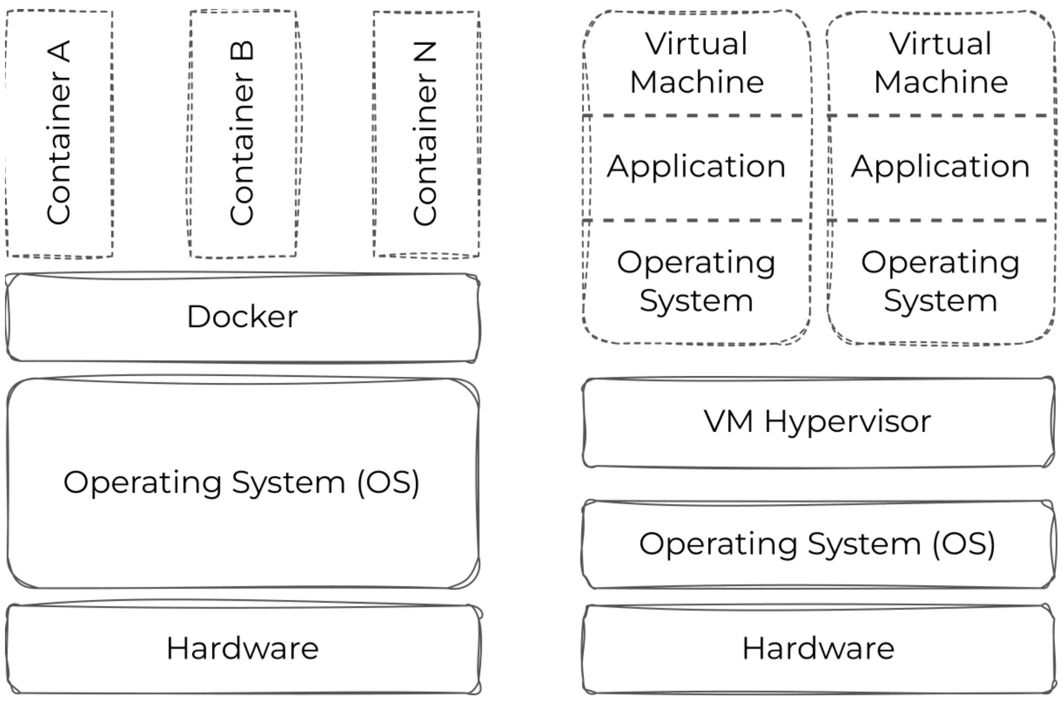


Fig. 4.1 Comparsion of Docker containers and Virtual Machine architecture.

So far, this text has only touched the container images. However, the images also take place in Docker architecture because we need a place to store them. Docker provides two ways how to store images. First is that developers can build the images and store them locally. This way, the images are only accessible on the current machine. The second method is to store images in external image registries. Image Registry is more or less the VCS for Images. Developers can push images into those registries under certain versions. This way, images can be accessible from every machine and can reach the external registry.

The container images are built based on their description. The description is named `Dockerfile`. This file is a text document that contains all the instructions that the developer would run in the VM to download and install all necessary software dependencies. Docker also allows for the extension of already existing images. Possible `Dockerfile` is displayed on Listing below 10.

The Listing 10 describes `Dockerfile` for the Apache HTTP server. `FROM` instructs the Docker to base this image on the existing *ubuntu* image. `COPY`, as the name suggests, copies the directory from which the build was initiated to the image. The next instruction runs the `apt` command. This command is the default package maintainer for Ubuntu. Firstly, it updates all the packages and gathers all the metadata from default repositories. Then, it installs two packages *apache2 and apache2-utils*. The `EXPOSE` instruction configures the image to open port 80 to external communication. This only

```
FROM ubuntu

COPY . .
RUN apt update
RUN apt install -y apache2 apache2-utils
EXPOSE 80

ENTRYPOINT["/bin/bash"]
CMD["apache2ctl", "-D", "FOREGROUND"]
```

Listing 10 Example Dockerfile for HTTP server.

configures the image if a developer wants to open communication with the container based on that image; the port must still be configured in the daemon. The last two instructions are `ENTRYPOINT` and `CMD`.

These two instructions are especially important because they configure the start-up command. Some developers often misinterpret these commands. The `ENTRYPOINT` sets the command that will be executed on container start-up. On the other hand, the `CMD` instruction sets the arguments for the previous instruction.

This Section went through the core principles of the containers and Docker. Here, you can find a short summary of this Section. Containers are a lightweight alternative to virtual images, which allows developers a relatively fast and reliable way to move code from development to production. Docker embraces the container architecture and creates a whole platform around it. It includes container image storage, daemon, and client. A combination of these creates an efficient and widely used container platform.

4.2 Kubernetes

An open-source container orchestration system. It is used to automate scaling, deployment, and management of containerized applications. Kubernetes was initially developed by Google, but in 2016, all development was moved under the *CNFC (Cloud Native Computing Foundation* [13]. Kubernetes is working on a higher abstraction level than Docker. It groups containers into logical units and distributes them across *Kubernetes cluster.*

Kubernetes cluster is the base of Kubernetes deployment. These deployment groups are hosts running Linux OS. These hosts can run either as *master* or *worker* nodes. Master nodes form the cluster *Control Plane.* The control plane takes care of all important decisions about the cluster, for example, scheduling of application *Pods* or handling API requests from the users. On the other hand, worker nodes take care of all the actual computation, so often, these clusters have more workers than masters [22].

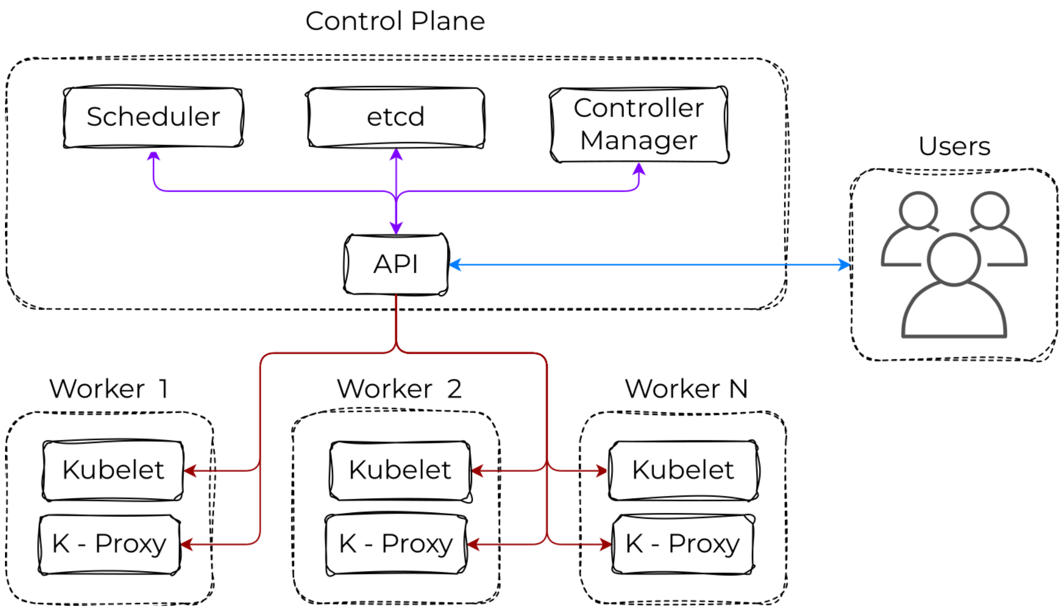


Fig. 4.2 Kubernetes cluster architecture.

On the architecture Figure 4.2, you can see a deeper dive into Kubernetes cluster architecture. You can see that there are a couple of unknown terms, like *etcd, Kubelet, Scheduler, and others.* Down below, you can find a brief overview of all of them.

- **etcd** – Key-value storage that offers high availability. Kubernetes uses *etcd* to store all cluster data. This storage is critical. If the synchronization of the *etcd* between master nodes fails, the whole cluster might become unavailable.
- **Kubelet** – Agent which runs in each Node. Kubelet ensures that all containers in the *Pod* are running healthy.
- **Scheduler** – Component that watches the cluster and waits for newly created Pods with no worker specified. For those Pods, the Scheduler selects an available worker and assigns this to the Pod. The scheduler selects workers based on multiple factors like HW, SW constraints, data locality, etc.
- **Controller Manager** – Spins up specific controllers for Kubernetes object, like Node or *Service Accounts*[2)].

[2)]Kubernetes non-human account that provide distinct identification. Service Account can be assigned to other specific Kubernetes Objects.

- **K-Proxy** – Network proxy which maintains network communication from and to the worker node.
- **API** – API server exposes Kubernetes API to users. The Kube API server is designed to be horizontally scaled. This can be achieved by deploying another instance of the server and then balancing the load between the instances.

Kubernetes architecture allows effective scaling of the whole platform and all applications that run on top of it. Kubernetes deployment of the application is described in multiple Kubernetes objects. Fundamentals Kube objects are – *Pod, Service, Deployment, Replica Set, and Persistent Volume.* These are fundamental objects that allow deployment of any application to the Kubernetes.

Pod is a fundamental object of Kubernetes that is deployable. The pod is a group of containers with shared storage and network. All the containers in one Pod are always scheduled on the same worker. The most common use case of Pod in Kubernetes is one container per Pod. Besides the application containers, Pod can also contain the *init container. Init container* runs during Pod startup, and they are usually responsible for doing some initial configuration or other pre-run steps.

Service provides developers functionality to expose Pods and containers to the other Pods running in the same cluster. Each Service object defines a new endpoint that includes a specific network port. If the Service should be accessible from the outside world, developers have to configure also a *Load Balancer or Ingress controller*[3)].

Persistent Volume (PV) is a piece of storage allocated from the cluster storage. This storage can be either manually allocated by the administrator or dynamically with *Storage Class*[4)]. If developers want to create Persistent Volume dynamically, they have to do it through the Persistent Volume Claim (PVC). PVC declares that once an object that needs the specified storage exists, it dynamically creates new PV through the Storage Class that is afterward mounted to the object.

[3)]Both Ingress and Load Balancer assure that Service is accessible via an external IP address. More information can be obtained at `https://kubernetes.io/docs/concepts/services-networking/`

[4)]Storage Class describes a specific driver that allows automatic allocation of storage. A detailed explanation of Storage Class is available from `https://kubernetes.io/docs/concepts/storage/storage-classes/`

Replica Set (RS) defines Pod and amount of desired replicas of this Pod that should be up and running. Replica Set does not care about what replicas are doing. It ensures only that they are running. If any replica fails, it immediately creates a new one.

Deployment provides higher abstraction to Replica Set and Pod. Deployments are used to manage the Pods in higher abstractions than just Replica Sets. Deployments store the desired state of the application. If the Pod becomes unhealthy or fails, the Deployment automatically replaces this Pod. Besides that, Deployment allows the application to be automatically scaled. Developers can set thresholds based on CPU utilization; for example, if some Pod exceeds 80% CPU utilization, Deployment spin up a new replica. However, Deployment also supports a feature called *Rolling Update*. This means that Deployment will maintain multiple versions of the applications. The deployment will then progressively update all the Pods, maintaining applications available throughout the process.

Although Kubernetes is a very handful tool that offers a broad amount of tooling and other features, it also has some obstacles that developers have to overcome. One of those obstacles is delivering new applications to the production environment in an automated way. This process is also known as *Continuous Delivery (CD)*. Thankfully, there are solutions that can help with this, and one of them is *GitOps*.

GitOps is a specialized type of continuous delivery that assures the application's deployment is always up to date with a specific Git repository. This workflow is showcased in Figure below 4.3 [9]. Once any change happens in the Git repository, the system will automatically detect this change and replace/update all the application's current cloud deployments. This approach is especially helpful when the development team releases a new stable version of the application container due to found vulnerability[5]. As the container images are immutable, redeploying the application is the fastest way to mitigate the risks from CVE. Once the version is updated, the system can update all necessary deployment descriptors and immediately replace the new stable version. One of the cloud-native solutions for GitOps is *ArgoCD*.

