Issue October 2019 | presented by JAXenter www.jaxenter.com #70

jaxmagazine

The digital magazine for enterprise developers

Java 13

The JDK's hidden treasures

Jakarta EE 8
Let the games begin

JDK 13
Why text blocks are worth the wait

OpenJFX 13
JavaFX gets its own identity

© Teguh Mujiono/Shutterstock.com, Pushkin/Shutterstock.com
Illustrationen: Sun Microsystems Inc., S&S Media

Let's celebrate Java – three times!

It's that time again: A new Java version is here! Java 13 was launched as planned, six months after the release of Java 12, and again it has some interesting features on board. In this issue of Jax Magazine, we've covered them for you in detail.

The good news doesn't end there, as JavaFX 13 has also been released. The UI toolkit is no longer included in the JDK but has adjusted its new version releases to the new Java release cadence. Find out what's new here!

Last but not least: Jakarta EE, the follow-up project of Java EE, has announced its first release under the umbrella of the Eclipse Foundation. We got hold of the executive director of the Eclipse Foundation, Mike Milinkovich, and asked him about the current status of Jakarta EE.

Happy reading,

Hartmut Schlosser

Index

© Tegüh Mujiono/Shutterstock.com, Pushkin/Shutterstock.com
Illustrationen: Sun Microsystems Inc., S&S Media

Every half a year, new Java is here!

Java 13 – a deep dive into the JDK's new features

Java expert Falk Sippach celebrates the release of Java 13 in this article talking about what's new, why it matters and what it means for the future of the programming language.

by Falk Sippach

You can tell this year sped by in a flash because Java's version number has already increased by two! Meanwhile, we Java developers should have got used to the short release cycles. After all, we can now regularly try out new functions and are not killed every few years by a huge range of new features. In this article we take a look at Java 13, which is released today, September 17th, 2019.

Java 13 at a glance

The following Java Enhancement Proposals have made it into JDK 13:

- JEP 350: Dynamic CDS Archives
- JEP 351: ZGC: Uncommit Unused Memory
- JEP 353: Reimplement the Legacy Socket API
- JEP 354: Switch Expressions (Preview)
- JEP 355: Text Blocks (Preview)

At first glance, it doesn't look like much. In fact, due to the short time since the release of Java 12, we can't really expect too many changes. Instead, the releases between the Long-Term Support (LTS) versions are mostly there to offer certain features as previews in order to get early feedback from the users. The functions are implemented in the JEPs and – as soon as they have reached a certain maturity – delivered with the next release of the defined half-yearly cycle.

This makes it impossible to predict exactly how many new functions relevant to the typical Java programmer will be included in the next release. The goal is to finalize the preview features by the next LTS version so that they are stable

enough and will look good for the next three years. In September 2021, Java 17 will take over the legacy of Java 8 and 11.

Enhancements for Switch Expressions

If you look at the feature list above from a developer's point of view, it's the last two points that are mainly interesting. For example, the Switch Expressions introduced as a preview in Java 12 have been extended due to user feedback. The Switch Expression is an alternative to the cumbersome and error-prone Switch Statement. A detailed overview of its use can be found in various articles on Java 12 [1].

The biggest change in Java 13 is the replacement of the keyword break in the switch expression by yield. The background is the better differentiation between switch statement (with possible break) and expressions (with yield). The yield statement exits the switch and returns the result of the current branch, similar to a return.

A code example follows in Listing 1. Above, we see a statement with break and fall through. In the direct comparison follows a switch expression with the new keyword yield and multiple labels. In contrast to the first variant, no variable is changed, but the result of the case branch is returned directly.

The Arrow syntax introduced in Java 12 still works (Listing 2).

Switch Expressions remain in preview for the time being, so there could be further adjustments in future Java versions. When compiling with JDK 13, the corresponding flags must therefore be specified:

```
javac --release 13 --enable-preview Examples.java
```

The preview feature must also be activated at startup. Of course, build tools (Maven, etc.) also have configuration switches:

```
java --enable-preview Examples
```

Text Blocks instead of Raw String Literals

Text blocks are actually only a small part of the Raw String Literals (JEP 326) originally planned for Java 12. The first implementation of Raw String Literals was not yet thought through down to the last detail, users' feedback raised many questions. The exact details can be found in the mailing list [2]. Java 13 "only" has multi-line text blocks for now.

But that's better than nothing. After all, many Java applications process code snippets from other languages such as HTML or SQL, which usually consist of several lines for the sake of clarity. Until now, such strings could only be defined in a very cumbersome way, which makes them difficult to read. For example, extra control commands (escaping with *\n*) must be used for line breaks. Other languages such as Groovy, Scala or Kotlin have long offered the possibility of defining multi-line texts.

The new text blocks use triple quotation marks as delimiters and can be used wherever normal strings are allowed. Listing 3 shows the differences between traditional and new syntax. The opening and closing triple quotes must be in a separate line. Nevertheless, the actual content starts with the second line. This increases the readability of the source code, as the indentation of the first line is displayed correctly in the source text.

Further examples can be found in Tim Zöller's article in this JaxMag, which deals specifically with JEP 355. Text blocks are replaced by normal strings at compile time; in the byte code, you can no longer see how the string was originally defined.

Dynamic CDS Archives

Besides the new features that are obvious to developers, a lot has happened under the hood of the JVM and in the class library. Class Data Sharing (CDS) was introduced back in Java 5. The goal of CDS is to shorten the start times of Java applications by storing certain information about classes in Class Data Sharing archives. This data can then be loaded at runtime and used by several JVMs.

Until Java 10, however, the shared archives were only accessible for the Bootstrap Class Loader. Starting with Java 10, CDS was extended by Application Class Data Sharing (AppCDS).