ArgoCD is an open-source GitOps tool for Kubernetes. It provides a declarative configuration, which allows for seamless integration of almost every cloud-ready

[5] *CVE (Common Vulnerabilities and Exposures)* is an entry in the common vulnerabilities and exposures database

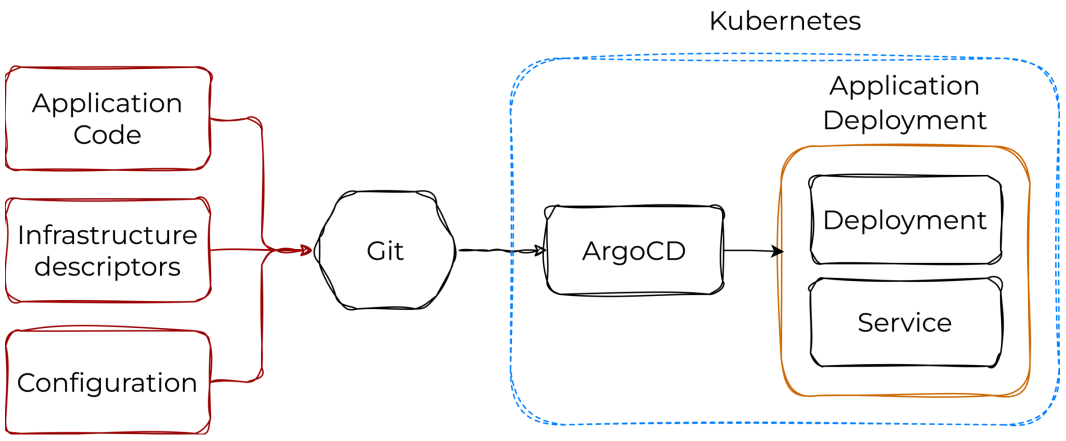


Fig. 4.3 GitOps workflow displayed using ArgoCD.

application [1]. Besides that, ArgoCD is also running as a Kubernetes deployment and can manage applications on multiple clusters. This way, it simplifies the transition from staging clusters into production ones.

Kubernetes is a powerful platform in modern software development. It allows developers to quickly move products from the development stage to production with containers. This Section covered all the fundamentals of Kubernetes. The most important outcome of it is the knowledge of the Kubernetes deployment objects such *Pod, Deployment, Service, etc.* Understanding these objects is necessary to move any application from a simple container into the cloud where it can really perform. Besides the deployment objects, the Section also covered Kubernetes architecture, which can help developers during the initial design process of the application that aims for maximal cloud potential.

Conclusions

This chapter aimed to review all the technologies necessary for implementing Distribution Builder. Gradually, it covered the most low-level technologies and principles, such as Design Patterns, Java, and Maven. These sections described the fundamentals and most significant bits of each subject.

During the overview of design patterns, the thesis dived into creational patterns. After considering all patterns, the Builder pattern emerged as the best solution for implementing the library because it allows nesting, which is necessary for building big objects with complicated structures. Then, the thesis moved to the Java and Object-Oriented Paradigm.

Knowledge of the OOP is absolutely necessary for the correct implementation of the Builder pattern. Because of that, the Section went through the core principles – Composition, Inheritance, Encapsulation, and Polymorphism. After a detailed explanation of these, the text evaluated Java and Maven. During the Java dive, the thesis covered mostly how JVM works and its advantages and disadvantages. After that, the thesis worked its way through the Maven, and `pom.xml` descriptor with special attention on the Maven assembly plugin will be necessary for the implementation.

After the fundamentals were covered, the thesis moved to more high-level technologies like Debezium, UI technologies, and the cloud. During the Debezium overview, the thesis covered all the primary techniques and principles on which Debezium stands. This Chapter also went through the Debezium Server, where we discovered what possibilities this technology unlocks. Lastly, it covered possible technologies for UI implementation and cloud deployment. Specifically, the thesis delved into Spring, Micronaut, ReactJS, Angular, and Kubernetes.

II. IMPLEMENTATION

This Chapter goes through the implementation of the Distribution Builder. However, before describing the implementation design, it is necessary to identify all possible use cases for both the standalone library and the UI.

Identifying library use cases is quite problematic because, in addition to users, the library will also be used in the UI. Figure below 4.4 showcases possible use cases for the library.

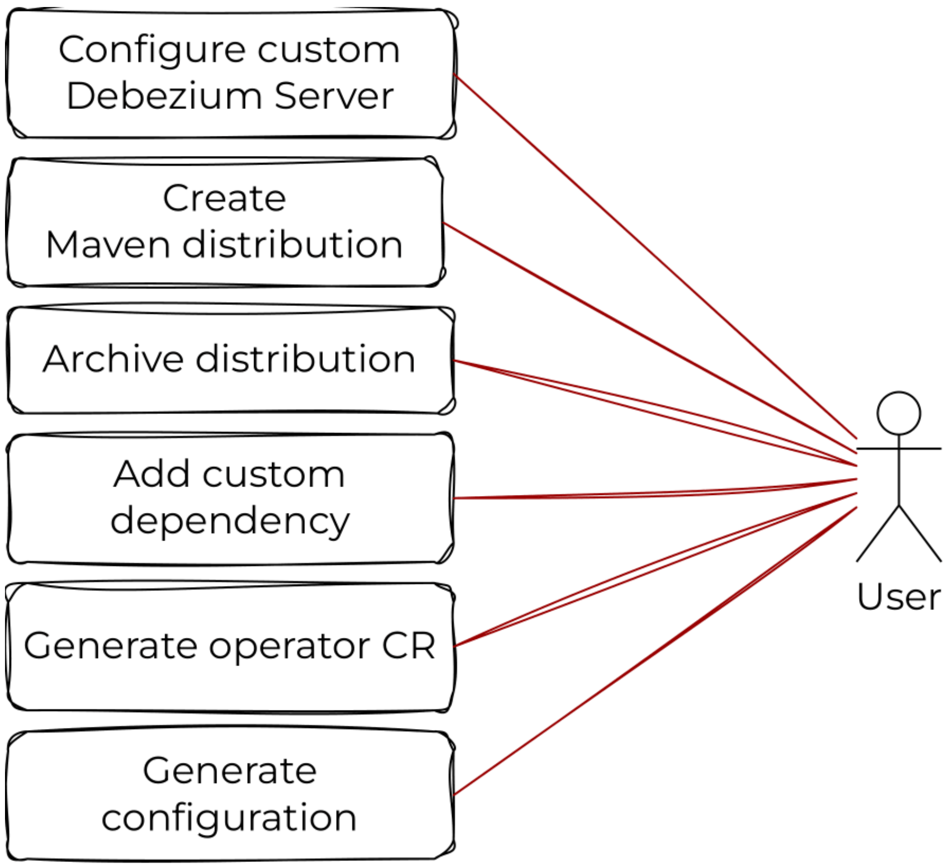


Fig. 4.4 Usecase diagram for library.

As you can see in Figure 4.4, there are six identified use cases for the library:

- **Configure custom Debezium Server** – The User is allowed to configure the Debezium Server based on the preferences. The library should map all the configurations to make it as easy as possible for the users.
- **Create Maven distribution** – As Debezium Server is based on the Maven build system, the library has to modify the descriptors to create the distribution as compact as possible.
- **Archive distribution** – Distribution is just a folder, which is not an ideal format for the users, as in most cases, the distribution has to be moved to a

specific path in the file system. Because of that, the library should provide the functionality that allows users to receive archives.

- **Add custom dependency** – In some situations, users might want to add, for example, custom transformations. In order to allow such behavior, the library has to allow custom Maven dependencies to be added to the distribution.
- **Generate configuration** – The First point mentions the configuration for the custom Server instance. The configuration of the server does not just involve mapping attributes to Java classes. The library has to allow serializing these classes to specific *properties* format, using *key= value* syntax.
- **Generate operator CR** – The Debezium Section 2 already mentioned the Debezium Operator project. As the operator is a great extension for the Debezium Server itself, it might be very useful to generate not only standard configuration files but also Debezium Server Custom Resource (CR). CR is essentially a custom Kubernetes object for the Debezium Server provided by the Debezium Operator. This file contains all the configurations for the Debezium Server instance.

The list above brings more clarity on what the library should offer. This set includes only the necessary features to provide the library with the best possible usability. Although the library and UI are pretty much tied together, the number of use cases for the UI is smaller. That is because the UI should be as simple as possible for this case. Use cases for the UI are displayed in the Figure below 4.5.

The figure identifies four use cases. Some of the use cases are exactly the same as the library, but for the UI, there is obviously an entirely different way to achieve these. Beneath, you can find an extended description for each use case of the UI:

- **Configurate custom Debezium Server** – Parallel to the library use case. The UI must provide proficient interface objects that are easy for the user to manipulate. For this case, the object should be something that provides the ability to pass user input – forms.
- **Generate custom archive distribution** – Once the server is fully configured and all inputs are passed, the UI should take the whole input, pass it to the library, and then return the archive as a downloadable file to the user.
- **Add custom dependency** – The UI should map the library functions for adding new dependencies to sufficient user interface structures.

- **Add custom truststore and keystore** – In order to allow security for the Debezium Server, it is necessary to bundle the keystore and truststore to the distribution. UI has to provide the user with a way to include those files in the generated distribution request.

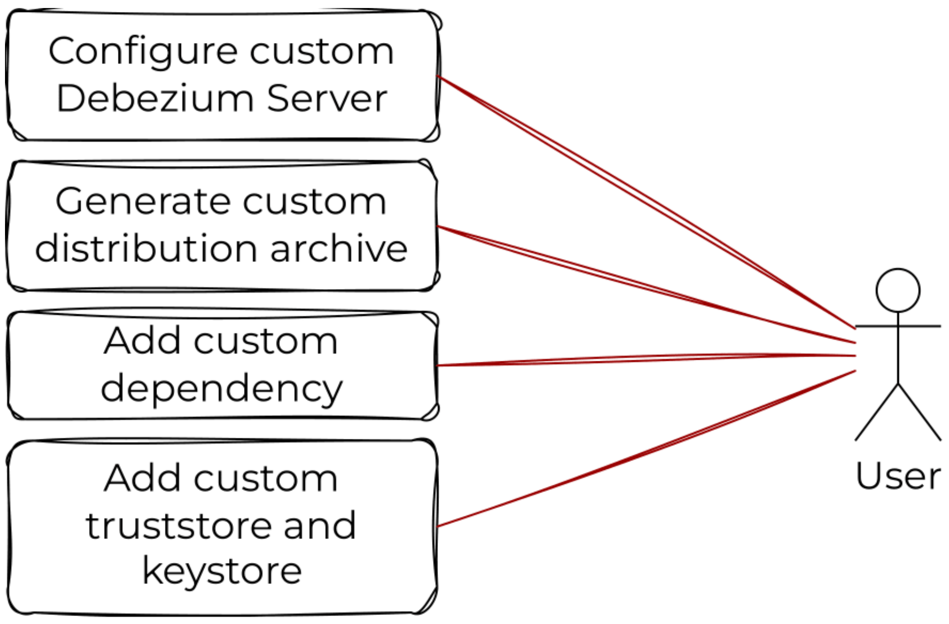

Configure custom Debezium Server
Generate custom distribution archive
Add custom dependency
Add custom truststore and keystore
User

Fig. 4.5 Usecase diagram for UI.

Identifying use cases is the initial step in the development process. Based on these use cases, the thesis can start with the design process. Designing such a complex system like this might be tricky. Because of this, the Chapter will introduce a couple of new diagrams. Specifically, we are talking about the *class and sequence* diagrams:

- **Class diagram** – Diagram describes the structure of the system by showcasing the class hierarchy. It models all relations between classes, but for this project, we will use only association (simple line without arrows), extension (simple line with the arrow), and implementation (dashed/dotted line with the arrow).
- **Sequence diagram** – Maps interactions between all parts of the system. For the thesis case, we will mostly talk about the interaction between the user, library, frontend, and backend. Based on the designed interactions, it is much easier to develop a proficient public interface for each system.

5 Design proposal

The design process has to start from a central object. In this case the main object of the whole design process is the Debezium Server. The server object should contain all necessary attributes. Based on the Debezium Server configuration [4] it can be identified that the server must contain *source connector, sink connector, internal schema history storage, and offset storage.* Besides that, the design of the library still has to allow proper mapping of the attributes into the Maven dependencies or assembly plugin filters.

The first class diagram 5.1 shows the design of the Debezium Server object. Because of space limitations, the diagram is not fully complete, and it is missing specific interface implementations, attributes, and other important design methods. This trend will be used across the whole design Section. The complete class diagram of the library can be found in Appendix 1

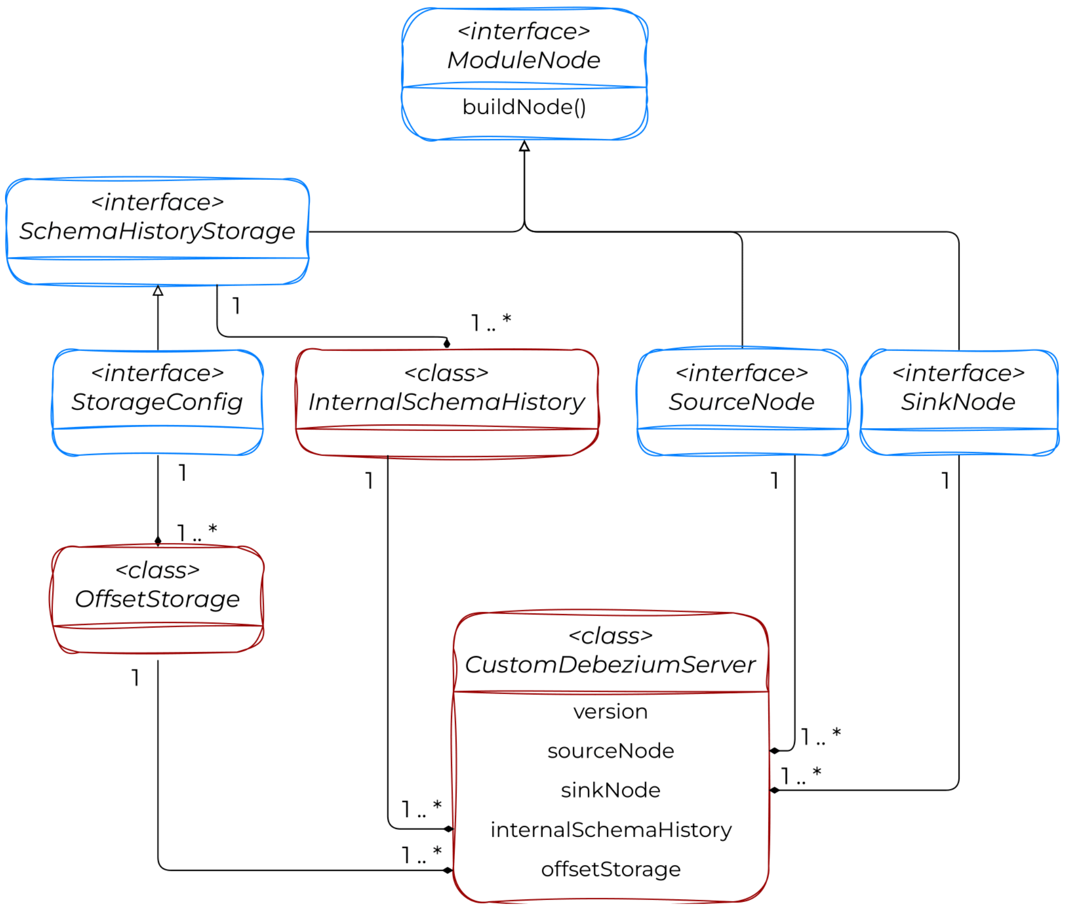


Fig. 5.1 Custom Debezium Server object class diagram.

The diagram contains a couple of interfaces that are very important for the whole implementation. The first and probably most crucial interface is `ModuleNode`. This interface effectively points to the object that should included in assembly filters or added to the subset of dependencies. You can see that the interface declares that all child classes must define the `buildNode` method. This method should be used to manipulate the Maven descriptors or to prepare the necessary structures for the descriptors. From the documentation, we can identify that for the functional Server distribution. The library must include a dependency for each already identified `CustomDebeziumServer` object attribute. From the Figure, we can see that all these attributes are bound to this interface.

The following important entities in the diagram are `SourceNode and SinkNode` interfaces. Each of these interfaces associates specific connector objects. The `SourceNode` groups all the database connectors, and a deeper description can be seen in Figure 5.2. On the other hand, `SinkNode` associates all sink connectors, showcased in Figure 5.3. Besides that, both interfaces extend the `ModuleNode`, which means that the implementation of the sink/source connector interface will add/filter appropriate dependency for the specific connectors.

The last important interface from Figure 5.4 is `SchemaHistoryStorage`. This interface groups all the storages that can store *schema history* for Debezium connectors. There are many storages that can serve both *schema history and offsets*. Because of that, the `StorageConfig` interface extends that `SchemaHistoryStorage`. A more specific description of the class hierarchy for storage can be seen in Figure 5.4. Implementations of these interfaces are then used in concrete classes included in the Debezium Server class as attributes. This design should allow the `CustomDebeziumServer` class to contain all the necessary information for the distribution build.

Based on the design for the `CustomDebeziumServer` class, we know that one of the attributes is `sourceNode`. This interface is the parent for all implementation of the Debezium database connectors. Although this might sound simple, it is not entirely true. Source connectors contain many different configuration fields, some of which are the same for all the connectors, some shared between general SQL connectors, and some specific to the exact connector. Mapping this relation is displayed in the Figure below 5.2.

The Figure above displays all the important relations with an example of two concrete implementations. The first noticeable change is that the design includes three new interfaces – `Config, YamlConfig, and PropertiesConfig`. These interfaces are essentially what force implementations to provide serialized configuration. The `Config` is essentially grouping the other configuration interfaces. More important interfaces are

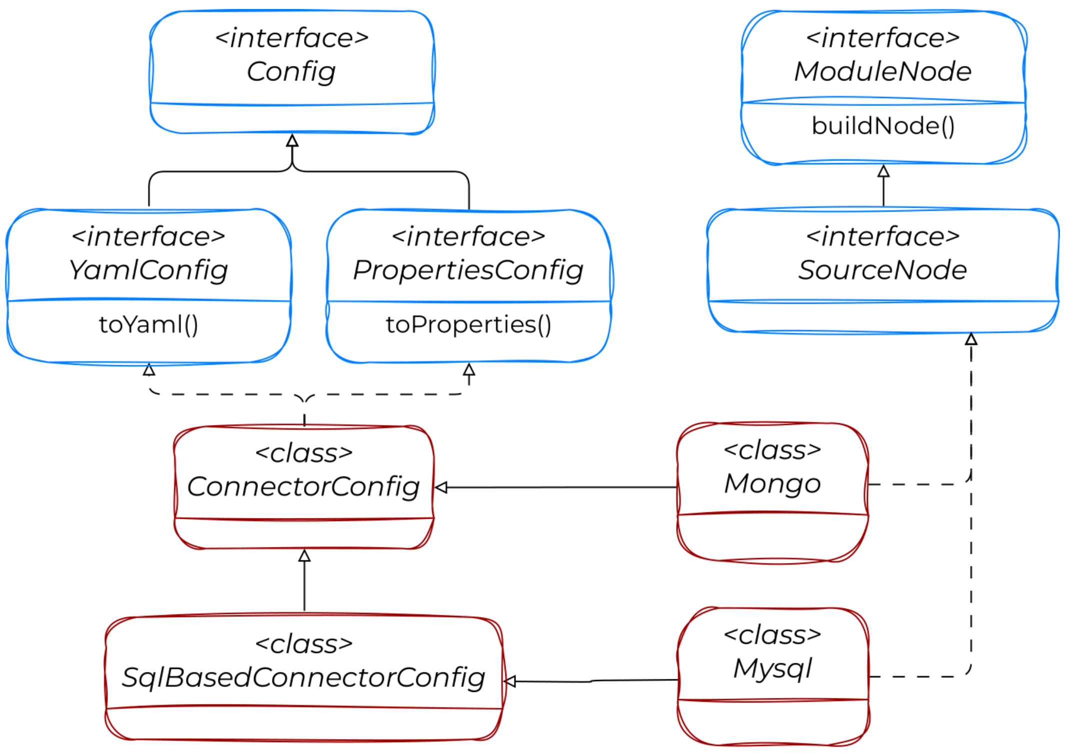


Fig. 5.2 Source node object class diagram.

the specific ones. `YamlConfig` instructs the implementations to implement `toYaml()` method which should contain serialization process which will return `Map<String, Object>` type.

The `HashMap<String, Object>` type is a collection of key-value pairs where the String declares that every key will be of String type and Object, on the other hand, states that each key will be the implementation of an Object class. The main reason for using Map is that the default serialization of this data type is exactly the same as *Yaml*.

You might ask why the *Yaml* when we have already talked about the configuration being in *properties* style. The *Yaml* is necessary to deploy the Debezium Server via Debezium Operator into the Kubernetes cluster. An example deployment file for the operator is displayed on the Listing 11 beneath.

Besides the configuration mapping, the Figure also showcases the configuration inheritance where the configuration is split into SQL-specific connector configurations, and the default configuration is shared across all the connectors. In Figure 5.2, you can identify two concrete implementations for database connectors – `Mongo` and `Mysql`. `Mysql` is a connector for SQL-based database, so it will contain configuration field from both `ConnectorConfig` and `SqlBasedConnectorConfig` classes.

```yaml
apiVersion: "debezium.io/v1alpha1"
kind: "DebeziumServer"
metadata:
  name: "custom-debezium-server"
spec:
  image: "IMAGE_PLACEHOLDER"
  source:
    class: "io.debezium.connector.mysql.MySqlConnector"
    config:
      debezium.source.tasks.max: 1
      debezium.source.database.port: 3306
      debezium.source.database.hostname: "test-dtb"
```

Listing 11 Example of yaml descriptor for the Debezium Operator with only source configuration.

On the other hand, `Mongo` connector is for *NoSQL* database, so it will inherit the configuration fields only from the common `ConnectorConfig` class. And, of course, the last piece of the puzzle, each implementation of the `SourceNode` interface will be transitively linked to `ModuleNode`, so it will have to implement the `buildNode()` method. The design for the sink side of the Debezium Server is essentially the same as for the source side. All the specific implementations for sinks implement all necessary interfaces – `YamlConfig, PropertiesConfig, and SinkNode`. Relations are showcased in Figure 5.3.

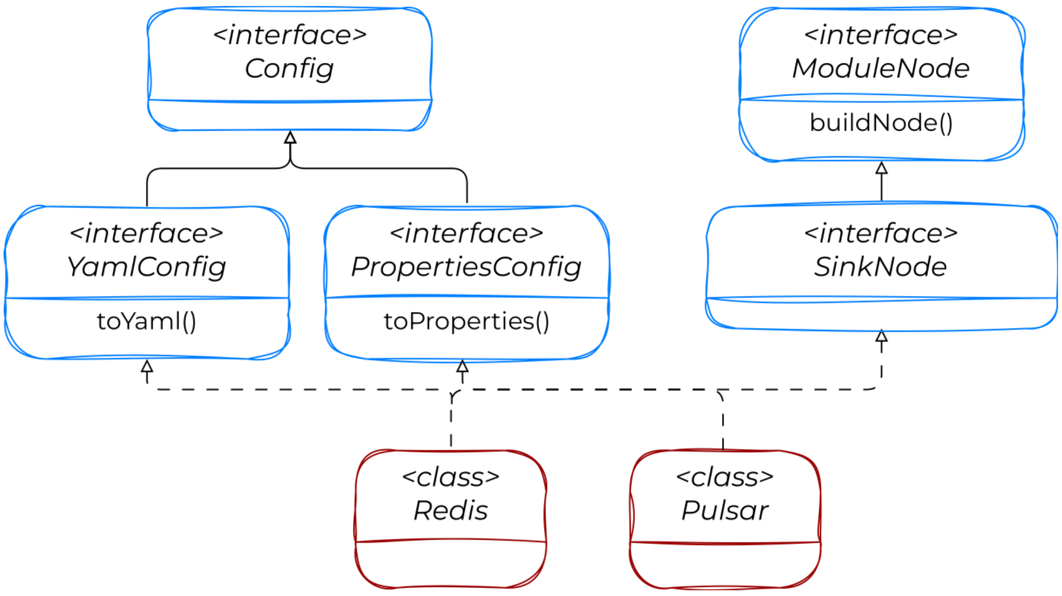


Fig. 5.3 Sink node object class diagram.