Listing 1

```
// Switch Statement with break
private static String statementBreak(int switchArg){
  String str = "not set";
  switch (switchArg){
    case 1:
    case 2:
      str = "one or two";
      break;
    case 3:
      str = "three";
      break;
  };
  return str;
}

// Switch Expression with yield
private static String expressionBreakWithValue(int switchArg){
  return switch (switchArg){
    case 1, 2: yield "one or two";
    case 3: yield "three";
    default: yield "smaller than one or bigger than three";
  };
}
```

Listing 2

```
private static String expressionWithArrow(int i) {
  return switch (i) {
    case 1, 2 -> "one or two";
    case 3 -> "three";
    default -> "smaller than one or more than three";
  };
}
```

Listing 3

```
// Without Text Blocks
String html = "<html>\n" +
          "   <body>\n" +
          "      <p>Hello, Escapes</p>\n" +
          "   </body>\n" +
          "</html>\n";

// With Text Blocks
String html = """
          <html>
             <body>
             <p>Hello, Text Blocks</p>
             </body>
          </html>""";
```

AppCDS enables the built-in system and platform class loader as well as user-defined class loaders to access the CDS archives. Class lists are required to create the CDS archives in order to identify the classes to be loaded.

Previously, these class lists had to be determined by trial runs of the application to determine which classes were actually loaded during execution. Since Java 12, default CDS archives are delivered with the JDK by default, which are based on the class list of the JDK.

Dynamic CDS Archives now build on this. The goal is to save the additional test runs of the application. After an application has been executed, only the newly loaded application and library classes that are not already contained in the Default/Base Layer CDS are archived. Dynamic archiving is activated with command line commands. In a future extension, the archiving of classes could then run completely automatically and transparently.

ZGC returns unused storage

Nowadays it is not uncommon that applications have to serve many thousands of users at the same time. These applications need a lot of memory, the management of memory is not trivial and affects the performance of the application. To meet these requirements, Oracle introduced the Z Garbage Collector (ZGC) in Java 11. It promises very short pauses when cleaning up heap memories with several terabytes.

Previously, however, it did not release the heap memory reserved for the application. As a result, applications usually consume far more memory than necessary over their lifetime. Applications that run in low-resource environments are particularly affected. Other garbage collectors such as the G1 and Shenandoah already support the release of unused memory.

Renewed Socket APIs

The java.net.Socket and java.net.ServerSocket APIs and their underlying implementations are remnants from the JDK 1.0. Most of them consist of legacy Java and C code. This makes maintenance and extensibility much more difficult. The NioSocketImpl is supposed to replace the outdated PlainSocketImpl now. NioSocketImpl is based on the already existing New I/O implementation and uses its existing infrastructure in the JDK.

The previous implementation is also not compatible with other planned extensions of the language. For example, concurrency problems hinder the future use of the lightweight user threads (fiber, part of the Project Loom).

Conclusion

The changes described so far were defined in the JEPs (Java Enhancement Proposals). There are, however, further adaptations directly in the class library. These can be visualized using either the JDK API Diff Report Generator [4] or the Java Almanac [5]. A closer look reveals that the old Doclet API has been removed from com.sun.javadoc [6]. Also, three new instance methods have been added to the String class as part of JEP 355 (Text Block): formatted, stripIndent and translateEscapes.

Since Java 10 we can look forward to new Java releases twice a year (March and September). Since of course not so much can happen in development in half a year, the respective feature lists are fortunately manageable. However, the short update cycles allow us to regularly try out the new features at short intervals. And the Java team at Oracle gets faster and more valuable feedback on the new features, some of which are in preview status for one or more releases. On the way to the next LTS version there will be user feedback and changes if necessary. For example, in the current LTS release Java 11 the work started in Java 9 and 10 was finalized.

For the productive use of its Java applications one can meanwhile choose between two variants. Either you stay with the current LTS release for three years or you update twice a year to the major version of the OpenJDK, which is updated by Oracle for half a year. The latter may be used freely, but must be updated to the next version after six months in order to continue to receive security updates and patches. With the LTS version, there are both paid offers (Oracle, Azul, …) and free installations such as AdoptOpenJDK, Amazon Corretto, Red Hat, SAP and Alibaba Dragonwell. After the first surprise about Oracle's new licensing policy, there are now a lot of alternatives to run Java in production.

In addition, Oracle's idea seems to work with the semi-annual major releases and regular, albeit small, updates. So far the versions have been delivered on time as announced, so we can already look forward to the next changes. But first we have to install JDK 13 [7] and try out the new features. Work on Java version 14 [8] is already running in parallel. At the time this article was written, the JEP 352 (Non-volatile Mapped Byte Buffers) was already planned. In addition, the previews of switch expressions and text blocks may be finalized. Furthermore, a new tool for packaging self-contained Java applications could appear within the framework of JEP 343. But as uncertain as these changes may seem at this point in time, the next release is already on its way. Because in March 2020, we'll get to say, "every half year" again …

Falk Sippach has over fifteen years of experience in the Java and JavaScript world and works as a trainer, software developer and project manager at OIO Orientation in Objects GmbH. When he has time, he also writes blog entries and articles, and occasionally appears at conferences. In his adopted home Darmstadt he organizes the local Java User Group together with others.

References

[1] https://jaxenter.com/java-12-is-here-156964.html

[2] https://mail.openjdk.java.net/pipermail/jdk-dev/2018-December/002402.html

[3] https://github.com/AdoptOpenJDK/jdk-api-diff

[4] http://download.eclipselab.org/jdkdiff/V12/V13/index.html

[5] https://bugs.openjdk.java.net/browse/JDK-8215584

[6] https://jdk.java.net/13/

[7] https://openjdk.java.net/projects/jdk/14/

At long last

Java 13 – why text blocks are worth the wait

For a long time, Java developers had to take unattractive detours when formulating multi-line string literals, while enviously switching to Scala, Kotlin, Groovy and other languages. With JEP 355, text blocks are now being previewed in Java 13.

by Tim Zöller

Writing string literals in Java code over multiple lines and possibly formatted with whitespace is annoying for many developers. For example, to format an SQL statement in code clearly over several lines, it is common to use the + operator with manually inserted line breaks at the end (Listing 1).

To see that most languages already contain the ability to properly format string literals, just look at Groovy, Scala and Kotlin. Other higher programming languages like C#, Swift or Python are in no way inferior. With such a high number of models, it is possible to adapt the best properties of the implementations for Java. This can be seen from JEP 326 for raw string literals, originally intended for Java 12. The community had many comments and objections, whereupon the proposal was withdrawn [1].

JEP 355 has taken many of the comments into account and now allows developers starting with Java 13 to easily define multi-line strings in code. The feature is first introduced in JDK 13 as a preview – so if you want to use it, you have to compile your code with the *--enable-preview* flag.

JEP 326: Raw Strings

In order to better classify some of the design decisions for text blocks, it is worth taking a look at the withdrawn JEP 326 [2]. This had the JDK 12 as its goal and intended to introduce raw strings in Java – strings that can extend over several lines and do not interpret escape sequences.

In order to be as flexible as possible and to be able to function independently of the contained text, an arbitrary number of backquotes (`) was suggested as the boundary sequence. If the string itself contains a backquote, the boundary sequence would have to contain at least two. If the sequence contains two consecutive backquotes, the boundary sequence would have at least three, and so on.

In the announcement that the JEP will be withdrawn, Brian Goetz mentions some criticisms from the community that led to this decision. Many developers feared that the variable number of characters in the boundary sequence could be confusing for people and development environments. There has also been criticism that the backquote introduces one of the few unused delimiters into code that would be "used up" and could not be used for future features.

Additionally, the fact that any multi-line string literal with the proposed feature would automatically have to be a raw string gave rise to the decision to withdraw JEP 236. The approaches contained in JEP 355 explicitly build on the findings of this process.

Listing 1

```
String statement = "SELECT\n" +
                   "   c.id,\n" +
                   "   c.firstname,\n" +
                   "   c.lastName,\n" +
                   "   c.customernumber\n" +
                   "FROM customers c;";
```

Delimiters and Formatting

Multi-line string literals are enclosed by a sequence of three quotation marks (“””). The opening character sequence may only

be optionally followed by whitespaces and a mandatory line break. The actual content of the literal starts in the next line of code (Listing 2). Single and double quotation marks may occur in the literal without escaping them. If the literal itself contains three quotation marks, at least one of them must be escaped.

Multi-line string literals are processed at compile time. The compiler always proceeds according to the same schema: All line breaks in the character string are first converted into a line feed (\u000A) independent of the operating system. Sequences explicitly entered with escape characters in the text such as \n or \r are excluded from this. In the second step, the whitespace owed to the code formatting is removed. It is marked with the position of the concluding three quotation marks. This allows you to place text blocks in the code so that it matches the rest of the code formatting (Listing 3).

For Scala and Kotlin, multiline literals must either be written left-aligned without whitespace in the source text, or they must be freed from whitespace by string manipulation. Java, on the other hand, is strongly based on Swift's procedure, a cleanup of the strings at runtime is not necessary. Finally, all escape sequences in the text are interpreted and resolved. After compiling, it is no longer possible to find out how a string was defined in the code, whether it was defined as a multi-line string or not.

Use cases

With the new possibilities that text blocks offer developers, code can be written cleaner and/or more readable in some cases. As mentioned, it is now possible, for example, to display SQL statements in Java code more clearly. Since text blocks can be used everywhere conventional string literals are allowed, this even applies to named queries with JQL or native SQL (Listing 4), which are defined within annotations.

Listing 2

```
String correctString = """ // First character only in next line, correct
                    {
                      "type": "json",
                      "content": "sampletext"
                    }
                    """;

String incorrectString = """{ // Characters after delimiter sequence, incorrect
                        "type": "json",
                        "content": "sampletext"
                      }
                     """;
```

Listing 3

```
String string1 = """
                {
                  "type": "json",
                  "content": "sampletext"
                }
                """;

// Content of string1.
// The left margin is marked by | for illustration purposes.
//|{
//|    "type": "json",
//|    "content": "sampletext"
//|}

String string2 = """
                {
                  "type": "json",
                  "content": "sampletext"
                }
           """;

// Content of string2.
// The left margin is marked by | for illustration purposes.
//|   {
//|      "type": "json",
//|    "content": "sampletext"
//|   }
```

Listing 4

```
@Entity
@NamedQuery(name = "findByCustomerNumberAndCity",
          query = """
                from Customer c
                where c.customerNo = :customerNo
                and c.city = :city
                """)
public class CustomerEntity {

  String customerNo;
  String city;

  public String getCustomerNo() {
    return customerNo;
  }

  public void setCustomerNo(String customerNo) {
    this.customerNo = customerNo;
  }
}
```

Listing 5

```
String htmlContent = """                          <li>Another entry</li>
              <div>                               </ul>
                <h2>My header</h2>              </div>
                <ul>                             """;
                  <li>An entry</li>
```

Further useful applications can be found if you want to use string literals as templates. This can be useful either for JSON Payloads in unit tests that you want to use to test a REST service, or for preparing HTML snippets on the server side (Listing 5).

In addition, text blocks significantly simplify the use of polyglot features, be it with the old rhino engine or with the context of GraalVM. The JavaScript code defined in the literal would be much less readable and maintainable if the construct mentioned at the beginning had to be used with string concatenation and manual line breaks (Listing 6).

Listing 6

```
ScriptEngine engine = new ScriptEngineManager().getEngineByName("js");
Object result = engine.eval("""
                    function add(int1, int2) {
                    return int1 + int2;
                    }
                    add(1, 2);""");
```

Listing 7

```
String.format(„Hallo, %s, %s dich zu sehen!“, „Duke“, „schön“);

String xmlString = """
            <customer>
              <no>%s</no>
              <name>%s</name>
            </customer>
            """.formatted("12345", "Franz Kunde");
```

Listing 8

```
// The left margin is marked by | for illustration purposes.
String example = " a\n  b\\n   c";

System.out.println(example);
//| a
//|  b\n   c

System.out.println(example.stripIndent());
//|a
//| b\n   c

System.out.println(example.translateEscapes());
//| a
//|  b
//|   c

System.out.println(example.stripIndent().translateEscapes());
//|a
//| b
//|  c
```

New methods

JEP 355 adds three new instance methods to the String class: *formatted*, *stripIndent*, and *translateEscapes*. While formatted is an auxiliary method that makes working with multiline literals easier, *stripIndent* and *translateEscapes* represent the last two compile steps in their interpretation. As mentioned, multi-line strings are excellent for defining templates in code. To fill placeholders in string templates with values, the static method format already exists in the String class. Since multi-line defined string literals do not differ in the use of single-line literals, they can of course also be formatted with this method.