Both sink and source connector designs were relatively straightforward. The more problematic part of the Server is the storage. Debezium Server must include a configuration and storage adapter for both schema history and committed offset storage. The main obstacle that has to be addressed is that some storage implementations, like Redis, can be used for both offset storage and schema history storage.

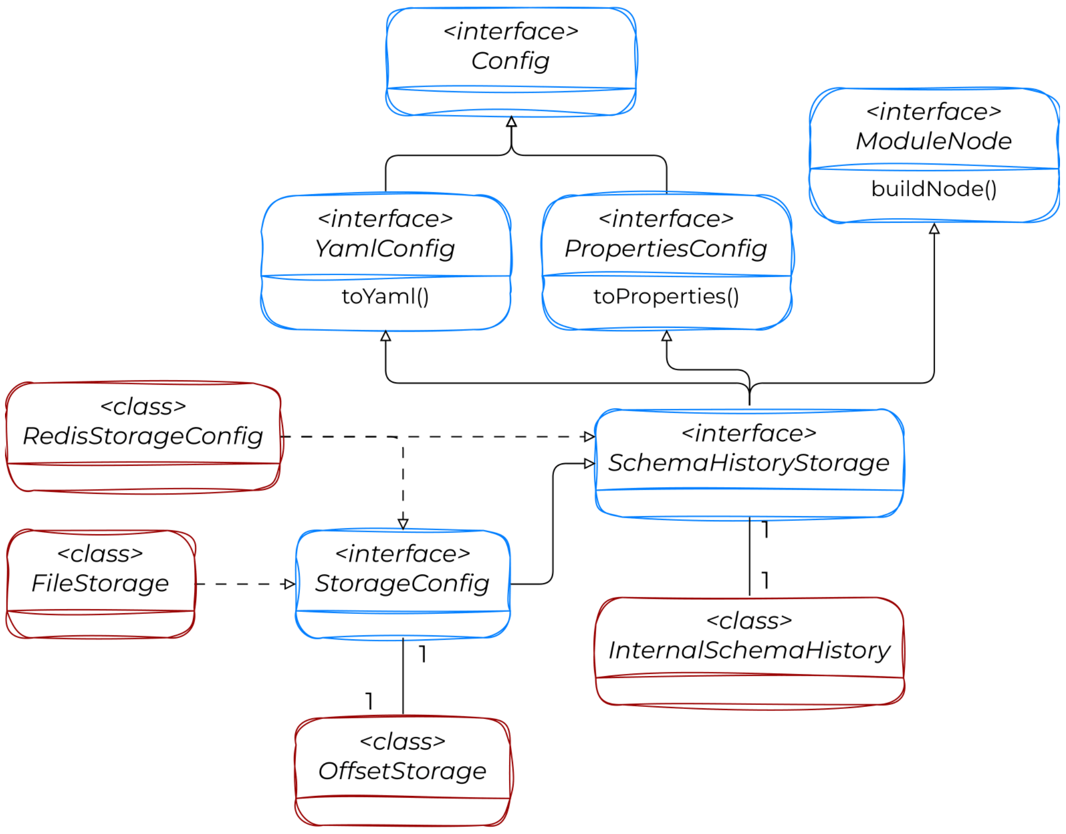

<interface> Config
<interface> ModuleNode
buildNode()
<interface> YamlConfig
toYaml()
<interface> PropertiesConfig
toProperties()
<class> RedisStorageConfig
<interface> SchemaHistoryStorage
<class> FileStorage
<interface> StorageConfig
<class> InternalSchemaHistory
<class> OffsetStorage
1
1
1
1

Fig. 5.4 Storage object class diagram.

Class design for the storage is displayed in Figure 5.4. From the diagram, you can see that the storage uses precisely the same top-level interfaces as the main parts of the Debezium Server. From the figure, we can identify two new interfaces – `SchemaHistory Storage and StorageConfig`. These two interfaces can then be implemented by specific storage adapters. There are two example implementations of those adapters. The first one is the `FileStorage`. This adapter can only be used to store offsets. The second implementation is `RedisStorageConfig`. This storage is the opposite case than `FileStorage`. It can serve as both offset and schema history storage.

These class diagrams are the core of the library design. Based on those diagrams, the implementation process can start. But there are still a few parts missing. You might be concerned about why any of those diagrams do not show signs of *Builder* pattern 1. There is a simple answer to this. Today, there are many technologies that are making lives of the developers easier, and there is one that can help with turning the *Builder* pattern into reality – *Sundrio* [25].

Sundrio is a Java library that allows developers to completely strip the code from the *Builder* implementation and generate the builders during the build process. On top of that, the Sundrio leverages the functionality on *Fluent API*. Fluent APIs are used to make code more readable and easier to read. The main characteristic of Fluent APIs is that they chain the method calls [7]. The main difference between fluent and non-fluent implementation of the builders is that fluent implementation is immutable [6]. Immutable, in this case, means that the builder will create a new object instance every time, not just change the internal state and return the same reference as the classic builder implementation.

Sundrio allows builders and specific classes to use Fluent API. During the compile process, Sundrio generates 3 additional Java classes – `FluentImpl, Editable, and Builder`. `FluentImpl` transforms the class into Java Fluent API implementation. `Editable` allows editing the fluent objects. It contains only one method `edit`, which returns `Builder` for the current class. `Builder` class contains builder for the fluent implementation.

You might ask how the Sundrio recognizes for which class it should generate the builder. The library works around Java annotations. Specifically, Sundrio provides developers the `@Buildable` annotation. This annotation instructs the library to generate all necessary classes for the specific annotated class. Overall, the Sundrio is a very user-friendly library that provides very powerful features that significantly help the developer. Sundrio is very popular in the Kubernetes world, where you always have to compose complicated objects (both Debezium Operator and also Strimzi[1)] project uses Sundrio).

Once the initial design of the library code structure is done, it is time to design how the user should interact with each part of the Distribution Builder and other systems. From the beginning of this chapter, it is apparent that sequence diagrams are needed.

In order to define the complex interactions between the user, UI, backend, and library, it is necessary to define interactions between the user and the library. The Figure 5.5 displays these interactions. From the diagram, we can see that the first step is that the user will build the `CustomDebeziumServer` object, which should include the whole configuration which should be included.

This building process is realized with the Sundrio described above. Once the build is complete, the user will obtain an immutable `CustomDebeziumServer` object. Then, the user can hand this object to the library and instruct it to build a custom distribution.

[1)] Open-source Apache Kafka operator.

We are now at the point where the library has everything it needs to build the distribution, and it has to complete several steps in order to create the distribution (some of the steps are stripped from the diagram in order to keep the diagram relatively straightforward).

The first step that the library must take is to obtain the Debezium Server sources; this might be done either by cloning the upstream git repository or by supplying it to the user. Then, the most critical part is modifying the Maven descriptors in order to minimize the number of dependencies in the final distribution. This could be achieved by modifying the POM or assembly descriptor, which will be described later.

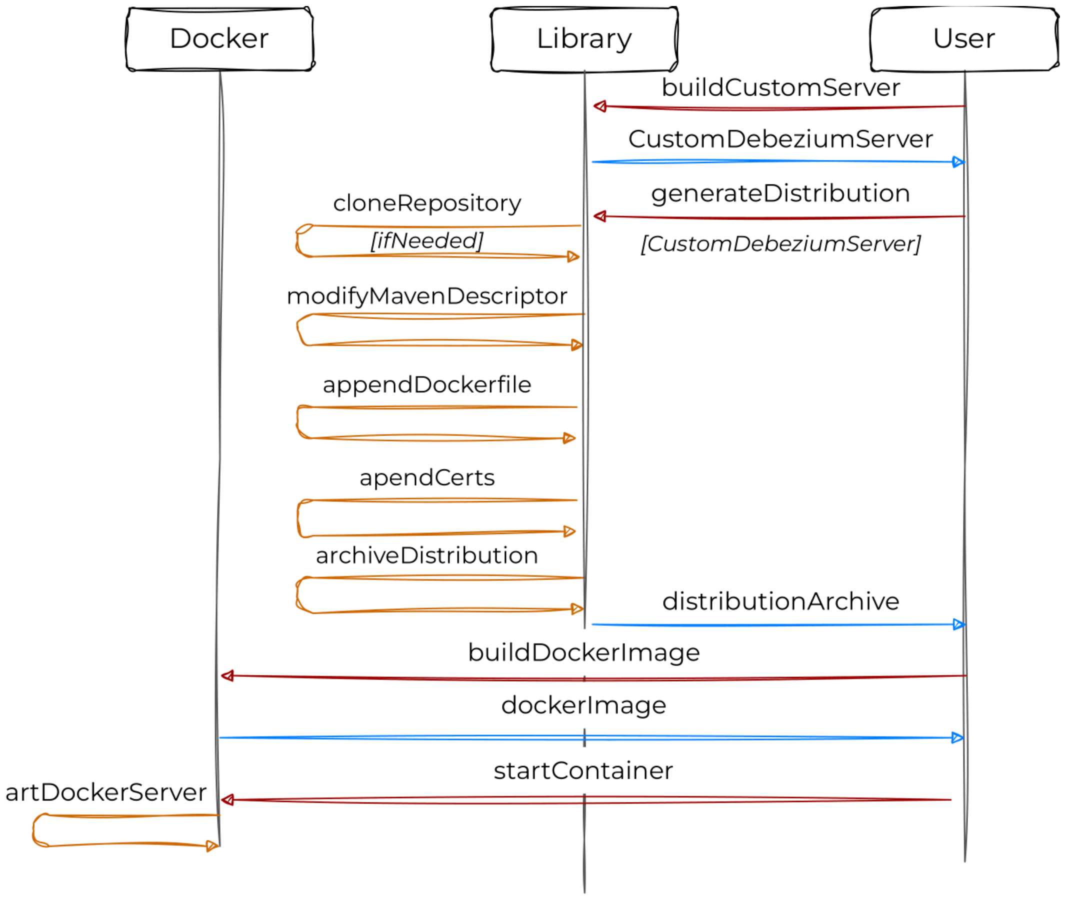


Fig. 5.5 Sequence diagram of java library.

Based on the use-case diagram, we can identify the rest of the necessary steps. First, the library needs to prepare the Dockerfile for the distribution and then append it to the specific place in the distribution sources.

The next operation is certificate cloning. If the user supplies the certificates that should be added to the distribution, the library has to take those files and clone them into specific places in the distribution sources so they are later reachable during the container image build. The last step which can do is to archive the sources, including new Dockerfile, certificates, and other supplementary files. The archive should be created at the specific path supplied by the user.

Once we have defined the sequence diagram for the standalone library, we can move to the higher lever. We can determine how the user will communicate with the UI and how the UI will then cooperate with the library. The Figure 5.6 specified four actors in the sequence:

- **User** – A person who uses the library.
- **Frontend** – Part of the UI is visible to the user. It provides necessary components that allow users to pass all information to the backend in the correct format.
- **Backend** – User interface logic. It exposes all endpoints that are accessible from the frontend.
- **Library** – The standalone library.