In order to provide a clearer approach for text blocks, the new instance method *formatted* has been added to the String class, which behaves analogously to the static method formatted. The result is also a string formatted with placeholders, but in combination with text blocks it results in code that is tidier (Listing 7).

The *stripIndent* method removes whitespace in front of multi-line strings that all lines have in common, i.e. moves the entire text to the left without changing the formatting. The method translateEscapes interprets all escape sequences contained in a string. Listing 8 clarifies this behavior once again.

Conclusion

With JEP 355, multi-line string literals are introduced in a way that quickly feels natural both in their declaration and in their formatting. The developers have benefited from the experience gained in other programming languages and have chosen a way that makes post-processing of strings, for example to remove annoying unnecessary whitespace and at the same time allowing a readable formatting of source code. The introduction is rounded off by three new methods in the String class: formatted, stripIndent and translateEscapes. Not only do they make the new text blocks easier to use, but they can also be used to format strings from other sources, remove indentations, or interpret escape sequences. Personally, I'm very much looking forward to the introduction of text blocks as a fully rounded out feature and notice almost daily how much I'm missing them at the moment.

Tim Zöller has been working with Java for ten years, and for fun he sometimes develops software in Clojure. He works as an IT consultant and software developer at ilum:e informatik ag and is co-founder of JUG Mainz.

References

[1] https://mail.openjdk.java.net/pipermail/jdk-dev/2018-December/002402.html

[2] https://openjdk.java.net/jeps/326

On your marks, get set, go!

Jakarta EE 8 is sprinting towards an exciting future for enterprise Java

After what seemed like eternity, the time has finally come – the long awaited first release of Jakarta EE is now officially in the hands of the public. Now the community can start hoping for faster development of the platform.

by Thilo Frotscher

The first release is called "Jakarta EE 8" and was released on September 10th. Jakarta EE 8 – as you can probably tell from the name – is functionally identical to Java EE 8, which was published two years ago. So, you might think that not much has happened in the meantime, but in fact the move of the Java EE platform to the Eclipse Foundation involved considerable effort; large amounts of code had to be migrated to a new build infrastructure. This includes the previous reference implementation "Glassfish" as well as the TCKs (Technology Compatibility Kits) which are now available as open source. Allegedly, these are a total of about 5.5 million lines of code and over 2.2 million comment lines in more than 61,000 files. This would be comparable with the backend of World of Warcraft and Linux kernel 2.6.0. It is therefore not a success to be underestimated if, with the release of Eclipse Glassfish 5.1, an application server compatible with Java EE 8 could be built on the infrastructure of the Eclipse Foundation and its compatibility could be proven with the help of the TCKs.

But that's not all. In addition, difficult legal negotiations were conducted behind the scenes, and trademark and usage rights in particular were negotiated. The specification documents of the individual Java EE technologies were transferred to the Eclipse Foundation, and a new specification process called EFSP (Eclipse Foundation Specification Process) was passed, which will replace the JCP in the future. The EFSP stipulates, among other things, that there should no longer be so-called Spec Leads, that new features should be developed with a "code first" approach instead of starting with the specification document, and that there should be "several compatible implementations" instead of a single reference implementation.

Over the course of the transfer to the Eclipse Foundation, the individual Java EE technologies also had to be renamed. Therefore it is no longer called "Java Servlet", but "Jakarta Servlet" and so on. While this renaming may be perceived as annoying here and there, it also has a positive aspect: time was spent ensuring the names of the technologies were standardized and also simplified in part. Until now, the names were quite inconsistent. Where previously we had such inconsistencies as "Java Architecture for XML Binding", but "Java API for JSON Binding", the new names in this particular case are now "Jakarta XML Binding" and "Jakarta JSON Binding". The recognition value of the new names can be rated as very good overall.

What next for Jakarta EE?

Now that the first release of Jakarta EE is available, the long-awaited further development of the platform can finally begin. Many developers hope for a general modernization and new features, especially for the development of microservices and for modern operating variants, such as cloud or Kubernetes.

Within the MicroProfile project, a number of interesting suggestions have already been developed, most of which already show good stability. After all, MicroProfile version 3.0 is already available, and various implementations are available for developers to download. These new features include support for metrics, heath checks and OpenTracing, support for OpenAPI, and extensions to improve the robustness of applications (Timeout, Retry, Fallback, Circuit Breaker, Bulkhead). Another useful innovation is the implementation of a Config API, which allows configuration parameters from different sources to be injected directly into the application code. The configuration sources can be a file, environment variables, system properties or proprietary sources such as a database table. The support of such proprietary sources is easily achieved by implementing a Service Provider Interface (SPI). The MicroProfile implementation automatically searches all

available sources for a suitable configuration value when a configuration parameter is injected.

It would therefore be obvious and desirable to migrate MicroProfile's innovations to Jakarta EE step by step. Since MicroProfile's development is already quite advanced, it should be possible to achieve this in a timely manner from a technical point of view. And further interesting innovations can already be found on the MicroProfile roadmap. There you will find support for Reactive Streams, GraphQL, Event Data or reactive access to relational databases. Apart from MicroProfile, however, some new features have also been advanced in some of the existing technologies, such as JAX-RS, which could also soon be incorporated into the platform. In this context it should also be mentioned that Jakarta EE will have a much shorter release cycle. Similar to what happened in Java SE, there will be more frequent releases, which will only bring minor changes. It is to be hoped that a version of Jakarta EE with new features will be released in the near future.

In addition to new features, application development in general and the introduction to the platform should also be simplified. Again MicroProfile offers a possible way; a "MicroProfile Starter" website has already been made available – very similar to that of Spring Boot. It can also be generally observed that many of MicroProfile's concepts are copied from other frameworks. However, this is not necessarily negative: Why shouldn't ideas that have proven themselves elsewhere be adopted? On the contrary, the recognition value makes it easier to find one's way around in a new (old) environment.

After all, a departure from the original deployment model has long been apparent. The days when monolithic application servers were installed on a powerful server and several Java Enterprise applications were deployed there are definitely a thing of the past. They no longer fit in with current trends such as the cloud or Docker. Over the years, it has become apparent that in many places only a single application is deployed to an application server, which in turn raises the question of why this application server must actually support all Java EE technologies. On the other hand, it would be sufficient if only the technologies actually required by this single deployed application were supported. In view of this development in operations, the manufacturers of application servers have long since modularized their products to such an extent that a custom-fit variant of an application server can be built or configured for each application. The keyword here is "Just Enough App Server". Build plug-ins are also available for Maven, for example, which pack the necessary application server modules together with the application into a single executable JAR file. A model that is much more compatible with Docker and the cloud – and has also been demonstrated by other frameworks.

Despite all modernization and further development, one of the central advantages of the past is to be maintained in the future: It is planned to maintain backward compatibility as far as possible, so that older applications can also run on current Jakarta EE implementations. However, as in the case of Java SE, it is planned to cut off some of the oldest bits and pieces, and to remove individual technologies from the platform in the medium term, or at least to mark them as optional. This should mainly concern APIs, which have always been quite exotic and are only used comparatively rarely.

One last thing ...

So there's a lot to do. Unfortunately, there is one last obstacle to Jakarta EE's development: An extension of the platform would inevitably result in an extension of the existing APIs. Oracle has, however, only approved the use of the existing *javax.enterprise.** packages in an unchanged state. In the event of a change or extension, the brand name "java" may no longer be used. This effectively prevents further development of Jakarta EE while maintaining the current package structure.

Finding a new package name was still quite easy (e.g. "*jakarta.**" and "*jakarta.servlet*"). How exactly the conversion should be done is still under discussion. If the underlying Java APIs are moved to new packages, all existing applications will inevitably become incompatible. So what to do?

First it has to be clarified whether the changeover for all Jakarta EE APIs will take place in a single hard cut, or whether the new package names will be introduced step by step. In each release, only the actually changed APIs could move to new packages. So far nothing has been decided. But if the advocates of a hard cut should prevail, then the next release, "Jakarta EE 9", would probably contain this change to the new packages as the only innovation, while new features would only be expected from "Jakarta EE 10". Fortunately, as already mentioned, much shorter release cycles are planned than usual for Java EE.

One related question is how best to support companies and developers in migrating existing Java EE applications. On the one hand, it would be possible to offer tools or IDE features that automatically rewrite all package dependencies of an existing application project to the new packages. This would still be comparatively easy to implement at code level, but could lead to greater difficulties with dependencies to external libraries. Another suggestion already made is that containers such as Thorntail, Glassfish, or OpenLiberty map old packages to new ones internally at runtime, making them completely transparent to application developers.

Conclusion

With the release of Jakarta EE 8, a very important milestone has been reached for the platform. The problem of package names remains to be solved, then the modernization and the addition of new features can finally start. This is good news for companies that have numerous Java EE-based applications in productive use. But even for the development of new applications, Spring Boot may not be the undisputed top dog in the future. A whole series of exciting new frameworks, such as Quarkus and Micronaut, are on the march in MicroProfile's sphere of influence. It should be an exciting competition.

Thilo Frotscher works as a freelance software architect and trainer. His technical focus is the Java platform as well as services and system integration. He supports companies by participating in development projects and conducting practical training courses.

Let the games begin!

Jakarta EE 8 release – the future is now!

Jakarta EE 8 is finally here! It heralds a new era for Enterprise Java, as finally the community can look forward to the platform's faster development. To celebrate, we sat down with Executive Director of the Eclipse Foundation, Mike Milinkovich to talk about the first release of Java EE under the umbrella of the Eclipse Foundation.

Jax Magazine: Hi and thanks for taking the time for this interview. After many months of hard work, Jakarta EE 8 is finally released. It is feature identical with Java EE 8, but this does not mean that this release is in any way unspectacular. Can you please elaborate on how big the impact of the publication of Jakarta EE 8 really is?
Mike Milinkovich: In a word, huge. It's honestly hard to overstate the impact of a diverse community of vendors, hundreds of dedicated community members, and foundation staff all coming together to deliver the Jakarta EE 8 specifications for the full platform and web profiles, TCKs, and Eclipse GlassFish 5.1 fully certified as a Jakarta EE 8 compatible implementation. All of this was done under the Jakarta EE Specification Process [1], an open and vendor-neutral process that leverages and augments the Eclipse Development Process. That's an incredible accomplishment.

With Jakarta EE 8, the community now has an open source baseline to collaborate on advancing enterprise Java and enabling the migration of real-world workloads to a world of containers, microservices, Kubernetes, service mesh, and other cloud native technologies that have taken off in industry interest and adoption.

JaxMag: Jakarta EE 9 will possibly include some new functionality, but until that is going to happen, another decision has to be made by the community: How to deal with the namespace problem. Will it be the big bang or the incremental approach?
Milinkovich: That's a great question and this is something that is still under active discussion. The entire community was invited to provide input and that conversation is going on as we speak. So far, we have more than 300 responses. Overall, we've seen a slight preference [2] for the Big Bang approach from the community. Regardless of which approach we take, we need to ensure that binary compatibility is taken care of.

JaxMag: Which one would you prefer?
Milinkovich: I think a clean separation and fresh start would be in the community's best interests, so I'm a fan of the Big Bang approach.

JaxMag: When can we expect to have a clear and definitive answer on how to proceed?
Milinkovich: Sometime in the next few months. The community has been heads-down delivering Jakarta EE 8, so after everyone catches their breath, release planning is top of the agenda. We expect all of the vendors in the Jakarta EE Working Group to certify their Java EE 8 compatible implemen-

Portrait

Mike Milinkovich has been involved in the software industry for over thirty years, doing everything from software engineering, to product management to IP licensing. He has been the Executive Director of the Eclipse Foundation since 2004. In that role he is responsible for supporting both the Eclipse open source community and its commercial ecosystem. Prior to joining Eclipse, Mike was a vice president in Oracle's development group. Other stops along the way include WebGain, The Object People, IBM, Object Technology International (OTI) and Nortel.
Mike sits on the Executive Committee of the Java Community Process (JCP), and is a past member of the Boards of the Open Source Initiative (OSI), and the OpenJDK community.