These four actors were chosen because the implementation based on the technologies specified in Theory chapter I leads to code separation that will look exactly like that. The workflow of the UI can be separated into three essential steps. The first step is startup. During the startup, the Backend has to load the configuration metadata from the library. This metadata should contain all the configuration properties available for the Debezium Server. The UI needs to have all the information about the configuration before the UI renders. This step happens only between the backend and the library. The UI is not yet loaded because the user has not yet loaded the page. At the beginning of the second step, the user loads the page. Once the user tries to load the UI, it makes a request to the backend to obtain all metadata. Backend returns already obtained and metadata model and returns it to the UI. UI then renders all the components and returns the generated frontend stack to the user.

The user now has a complete UI, which he can interact with and fill in all the fields he wants. Once he has filled in all the fields and uploaded the files, he can instruct the UI to generate a custom distribution. The UI now has to serialize all the data obtained from the user. Once this process is complete, the UI passes all the information to the backend. The backend then parses the data and prepares everything, like storing the files in proper locations. After the preparations are completed, the backend calls the library to prepare the distribution. Everything that the library has to do for this

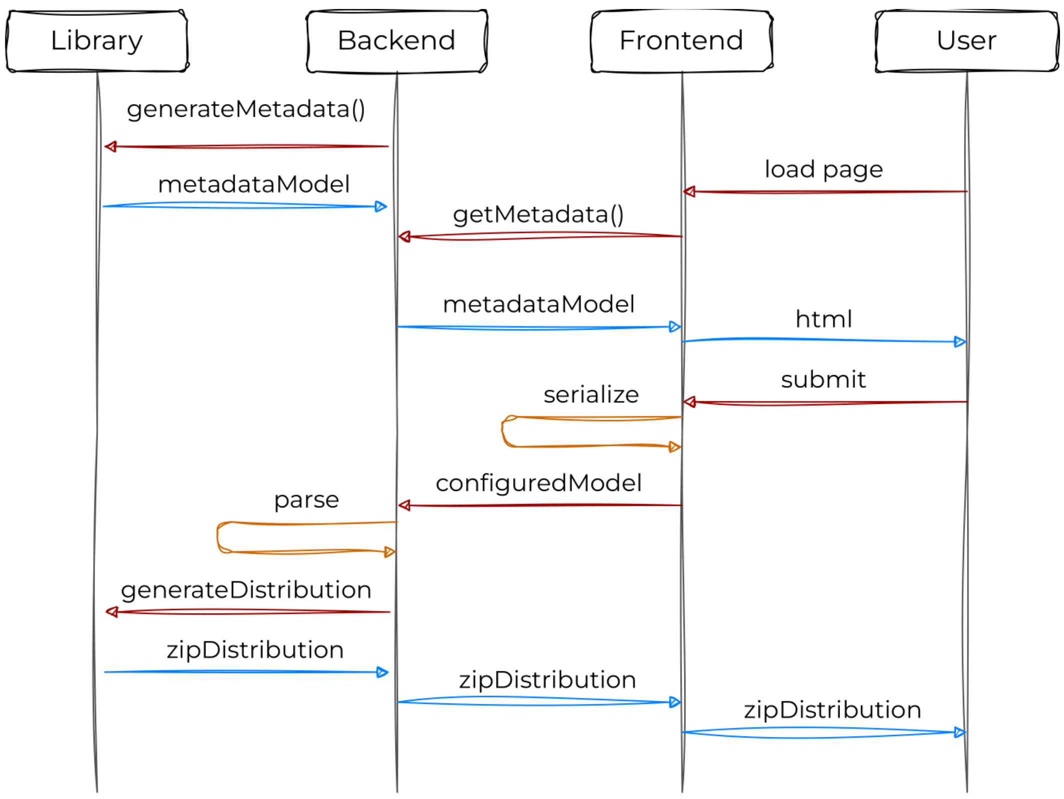


Fig. 5.6 Sequence diagram of Distribution Builder UI.

step to be completed is showcased in the previous sequence diagram 5.5. Once the archive is prepared, the backend transforms it to the byte sequence handed over to the front end and then to the user. The user's browser sees this byte sequence as the file, so it will immediately download it. At this point, the whole design of the library is complete. We know what the implementation of the standalone library should look like. We discovered that for library class design, the implementation has to take care of several main key obstacles:

- **Maven descriptors** – All the important library objects have to contain the necessary information to modify the Maven descriptors. And it depends on the implementation of how this information will be used.
- **Configuration mapping** – The library has to contain all the configuration mapping for the Debezium Server. This might sound like an easy task, but it is not. Implementation has to strip the configuration into the smallest possible formations.
- **Configuration serialization** – All the configuration objects translate into properties or YAML format in different ways, so every object has to implement its own serialization.

Besides that, the main obstacle to the implementation to overcome would be to generate the metadata model for the UI. Parse it in the frontend and dynamically generate the UI components based on this model. This step might be especially tricky because you basically don't have any information about the model prior to the render except the structure of it.

6 Code overview

The code is divided into three separate projects. This might just be a temporary solution, as the right place for this project is still being discussed. Essentially, the whole Distribution Builder, including UI, could be in one repository. This Section goes through each part of the project and describes the most essential pieces of code found there. Besides that, it also mentions the biggest obstacles that were found during the implementation and how the implementation overcame those.

Library

Based on the design diagrams created in the previous Section 5, it is clear what are the steps that should be covered by the library. This section will go through the most exciting bits of the implementation. The first obstacle found during the implementation was that the library had to append only defined attributes to the properties or YAML configuration.

```java
package io.debezium.server.dist.builder.modules.config;

public interface ConfigBuilder<T extends Config> {
    void put(String key, Object value);
    <E extends Enum<E>> void putEnumWithLowerCase(String key, E e);
    <E extends Enum<E>> void putEnum(String key, E e);
    void putList(String key, List<String> list);
    void putBoolean(String key, Boolean value);
    void putAll(T p);
    void putAllWithPrefix(String prefix, T p);
    void putAllWithPrefix(String prefix, Map<String, Object> map);
}
```

Listing 12 Library listing – configuration builder interface

In order to achieve such a thing, the library introduces `ConfigBuilder` interface. This interface is built around generics. On the Listing 12, you can see that the implementation of this interface needs to pass a type argument. This argument would be either `PropertiesConfig` or `YamlConfig`.

These two interfaces were introduced in the design proposal Section. This eventually means there will be two implementations of the `ConfigBuilder`, first for the properties and second for YAML. The methods declared in the interface allow adding all the possible objects to the properties with various modifications like prefixes and others.

One more bit that could catch your eye is `<E extends Enum<E»`. This defines that `E` can be any Java enumeration implementation.

One of the next steps introduced during the design process is for the library to compress the distribution into a compact archive. That alone would not be anything interesting, but Java does not provide tooling for compressing the whole directory. Because of that, the library has to introduce a process that will go through the files and add them one by one to the archive. The iteration over all the files in the directory is achieved by implementing `java.nio.file.Files.walkFileTree`.

```
                    ...
try (ZipOutputStream zos = new ZipOutputStream(os)) {
        Files.walkFileTree(source, new SimpleFileVisitor<>() {
            @Override
            public FileVisitResult visitFile(Path file,
                                                 BasicFileAttributes
                                                     attributes) {
                try (FileInputStream fis = new FileInputStream(file.
                    toFile())) {
                    Path targetFile = source.relativize(file);
                    zos.putNextEntry(new ZipEntry(targetFile.toString
                        ()));
                    byte[] buffer = new byte[1024];
                    int len;
                    while ((len = fis.read(buffer)) > 0) {
                        zos.write(buffer, 0, len);
                    }
                    zos.closeEntry();
                    ...
```

Listing 13 Library listing – directory compressor

That method goes over all the files, and on each file, it executes the `visitFile` callback. The implementation of this callback is the critical bit. At the beginning, a new zip entry is added to the archive.

This essentially moves the `ZipOutputStream` pointer to the beginning of the new file. After that, the file is continuously read in chunks until it is complete. These chunks are also continuously written into the `ZipOutputStream`. Once this process is done, the entry is closed, and the callback is complete.

This implementation is essentially universal and should work on all possible file sizes. There is a possible modification that would make a method simpler. We could replace the whole read and write loop with `java.nio.file.Files.readAllBytes(file)` but that is a little dangerous because this method might fail for larger files with `OutOfMemoryException`. Even if the occurrence of a file such as big is very minimal, the author decided to use a more generic approach showcased on the Listing 13.

From the previous section, we also know that all the communication between separate system parts is handled via JSON format. Commonly, the JSON arrays are interpreted in a specific format. This format is showcased on the Listing 14.

```
[
"item␣1",
"item␣N"
]
```

Listing 14 JSON array

However, forcing the users in the UI to construct such an array with multiple separate inputs would be overwhelming for such a widely scoped configuration. Most applications, including Debezium, use just the comma-separated items to interpret arrays. Implementation of the library uses the *FasterXML/jackson* [1)] library for all the serialization and parsing work. In order to allow parsing the comma-separated array format, Jackson's default behavior must be overridden.

The Listing 15 displays the implementation of custom `DeserializationProblem-Handler`. This handler is called whenever Jackson's parser encounters a problem with the input syntax. In this case, the custom deserializer overrides this callback and checks if the input value is String, and the expected type should be collection-like. If the conditional is true, the input sequence will be parsed as a comma-separated list. Otherwise, it will just call the default `DeserializationProblemHandler` to proceed as usual, which, in this case, means raising an exception.

During the implementation of the library, a couple of obstacles occurred. Most of them were, thanks to good design, very minor and could be fixed reasonably simply. However, there is one issue that has still not been figured out. How will the implementation construct custom Maven distribution?

The Theory Chapter I presented that there are two possible ways how to achieve the desired state, via modified `pom.xml` or `assembly.xml`. Choosing between these two is very hard, and choosing the right one without further investigation is impossible.

[1)] Information about the Jackson library can be obtained from: `https://github.com/FasterXML/jackson`

```java
public class CommaSeparatedListProblemHandler extends
    DeserializationProblemHandler {
    @Override
    public Object handleUnexpectedToken(DeserializationContext ctxt,
        JavaType targetType, JsonToken t, JsonParser p, String
        failureMsg) throws IOException {
        if (t == JsonToken.VALUE_STRING && targetType.
            isCollectionLikeType()) {
            return deserializeCommaSeparatedList(targetType, p);
        }
        return super.handleUnexpectedToken(ctxt, targetType, t, p,
            failureMsg);
    }

    private Object deserializeSimpleCommaSeparatedList(JsonParser
        parser) throws IOException {
        String[] vals = parser.getText().split(",");
        return new ArrayList<>(Arrays.asList(vals));
    }
                              ...
}
```

Listing 15 Library listing – custom deserializer for comma-separated array

At first look, the `assembly.xml` modification seems like a better approach, as this descriptor is specifically for the distribution generation. Let's take a deeper look at what would be necessary to go this path. As already mentioned earlier in the thesis, this descriptor is composed of filters. These can be either include or exclude type. Obviously, the first thought is that it is pretty simple: exclude everything and include only dependencies for source, sink, etc. That would definitely be a very clean approach.

However, it is impossible to do it. If this descriptor excludes everything at the start, it will exclude all the transitive dependencies. So, the library would need to obtain a dependency tree for each requirement and then add all those libraries as explicitly allowed in the descriptor. That might be all right if we spoke about 2-3 dependencies. In the case of the Debezium Server, we are talking about a much higher number. So, this approach is not an option.

However, there is still the opposite way to handle the dependencies in that descriptor. If the library uses the exclude filter and explicitly excludes all other dependencies than the required ones, everything should be in place. Well, that sounds a little too easy, and of course, there is a catch! The Maven assembly plugin does not have totally deterministic behavior regarding transitive filtering. So, it might happen that if the library excludes some dependency, it would exclude this dependency and possibly all its transitive dependencies. And that could possibly exclude some child dependency on distribution requirements. During the implementation, the author even found the issue directly in the Debezium Server that completely corrupted the current distribution [2]. This issue highlights how dangerous this path is and how easily it can go sideways.

So, using the `assembly.xml` to construct a custom distribution does not sound like a good idea. So it is evident that the only way this can be implemented is via modifying the distribution `pom.xml`. The default `assembly.xml` will include all dependencies specified in the `pom.xml`, so eventually, the only thing needed is to add only the necessary dependencies.