"I have no doubt that we will work on implementing some items from the community's wishlist, but nothing has been set in stone just yet."

tations as Jakarta EE 8 compatible, so there will be a lot of work to be done around that.

JaxMag: With the namespace problem out of the way, a new era of enterprise Java will be open for exploring. Are there any ideas for features yet that will likely be implemented in Jakarta EE 9 and beyond?
Milinkovich: There is certainly a "wishlist" from the community and I have no doubt that we will work on implementing some of these items, but nothing has been set in stone just yet. I can definitely say that things like CDI alignment, modularity, and support for reactive streams are popular. There is also a lot of interest in greater support for microservices and Kubernetes-native deployments.

JaxMag: There is a new Process for determining new specs for Jakarta EE that replaces the Java Community Process (JCP). Can you explain how it is different from the JCP we all know and hated?
Milinkovich: The Jakarta EE 8 specifications were developed under the Jakarta EE Specification Process and Eclipse Development Process, which are open, community-driven processes that replace the JCP for Java EE.

I think that the biggest change is that there is no longer a Spec Lead which has special intellectual property rights under the JCP. Historically, the Spec Lead also usually exerted strong control over the output of the expert group. The Jakarta EE Specification Process is a level playing field in which all parties are equal, and collaboration is a must. Some of the other significant differences include a code-first approach, rather than a focus on specifications as the starting point. You can also expect a more fully open, collaborative approach to generating specifications, with every decision made collectively by the community. All documentation related to the specifications, as well as the TCK will be based on the open source model of community generation. You also won't see reference implementations but rather multiple, compatible product implementations, as well as a process of self-certification, rather than something run by a single organization.

JaxMag: Another thing that is not yet decided is the release schedule for the upcoming Jakarta EE releases. Will it be a regular and continuous process like Java (SE) has now or will it be more like in the olden days, when Java "was done when it was done"?
Milinkovich: As a community, we are still working through the release cycles, so we can't commit to a particular cadence just yet. That said, there is a strong commitment by everyone at the table to significantly increase the pace of innovation. Obviously, we need to strike a balance between factors like stability versus the pace of innovation, but the fact that a diverse community of vendors, developers, and enterprises will be discussing these issues in an open and inclusive manner that will help drive the success of Jakarta EE.

As you know, the broader Eclipse community has a track record of shipping large, complex software releases on time, to the day, for 15 years now. Last year, we moved to a quarterly release train for the Eclipse IDE and the maturity of our processes and our people have continued to deliver. In other words, the Eclipse community and staff have the maturity of process to support whatever release cadence the Jakarta EE community decides is in its best interests.

JaxMag: What are the next steps that have to be taken care of, say, until New Year's Eve?
Milinkovich: We're definitely going to be planning for Jakarta EE 9, as well as helping the entire community transition to our new processes. And that should be plenty to keep us occupied until the New Year. Stay tuned though. We will have plenty to talk about between now and then.

Thank you for the interview!

Interview questions by Dominik Mohilo

References

[1] https://jakarta.ee/about/jesp/

[2] https://github.com/eclipse-ee4j/jakartaee-platform/blob/0c59018e44f9742208fa8514d5f9cd386470b996/namespace/README.adoc

To OpenJFX 13, and beyond!

OpenJFX 13 – "JavaFX gets its own identity"

JavaFX 13 is here! To welcome it to the world, we sat down with Java Champion Dirk Lemmermann to talk about what the new release has in store for developers, what his personal highlights of this new release are and what the future holds for JavaFX.

Jax Magazine: Hello Dirk and thanks for taking the time to talk to us. Can you give a little insight on what JavaFX 13 has in store for developers?
Dirk Lemmermann: OpenJFX 13 brings new features, bug fixes, and better basics for the further development of JavaFX. Highlights include support for native rendering, support for e-paper displays, and upgrading the Webkit engine to version 2.24.

JaxMag: What is your personal highlight of this release and why?
Lemmermann: Among the new features, support for native rendering clearly stands out. With a public API (without tricks and workarounds) it is possible to share memory between the JavaFX application and native applications and thus integrate technologies like OpenGL, D3D or, in the future, Metal (from Apple) into your own applications. Extremely graphic-intensive applications such as CAD systems, 3D editors or games could be implemented with JavaFX.

JaxMag: This is the second release under the new release schedule, what are your thoughts on this? Is the "half year per update" approach reasonable or will the schedule eventually change to a quarterly or yearly release?
Lemmermann: The coupling to the release cycle of the JDK provides structure and predictability on all sides; I consider this to be very important, especially at the moment. There has been a lot of change in JavaFX since Java 8 – APIs have changed, JavaFX has been decoupled from the JDK, and IDEs and projects that want to use JavaFX have to be configured differently. This was not always easy for developers to digest and caused some irritations, resulting in the adoption of JavaFX not being encouraged. A freer release policy could turn out to be another stumbling block.

JaxMag: JavaFX is not part of the official JDK release anymore for quite some time now. Did this have a positive or negative effect?
Lemmermann: I have said from the beginning that I see the decoupling of JavaFX from the JDK as positive. JavaFX gets its own identity, its own processes, its own community, its own pace. Everyone noticed how much the development of JavaFX was slowed down when it was coupled to the JDK development process. Previously, all update releases of Java 8 had new features and real progress. After that it became very tough. Since JavaFX was released again we see progress and the will of the community to advance the toolkit.

JaxMag: Next up is JavaFX 14, any major changes planned for this release or the near future?
Lemmermann: JavaFX 14 will probably be the first release developed completely on GitHub. Work is currently underway to move the repository there (there is currently only a mirror on GitHub). This should simplify all processes and also ensure that more and more developers can participate in the further development of JavaFX. In the past it was not easy to build OpenJFX, that should change soon. As far as new features are concerned, it is still too early to say anything about it. This will become clear as soon as 13 is through the door.

Thanks very much for talking to us!

Interview questions by Dominik Mohilo

Portrait

After his studies in Germany and several placements in the USA (Carnegie Mellon University in Pittsburgh, start-up in Boston), Dirk Lemmermann settled in Switzerland as an independent consultant and developer. The focus of his work is mostly on the development of user interfaces. Currently, this is done using JavaFX, for which he has developed several commercial frameworks. He also writes a JavaFX blog and is a regular comitter in the ControlsFX open source project. Meanwhile, he is also increasingly seen speaking at conferences. Last year he won the Java One Rockstar award.

JavaFX to mission control: All systems are go!

JavaFX 13 release interview with Johan Vos

Jax Magazine: JavaFX 13 has been released. Could you describe the main features of the new version?
Johan Vos: We're continuing the development that we started with the new release cycle. That means we work on bugs and features, and when they are ready to be included in a release, we include them.

JavaFX 13 contains 97 bug fixes and new features, in the different subcomponents. Controls and layout issues are fixed, the WebView and underlying webkit implementation is updated, the build system has been improved, and a new API is added allowing to more easily integrate native content generated by third party applications.

JaxMag: What is your personal highlight of this release and why?
Vos: Probably the most relevant change in JavaFX 13 is JDK-8167148 [1], which adds support for rendering native third-party content. This is a feature that has been requested by JavaFX developers for a very long time. It's a tricky thing to do though, as it is hard to do it right without breaking the cross-platform implementation compatibility of the JavaFX API's. Following the general OpenJDK guidelines, we prefer not to be too intrusive while preserving backward compatibility. On the other hand, it would be less than ideal if developers had to implement platform-dependent code in their own application.

The way it is handled in JavaFX 13 doesn't break any backward compatibility, removes the risk on platform-specific behavior, and provides a good basis for developers to add functionality on top of it.

We had this feature in early-access releases, and it has been discussed intensively on the mailing list and especially in the GitHub issue tracker. Many developers who requested the issue already experimented with the new API.

JaxMag: This is the second release under the new release schedule, what are your thoughts on this? Is the "half year per update" approach reasonable or will the schedule eventually change to a quarterly or yearly release?
Vos: It creates some overhead as we are providing update releases for JavaFX 12 while at the same time providing early access releases for JavaFX 13, and working on JavaFX 14. For Gluon, it's even more complicated as we also support

Portrait

Johan Vos started to work with Java in 1995. He was part of the Blackdown team, porting Java to Linux. His main focus is on end-to-end Java, combining back-end systems and mobile/embedded devices. He received a Duke Choice award in 2014 for his work on JavaFX on mobile.
In 2015, he co-founded Gluon, which allows enterprises to create (mobile) Java Client applications leveraging their existing backend infrastructure. Gluon received a Duke Choice award in 2015.
Johan is a Java Champion, a member of the BeJUG steering group, the Devoxx steering group and he is a JCP member. He is one of the lead authors of the Pro JavaFX books, the author of Quantum Computing for Java Developers, and he has been a speaker at numerous conferences on Java.
He contributes to a number of projects, including OpenJFX, OpenJDK, GraalVM. He is also the project lead for OpenJDK Mobile and the co-lead for OpenJFX.

JavaFX 11, hence we are working on 4 different versions simultaneously.

However, the tools are now in place to automate this as much as possible, including automated testing and building. For OpenJFX developers, this release schedule means less stress, since if their feature doesn't make it for a specific release, there is not much lost as they get a new chance in 6 months.

For the JavaFX developers (the ones using JavaFX APIs), it means smoother updates. If there is 4 years between 2 major revisions, chances are high that companies don't want to upgrade immediately, and the cost of updating can become very high. By having faster releases, it's easier for developers and companies to keep using the latest versions without worrying about huge changes.

From an expectation management point, it's important to stress that we continue working at the same pace as before. We don't want to lower the quality bar in order to rush more features in. But developers understand that in 6 months time, there can be less big changes/features in a release than in 4 years time.

JaxMag: JavaFX is not part of the official JDK release anymore for quite some time now. Did this have a positive or negative effect?

Vos: More and more developers are now very comfortable with the fact that JavaFX is a separate download, either via an SDK or via artifacts in Maven Central. We can see the positive effects already: the native rendering addition has been integrated early in the process in JavaFX 13 Early Access builds, so that we could get feedback from developers.

Download statistics and replies from developers show that those EA builds are very often used, especially via Maven Central. It's indeed very easy for a developer to simply change the version of the JavaFX dependencies in his pom.xml or build.gradle from e.g. 12.0.2 to 13-ea+12. There is no reason for a new manual download just to test this single feature.

JaxMag: Next up is JavaFX 14, any major changes planned for this release or the near future?

Vos: It's really up to the OpenJFX developers to define the shape of the next release. With Gluon, for example, we listen to our customers. If our JavaFX 11 LTS customers want more focus on a specific area, we work on that. After all, these customers pay the engineers working on OpenJFX, so they get a big influence on the roadmap.

One of the things I'm working on is on improving the build system for more cross-platform support. We want to get to the point where we can use 100 percent of the OpenJFX code to build JavaFX for iOS, Android and embedded ARM (32 and 64 bit) devices. We want JavaFX to work on more platforms than the current ones.

In the OpenJDK team, work is being done for supporting Metal. We'll investigate how that work can be leveraged in OpenJFX.

Thanks very much for talking to us!

Interview questions by Dominik Mohilo

References

[1] https://bugs.openjdk.java.net/browse/JDK-8167148

Jeff Sussna discusses the past ten years of DevOps

The state of DevOps

10 years ago, DevOps introduced the idea that IT could be more effective without silos between development and operations. Since then, "cloud native" has entered the scene, along with practices and technologies such as Platform-as-a-Service (PaaS), Site Reliability Engineering (SRE), and Serverless.

by Jeff Sussna

From reading the value propositions for these approaches, one might think that cloud-native goes in the opposite direction from DevOps. PaaS talks about "decoupling development from operations". Google's SRE team views itself as an operations organization, and sets rules governing its engagement with development teams. Serverless touts the benefit of not having to know or care about infrastructure, and seems to go hand-in-hand with NoOps, which seems to be about getting rid of operations altogether.