As the `pom.xml` descriptor is the XML document, which means it can be manipulated via *DOM (Document Object Manipulation API*. The DOM allows developers to manipulate the XML document programmatically. Firstly, the library has to prepare a base template that should contain all information that is shared across all the distributions. This base is included in thesis Appendix 2.

It can be parsed into the DOM document. Before the implementation can append all necessary dependencies, it is necessary to find the correct parent tag for where these should take place. This can be achieved just by searching for the `dependencies` tag. The base contains just one of those, so if implementation finds more of them, it can assume the base is corrupted. Once the implementation identifies the `dependencies closure`, it can build the `dependency` closure for each dependency.

Construction of that closure must be implemented for each object representing the dependency as described in the design Section. Debezium dependencies do not require setting the version because they will inherit it from the parent. Building a DOM node for the dependency is showcased in the Listing 16. This section went through the most essential bits of the library implementation. Specifically, it covered building configuration sets, archiving the distribution folder, and parsing specific comma-separated arrays. Lastly, the Section described the most crucial bit, which is building a custom distribution. It goes through all the possible implementation possibilities. With all the gathered information, the thesis can now proceed to the Backend implementation.

[2]Pull request with the fix can be found at: `https://github.com/debezium/debezium-server/pull/73`

```java
 private static Node buildBase(Document document, String artifactId) {
    Node dependency = document.createElement("dependency");

    Node groupIdNode = document.createElement("groupId");
    groupIdNode.setTextContent(groupId);
    dependency.appendChild(groupIdNode);

    Node artifactNode = document.createElement("artifactId");
    artifactNode.setTextContent(artifactId);
    dependency.appendChild(artifactNode);

    return dependency;
}
```

Listing 16 Library listing – Maven dependency node builder

Backend

The implementation of the Backend is realized with the Spring Boot framework. Thanks to that, the implementation can be really simple, without any necessary overhead. If we look into this through the MVC model, the backend is the controller for the Distribution Builder system. This controller must manage two desired endpoints. One is for distribution generation, and the second will allow the UI to obtain metadata.

This endpoint basically takes the generated JSON metadata file and returns it as a stream of bytes. The generation of metadata must take place right after the initial startup of the backend. This can be achieved with `@PostConstruct` annotation. Which instructs the Spring Boot to run a specific method. This method is displayed in the Listing 17. You can see that this process precisely follows what was specified in the

```java
@PostConstruct
public void generateMetadata() throws IOException,
   ClassNotFoundException {
    logger.info("Generating␣metadata");
    metadataFile = MetadataUtils.getMetadataFilePath();

    MetadataGenerator metadataGenerator = new MetadataGenerator();

    metadataGenerator.generateMetadata(
    new FileOutputStream(metadataFile.toAbsolutePath().toString()));
    cloneServer();
}
```

Listing 17 Backend listing – metadata generation

sequence diagram 5.6. The second endpoint is used for the distribution generation. What is interesting about the endpoint is that it expects the request to contain two files (keystore and truststore). That alone would not be anything interesting. But these files are not transmitted as a whole. They are separated into multiple chunks. Thankfully, Java offers a solution to accept such files – `MultipartFile`. The controller waits for the whole file to arrive and then completes it into this object, which then can be manipulated just as a regular Java File object.

Overall, the implementation of the Backend is the simplest of all the system parts. Spring helps the developer and takes care of all the problematic parts like the `MultipartFile` and handling the HTTP protocol. The more problematic part of the implementation takes place on the frontend side.

Frontend

The frontend of the application is implemented using the Angular framework. The implementation is separated into two main components. The first component (`main-page`) is just a simple component that returns a page that contains information about the application. It is basically an alternative to *About Us* page that you can find on almost every page. The second component (`distribution-builder`) is used to configure custom server distribution.

This component is much more complicated. First, metadata has to be retrieved from the backend. This is handled by `metadata-http` service. This server wraps up the Angular HTTP client implementation into prefabricated methods. HTTP implementation that is provided by Angular works asynchronously, which brings complications. The component needs the metadata before it starts the rendering. Thankfully, there is a solution that can help overcome this obstacle.

The Listing 18 displays the initialization of the `distribution-builder`. You can see that during the initialization, the method calls for the HTTP service to retrieve metadata. But this call is async, so the method will only call that and then immediately move away from it. At this point, all templates have a conditional that disables the render process if the `ready` variable is false. Once the async call completes, it executes the callback.

At first, the callback parses the metadata using the `metadata-parser` service. Then, it sets all the attributes needed for render, sets `ready` to true, and calls the `cdRef` object

```
ngOnInit(): void {
    this.serverConfig = new DebeziumServerFormBase<any>();
    this._form = new FormGroup<any>({});

    this.httpService.getMetadataFromBackend().then(value => {
        this._form = this.parserService.parseMetadata(this.
           serverConfig, value);
        this.dependencyList = this.serverConfig.children[4];
        this.ready = true;
        this.cdRef.detectChanges();
    }).catch(() => {
    //error processing - shortened for listing
        console.log("Error!");
    });
}
```

Listing 18 UI listing – initialization of the component with async client.

to detect changes. This object can trigger the rendering of the page if something changes in its internal structure. At this point, it detects that the internal attributes of the component changed their value, so it triggers the render. Once this happens, the templates again check if the `ready` attribute is true. If yes, all components are rendered, and the user receives the prepared UI.

You might ask why there is a separate service for parsing the metadata when it is just a simple JSON file. Well, this is not just regular parsing because besides just parsing the JSON, it must also prepare the whole `FormGroup` structure (described in Angular Section 3.2) because templates need complete information about the form structure before they render.

So, at first, the parser parses the JSON into the `MetadataObjectModel` object. Once that is done, the service goes through that object and all its children, and for each, it creates a specific form entry. This entry can be either `FormGroup`, `FormArray`, or `FormControl`, depending on the item type. Besides that, it also prepares an additional object `DebeziumServerFormBase` for each item. These objects then essentially form a double-linked list that contains additional information about the form entries.

The most intricate part of the UI is forming and rendering the HTML entries that are correctly linked to all callbacks in Angular. What makes this process extremely complicated is that the UI does not have any information about the configuration it will display before it receives the metadata. So, the implementation must be flexible and work with a dynamic number of components.

You might think that this should not be so complicated. You properly implement recursion and create specific components for each type of entry, and everything should be all right.

Well, that is not possible because once one component is complete and returned to the main one, Angular will trigger a form validation process that will fail immediately because the form structure will not be complete. That essentially means that the generation of all HTML components must take place in a single component template. In order to satisfy this requirement, the implementation must find a way to iterate over metadata and generate the components without triggering the validation process. Thankfully, there is a solution that allows such behavior – `ng-template`.

This element allows the implementation to wrap up a piece of code that will be separated from the rest of the template. It will create a virtual environment, including variables available only for its scope. But most importantly, the `ng-template` allows the implementation to call itself recursively. That is a massive thing because going through the metadata object in iteration would make code much more complicated. '

```
<ng-template #treeForm let-data let-group="specificGroup">
                              ...
    <div class="container">
      <!--   Handle interface implementations     -->
      <ng-container *ngFor="let␣option␣of␣data.options">
        <ng-container [ngTemplateOutlet]="treeForm" [
            ngTemplateOutletContext]="{␣$implicit:␣option,␣
            specificGroup:␣this.getFormGroup(option.label,␣group)␣}">
          <!--      Recursion template          -->
        </ng-container>
      </ng-container>
    </div>
                              ...
</ng-template>
```

Listing 19 UI listing – recursion in HTML templates.

On the Listing 19, you can see code snippet from the `distribution-builder` component template. You can see that this specific piece of code goes over all the options included as an implementation for the interface. For example, you can imagine the interface as `SourceNode` and options as `MySQL, PostgreSQL`. The template iterates over those options, and for each option, it calls the same `ng-template`, but this time, it passes only the options object and current form group so that the recursive call will work only of this tree subset.

Once the recursive iteration through the whole metadata model is completed, validation is triggered, and the HTML is passed to the user. The user can then specify all desired configuration bits, upload all the files, and instruct the UI to generate the distribution. An implementation serializes the configuration, appends the files, and sends it to the backend endpoint. Once everything is processed on the backend side, the distribution archive is returned as a Byte stream. This stream is then handed to the user as a downloadable file. This process is handled by `file-saver` library[3)].

The implementation of this system was very interesting, and it discovered many problems that one might not think existed. Implementation of each system part is exceptionally different; the library is strictly object-oriented, where almost everything is mapped to a specific class. The UI, on the other hand, could be described in one single world – recursion. But implementation is not everything. It is necessary to do some testing before going to some publicly reachable environment. Because of that, the next section will go over the test implementation for this system.

[3)] More information about the library can be obtained at: `https://github.com/eligrey/FileSaver.js`

Testing

Testing of Distribution Builder is quite tricky as it is separated into three different projects. In order to achieve the best possible test coverage of all the functions, the system uses two layers of testing – unit and integration.

Unit tests are mainly located in the library repository[4]. The library is a great target for unit testing because it offers many small features that can be tested individually, which is essentially the main focus of the unit testing. These tests are written using *JUnit* framework. This framework is, without doubt, the most used testing framework for Java applications [12]. It handles all test execution so developers can just write tests. The Listing 20 displays one of the tests implemented for the library. You can see that there is `@Test` annotation, which marks the `put()` method as *JUnit* test. In the method's body, you can see some object operations, and after that, *assertThat* method. This method effectively creates the test output. If those two arguments match, the test passes. Otherwise, it fails.

```java
    @Test
    void put() {
        HashMap<String, Object> props = new HashMap<>();
        props.put("test-key", "value");
        YamlBuilder pb = new YamlBuilder();
        pb.put("test-key", "value");
        pb.put("test-null", null);
        assertThat(pb.getYaml(), is(equalTo(props)));
    }
}
```

Listing 20 Example implementation of the unit test in Java.

Integration tests are used to verify the functionality of the whole system working together. Because of that, it is necessary to build the integration tests around the UI, which integrates all the parts of the ecosystem. Testing the system through UI can be complicated because it requires the test system to act as a user. Thankfully, there are many testing frameworks that allow such behavior. One of them is *Playwright*. *Playwright* is a testing framework that allows developers to locate specific components of the web page and trigger specific events like mouse clicks or keyboard events.

[4]Repository can be found at: `https://github.com/obabec/debezium-server-dist-builder`

The integration tests are in the UI repository [5]. Those tests progressively fill out different configuration fields and then request the distribution archive. Once the archive is generated, the tests take it and compute the md5 checksum on the file. Then, the tests compare this checksum to the expected one. If the checksum is the same, the test passes.

The Listing 21 displays one of the integration tests developed for the Distribution Builder. You can see that the tests open up the page on a default network port. Once the page is open, it will navigate to `Builder` subpage. Then, it generates the empty distribution. If the empty distribution is generated without an exception and it is empty, the test passes.

```
import { test, expect } from '@playwright/test';


test('test-empty-distribution', async ({ page }) => {
  await page.goto('http://localhost:4200/');
  await page.goto('http://localhost:4200/intro');
  await page.getByRole('link', { name: 'Builder', exact: true }).click
      ();
  const downloadPromise = page.waitForEvent('download');
  await page.getByRole('button', { name: 'Create␣distribution' }).
      click();
  const emptySum = '6a8c833ef6d8b0c943c1295016e2d6ed';
  const md5File = require('md5-file');
  md5File(await (await downloadPromise).path()).then((hash) => {
    expect(hash).toEqual(emptySum);
  })
});
```

Listing 21 Example implementation integration test using the Playwright.

Besides the tests the repositories also contain preparation for ArgoCD automation. This means that each part of the project contains Kubernetes deployment files (PVC, Service, Deployment)[6].

[5]Repository is located at: `https://github.com/obabec/dist-builder-Angular-ui`

[6]Each repository (API and UI) contains the folder deployment with all Kubernetes deployment files

7 Application in commercial security systems

In order to identify use cases in security systems, we have to look from a higher perspective. As mentioned, Debezium Server can serve as a very effective feed for machine learning or artificial intelligence (AI) in general. This is precisely where it could help security systems. There are two possible use cases where Distribution Builder and Debezium Server could help.

The first use case is in modern security systems, which are based on Internet of Things (IoT) devices. These devices operate on the edge, where there is very limited computation power, so using AI is almost impossible. System providers can, at this moment, generate the specific distribution of Debezium Server that will have a significantly smaller memory footprint than a normal distribution. This distribution can then stream changes from all the sensors into an external data center where all the AI computation takes place. Let's visualize this scenario on the example displayed in Figure 7.1.

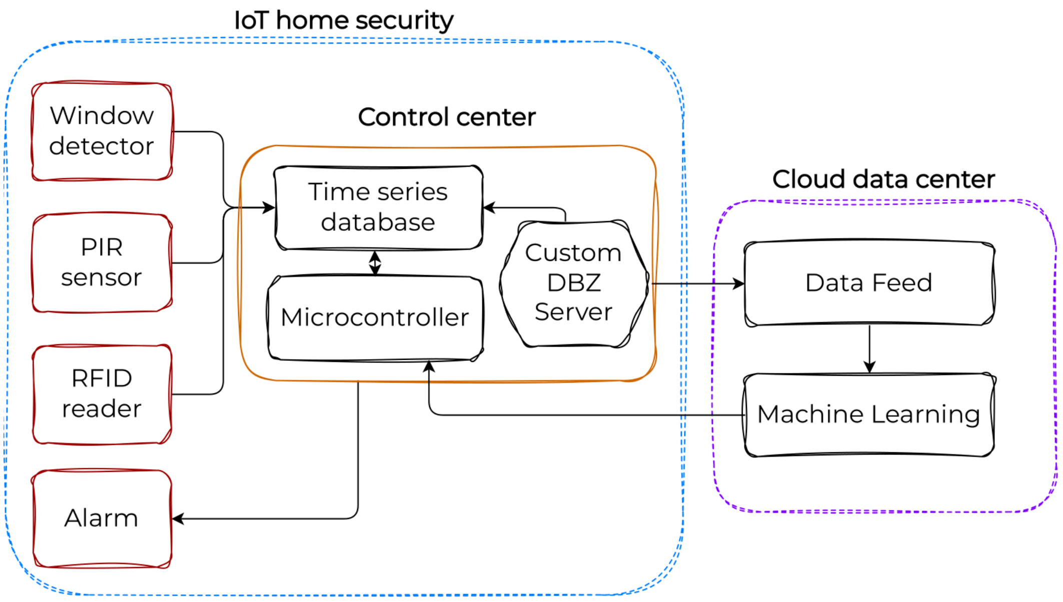


Fig. 7.1 Application of Debezium Server distribution in IoT security systems.

Figure 7.1 showcases possible use cases in simple home security systems. You can see that the security is composed of several inputs – window detector, PIR detector, and RFID reader. These inputs store all the changes in some time series database. This could be, for example, *TimescaleDB* [1]. The database is connected to the microcontroller. Together, these two components create the brain of the whole security system. Besides that, the system also contains custom Debezium Server distribution. This can

[1]TimescaleDB is an extension to the casual PostgreSQL database.

run on the microcontroller or some other chip. Debezium Server now streams all the data from the database to the external data center.

The data center can contain some data feed systems (for example, Redis) and machine learning computations connected to this feed. Machine learning will progressively learn the user's routines and reactions to specific events. Once the machine learning identifies the bare minimum of the relations tree, it can give specific feedback to the microcontroller. This feedback varies depending on what sensors are used and where they are located. An example might be the detection of false positive intruder detection.

For example, if the homeowner arrives home with the children. He opens the garage, but before he can unarm the security system, children will run to the garden and trigger PIR detector. At this point, the security system triggers the alarm. The ML will detect this series of events and notify the controller to deactivate the alarm because, from previous situations, the ML knows that this happens quite often and that there is no risk in doing so. This significantly improves the user experience of such a system. This situation was one of many that could be mitigated using the custom distribution of the Debezium Server and machine learning together.

Of course, this is just a simple deployment. If the object is bigger and has multiple different technologies on board, the provider can always plug more specific distribution of the server for each and stream the events independently on each other to the data feed. Everything at this point is in the hands of the provider as the Distribution Builder allows building the distribution for a wide variety of Debezium Server sinks and sources.

The second use case that was considered essential during the implementation of this project was to make the quantitative risk analysis more commonly used. These days, security experts have a lot of statistical data that can be used for much more accurate risk analysis, like unemployment statistics or maps of criminal activities. All of these sources are valuable assets for the risk analysis. However, these data sets have to be processed. The system has to create something like a weighted sum to combine them into meaningful results. Doing this manually for each analysis is complicated, and that is why most experts don't take this path.

At this point, custom Debezium Server distribution entries are in the field. Debezium Server can collect data from all the sources and push these into the Machine Learning data feed. The machine learning can then process all the data and identify the risk for the specific location. Let's display this situation using simple schematics.

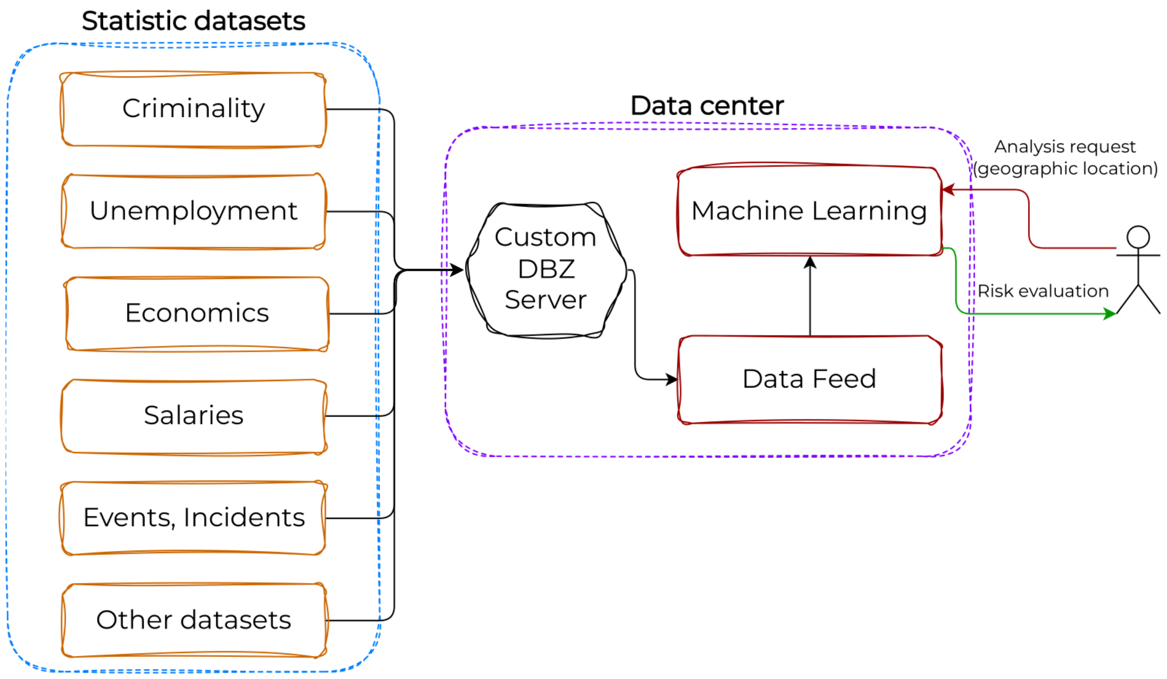


Fig. 7.2 Application of Debezium Server distribution for quantitative risk analysis.

Figure 7.2 displays possible deployment for such a use case. You can see that there are statistical datasets. Depending on how these datasets are stored, it may be necessary to transform them into different storage. Most commonly, these sets are stored in SQL databases so they can be effectively queried. Then, there is a data center that hosts all computation processes, including custom Debezium Server distribution. During the start-up, the Debezium Server creates a snapshot of the data sources, which essentially migrates all the data to the machine learning feed. Once the snapshot is done, the Debezium Server starts to stream all the changes on the datasets so the feed is always up to date.

Machine learning takes all the data from the feed and processes it according to its model. Once the machine learning establishes the base relationships, users can request the ML create an analysis for a specific geographic location. Based on the prepared model, it processes the location and returns a risk evaluation.

This Section went through the possible use cases for custom distributions of Debezium Server in security systems. All these use cases are just theories, and they would need a lot of work to turn them into reality. This thesis does not aim to explain or deep dive into the machine learning principles of artificial intelligence in general, so it uses just a generic description. Not all terms are exactly precise from the AI point of view, but they are not necessary for this thesis. These use cases might not be the only ones, but they are probably the most important.

CONCLUSION

The main aim of this thesis was to design and implement a system that would allow the building of custom distributions for the Debezium Server. In order to achieve that, the thesis went through all the necessary technologies that could be used for the implementation. First Chapter I goes through all these technologies, including Java, Maven, Debezium, UI, and Cloud. The main aim of that Chapter was to build a solid knowledge base that would allow efficient implementation.

The next part of the thesis covers the design and implementation. During the design process 5, several use cases were identified that provide a solid understanding of how the public interface should look. In addition to the use cases, the design part also identified sequential diagrams that provide a more profound understanding of how each part of the system interacts with the others.

With the design figured out, the thesis reviewed the obstacles found during the implementation. The most problematic was identifying the correct way to modify the Maven descriptors to achieve the desired distribution. Full implementation is available on GitHub as an Open-Source project under MIT license[2)] and in the appendices. The last part of the implementation covered test suites prepared for the Distribution Builder. This Section went through the unit and integration tests, how they are implemented, and where they are located.

The last part of the thesis discusses arguably the most important topic, the application of the Distribution Builder in security systems 7. This thesis describes two of many possible use cases. The first use case determined in this thesis is an application in modern IoT-based security systems. This application describes how a custom Debezium Server could fit the hardware-limited deployment in the IoT microcontrollers and stream the events from all the sensors to an external data center where machine learning and its feed run. Debezium Server streams all the changes into the feed, and machine learning picks them and learns the routines and everyday issues like false positive alarms. Once the base relationships are established, the machine learning gives feedback back to the microcontroller, which can, based on them, modify its own behavior and responses.

The second and maybe even more critical application is that the custom distribution of the Debezium Server should allow the gathering of crucial data from many statistical

[2)] Library repository available at `https://github.com/obabec/debezium-server-dist-builder`. Backend project available at `https://github.com/obabec/debezium-dist-builder-api`. UI available at `https://github.com/obabec/dist-builder-Angular-ui` data sources and feed those into AI. AI would then (with the proper model) help security experts during risk analysis and make it more precise by moving from qualitative to quantitative analysis.

This thesis results in the fully functional implementation of the Distribution Builder for the Debezium Server. However, the work is not fully complete. This thesis should serve as a baseline for future academic or field projects that would progressively turn the possible use cases into reality.

REFERENCES

[1] *ArgoCD documentation* [online]. 2023. [visited 2024-03-24]. Available at: `https://argo-cd.readthedocs.io/en/stable/`.

[2] BUDD, T. *An Introduction to Object-Oriented Programming.* 2nd ed. Addison-Wesley, 1996. ISBN 978-0201824193.

[3] *What is Cloud-native* [online]. 2024. [visited 2024-03-24]. Available at: `https://cloud.google.com/learn/what-is-cloud-native`.

[4] *Debezium Server* [online]. 2023. [visited 2024-02-25]. Available at: `https://debezium.io/documentation/reference/stable/operations/debezium-server.html`.

[5] FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures.* Irvine, 2000. PhD thesis. University of California. Available at `https://ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf`.

[6] TRANDAFIR, E. *Difference Between Fluent Interface and Builder Pattern in Java* [online]. 2024. [visited 2024-03-24]. Available at: `https://www.baeldung.com/java-fluent-interface-vs-builder-pattern`.

[7] FOWLER, M. *Domain Specific Languages.* 1st ed. Addison-Wesley Professional, 2010. ISBN 978-0-321-71294-3.

[8] GAMMA, E. *Design patterns : elements of reusable object-oriented software.* 1st ed. Boston: Addison-Wesley, 1995. Addison-Wesley professional computing series. ISBN 0-201-63361-2.

[9] *What is GitOps* [online]. 2023. [visited 2024-03-24]. Available at: `https://www.redhat.com/en/topics/devops/what-is-gitops`.

[10] *Gradle vs Maven: Performance Comparison* [online]. 2024. [visited 2024-02-25]. Available at: `https://gradle.org/gradle-vs-maven-performance`.

[11] *The Java® Language Specification* [online]. 2021. [visited 2024-02-25]. Available at: `https://docs.oracle.com/javase/specs/jls/se17/jls17.pdf`.

[12] *JUnit vs TestNG: Which Testing Framework Should You Choose?* [online]. 2024. [visited 2024-03-24]. Available at: `https://https://www.harness.io/blog/junit-vs-testng`.

[13] *Kubernetes | CNCF* [online]. 2024. [visited 2024-03-24]. Available at: `https://www.cncf.io/projects/kubernetes/`.

[14] Larrucea, X., Santamaria, I., Colomo Palacios, R. and Ebert, C. Microservices. *IEEE Software.* 2018, vol. 35, no. 3, p. 96–100. DOI: 10.1109/MS.2018.2141030.

[15] M, A., Dinkar, A., Mouli, S. C., B, S. and Deshpande, A. A. Comparison of Containerization and Virtualization in Cloud Architectures. In: *2021 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT).* 2021, p. 1–5. DOI: 10.1109/CONECCT52877.2021.9622668.

[16] *Apache Maven Assembly Plugin* [online]. 2023. [visited 2024-02-25]. Available at: `https://maven.apache.org/plugins/maven-assembly-plugin/`.

[17] *Micronaut* [online]. 2024. [visited 2024-03-24]. Available at: `https://docs.micronaut.io/index.html`.

[18] Rouse, M. *Micronaut framework* [online]. Techtarget, 2021. [visited 2024-03-24]. Available at: `https://www.techtarget.com/searchapparchitecture/definition/Micronaut-framework`.

[19] Murray, N., Coury, F., Lerner, A., and Taborda, C. *Ng-book.* 1st ed. Fullstack.io, 2020. ISBN 978-1985170285.

[20] Narkhede, N., Shapira, G. and Palino, T. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale.* 1st ed. O'Reilly Media, 2017. ISBN 978-1491936160.

[21] Noback, M. *Principles of Package Design: Creating Reusable Software Components.* 1st ed. Apress, 2018. ISBN 978-1484241189.

[22] Poulton, N. and Joglekar, P. *The Kubernetes Book.* 1st ed. Nigel Poulton and Pushkar Joglekar, 2023. ISBN 978-1916585003.

[23] *ReactJS* [online]. 2024. [visited 2024-03-24]. Available at: `https://legacy.reactjs.org/`.

[24] *Spring* [online]. 2024. [visited 2024-03-24]. Available at: `https://spring.io/`.

[25] *Sundrio: Code generation toolkit for Java* [online]. 2023. [visited 2024-03-24]. Available at: `https://github.com/sundrio/sundrio/blob/main/readme.md`.

[26] Turnbull, J. *The Docker Book.* 1st ed. James Turnbull, 2018. ISBN 978-0988820234.

[27] *TypeScript Documentation* [online]. 2024. [visited 2024-03-24]. Available at: `https://www.typescriptlang.org/docs/`.

LIST OF ABBREVIATIONS

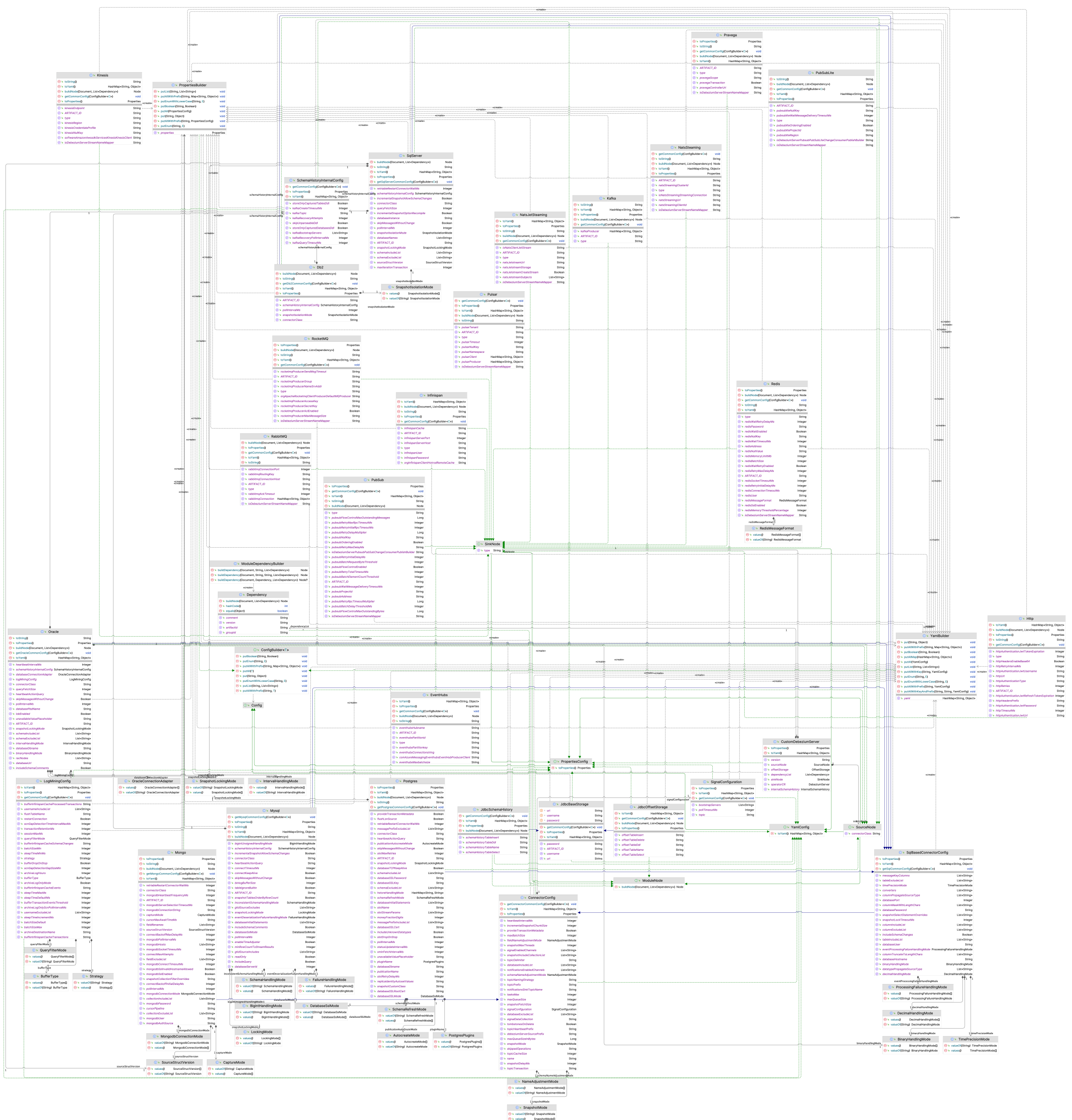
CDC	Change Data Capture
OOP	Object Oriented Paradigm
JVM	Java Virtual Machine
JIT	Just In Time
JRE	Java Runtime Edition
JDK	Java Development Kit
JS	Java Script
TS	Type Script
KC	Kafka Connect
SMT	Single Message Transformation
AoT	Ahead of Time
IoT	Internet of Things
JPA	Java Persistent API
EE	Enterprise Edition
MVC	Model View Controller
ORM	Object-Relational Mapping
JAR	Java Archive
JMX	Java Management Extensions
UI	User Interface
DOM	Document Object Model
VCS	Version Control System
CNFC	Cloud Native Foundation
CD	Continuous Delivery
MVN	Maven
DBZ	Debezium
ML	Machine learning
AI	Artificial Intelligence

LIST OF FIGURES

LIST OF LISTINGS

LIST OF APPENDICES

APPENDIX P I. CLASS DIAGRAM FOR THE LIBRARY IMPLEMENTATION



APPENDIX P II. BASE MAVEN DESCRIPTOR FOR THE LIBRARY

```xml

<?xml version="1.0"?>
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://maven.apache.org/POM/4.0.0"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <groupId>io.debezium</groupId>
        <artifactId>debezium-server</artifactId>
    </parent>

    <modelVersion>4.0.0</modelVersion>
    <artifactId>debezium-server-custom-dist</artifactId>
    <name>Custom Debezium Server Distribution</name>
    <packaging>jar</packaging>

    <properties>
        <assembly.descriptor>server-distribution</assembly.descriptor>
        <quarkus.package.type>legacy-jar</quarkus.package.type>
        <version.debezium>${project.parent.version}</version.debezium>
        <version.quarkus>3.2.9.Final</version.quarkus>
    </properties>

    <build>
        <plugins>
            <plugin>
                <groupId>io.quarkus</groupId>
                <artifactId>quarkus-maven-plugin</artifactId>
                <version>${quarkus.version.runtime}</version>
                <configuration>
                    <systemProperties>
                        <quarkus.kubernetes-config.secrets.enabled>true</quarkus.kubernetes-config.secrets.enabled>
                        <!-- Required in order to support SRV protocol in MongoDB connector (due to java drive) -->
                        <quarkus.naming.enable-jndi>true</quarkus.naming.enable-jndi>
                    </systemProperties>
                </configuration>
                <executions>
                    <execution>
                        <goals>
                            <goal>build</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>

    <profiles>
        <profile>
            <id>assembly</id>
            <activation>
                <activeByDefault>false</activeByDefault>
            </activation>
            <dependencies>
                <dependency>
                    <groupId>io.debezium</groupId>
                    <artifactId>debezium-scripting</artifactId>
                    <version>${version.debezium}</version>
                </dependency>
                <dependency>
                    <groupId>io.debezium</groupId>
                    <artifactId>debezium-scripting-languages</
                       artifactId>
                    <version>${version.debezium}</version>
                    <type>pom</type>
                </dependency>
                <dependency>
                    <groupId>io.debezium</groupId>
                    <artifactId>debezium-server-core</artifactId>
                    <version>${version.debezium}</version>
                </dependency>
                <dependency>
                    <groupId>io.debezium</groupId>
                    <artifactId>debezium-core</artifactId>
                    <version>${version.debezium}</version>
                </dependency>
                <dependency>
                    <groupId>io.quarkus</groupId>
                    <artifactId>quarkus-logging-json</artifactId>
                    <version>${version.quarkus}</version>
                </dependency>
            </dependencies>
            <build>
                <plugins>
                    <plugin>
                        <groupId>org.apache.maven.plugins</groupId>
                        <artifactId>maven-assembly-plugin</artifactId>
                        <version>${version.assembly.plugin}</version>
                        <executions>
                            <execution>
                                <id>default</id>
                                <phase>package</phase>
                                <goals>
                                    <goal>single</goal>
                                </goals>
                                <configuration>
                                    <appendAssemblyId>false</
```

```xml
                                        appendAssemblyId>
                                    <attach>true</attach>  <!-- we want attach & deploy these to Maven -->
                                    <descriptors>
                                        <descriptor>src/main/resources/assemblies/${assembly.descriptor}.xml</descriptor>
                                    </descriptors>
                                    <tarLongFileMode>posix</tarLongFileMode>
                                </configuration>
                            </execution>
                        </executions>
                    </plugin>
                </plugins>
            </build>
        </profile>
    </profiles>
</project>
```

APPENDIX P III. CONTENTS OF CD

```
+-distribution-builder-sources
| +--library
| +--api
| \--ui
+-thesis-sources
+-demo.mp4
```