So what's going on? Are silos all the rage again? Was DevOps just a dream from which we quickly awoke? Not when you look beneath the surface. Done properly, cloud-native isn't about reinstating silos. It's about turning them on their head. Traditional IT was the "department of no". It sought to manage its challenges by asserting control, setting requirements, dictating the terms of engagement, and expecting its users to adapt to it rather than the other way around. "If you want a new server", IT said, "you need to fill out these four confusing forms, then wait 20-40 days." And so forth.

Cloud-native platforms and practices are taking hold because they offer capabilities that are both useful and usable. PaaS is able to decouple development from operations because it provides a single solution that addresses the needs of both perspectives: the ability to accelerate the delivery of code without degrading the scalability, resilience, security, or compliance of the environment in which that code runs. A healthy SRE practice doesn't say "sorry, you're out of luck and on your own until you figure out how to meet the error budget we've imposed on you."; it proactively helps application teams understand the meaning of that budget and how to meet it. Serverless works because it narrows the impedance mismatch between operations-centric concepts such as "computers", "networks", "storage" and development-centric concepts such as "functions" and "methods".

What all of these strategies have in common is their focus on helping applications teams accomplish their goals. In a delightful irony, ITIL, DevOps' favorite bogeyman, actually got it right. ITIL v3 defines "service" as "facilitating desirable outcomes". What do digital organizations want to accomplish? They want to deliver value to customers with as little friction as possible. They want to make that value available in a form that is scalable, resilient, and secure. Facilitating those outcomes is what cloud-native is all about.

Very few organizations are small enough to put the entirety of dev and ops into a single work-pod, take them all out to lunch together, or give them a single foosball table to share. Nearly every organization needs to figure out how to build a coherent whole out of multiple teams and disciplines. It was more than five years ago that I introduced the idea that empathy is the essence of DevOps. Empathy is the lubricant that makes systems thinking work. It's what enables us to understand and thus facilitate others' desired outcomes. Cloud-native only works when it reflects empathy with one's customers; when it reflects a service-centered approach ("how can I help?") rather than a product-centered approach ("here's the thing I tipped up"). When decoupling provides service, it provides value, and thus is entirely compatible with the essence of DevOps.

Jeff Sussna is the founder of Ingineering.IT, a Minneapolis technology consulting firm that helps enterprises and Software-as-a-Service companies adopt 21st-century IT tools and practices. He is also the author of 'Designing Delivery: Rethinking IT in the Digital Service Economy' and is a regular speaker on the topics of design and DevOps.

Function-as-a-Service with AWS Lambda and Knative

Kubernetes as a multi-cloud operating system

In the future, many companies will try to grow their IT infrastructure with the help of the Cloud or even relocate it completely to the Cloud. Larger companies often request multi-cloud solutions. In terms of serverless frameworks, there are several ways to achieve multi-cloud operation. Using AWS Lambda, a function can be provided and the whole thing can be made cloud-independent with Knative.

by Patrick Arnold

So what exactly is a multi-cloud? A multi-cloud involves the use of several multiple cloud providers / cloud platforms, while providing the user with the feeling that it is a single cloud. For the most part, people try to advance to this evolutionary stage of cloud computing to achieve independence from individual cloud providers.

The use of multiple cloud providers boosts resilience and availability, and of course enables the use of technologies that individual cloud providers do not provide. As an example, deploying your Alexa Skill in the Cloud is relatively difficult if you've decided to use Microsoft Azure as your provider. In addition, a multi-cloud solution gives us the ability to host applications with high requirements in terms of processing power, storage and network performance with a cloud provider that meets these needs. In turn, less critical applications can be hosted by a lower-cost provider to reduce IT costs.

Naturally, the multi-cloud framework is not just all benefit. By using multiple cloud providers, the design of your infrastructure becomes much more complex and more difficult to manage. The number of error sources can increase and the administration of billing for each individual cloud providers becomes more complex.

You should compare both the advantages and disadvantages before you make a decision. If you determine that you don't really need to be afraid of being dependent on a single cloud provider, you should rather invest the time in the use of the cloud services.

What is Function-as-a-Service?

In 2014, the Function-as-a-Service (FaaS) concept first appeared on the market. At that time, the hook.io concept was introduced. In the years that followed, all the major players in IT jumped on the bandwagon, with products such as AWS Lambda, Google Cloud Functions, IBM OpenWhisk or even Microsoft Azure Functions. The characteristics of such a function are as follows:

- The server, network, operating system, storage, etc. are abstracted by the developer
- Billing is based on usage with accuracy to the second
- FaaS is stateless, meaning that a database or file system is needed to store data or states.
- It is highly scalable

Yet what advantages does the entire package offer? Probably the biggest advantage is that the developer no longer has to worry about the infrastructure, but only needs to address individual functions. The services are highly scalable, enabling accurate and usage-based billing. This allows you to achieve maximum transparency in terms of product costs. The logic of the application can be divided into individual functions, providing considerably more flexibility for the implementation of other requirements. The functions can be used in a variety of scenarios. These often include:

- Web requests
- Scheduled jobs and tasks
- Events
- Manually started tasks

FaaS with AWS Lambda

As a first step, we will create a new Maven project. To be able to use the AWS Lambda-specific functionalities, we need to add the dependency seen in Listing 1 to our project.

The next step is to implement a handler that takes the request and returns a response to the caller. There are two such handlers in the Core Dependency: the *RequestHandler* and the *RequestStreamHandler*. We will use *RequestHandler* in our sample project and declare inbound and outbound as a string (Listing 2).

If you then execute a Maven build, a *.jar* file will be created, which can then be deployed at a later time. Already at this point you can clearly see that the *LambdaDependency* creates permanent wiring to AWS. As I had already mentioned at the beginning, this is not necessarily a bad thing, but the decision should be made consciously. If you now want to use the function to operate an AWS-specific database, this coupling grows even stronger.

Deployment of the function: There are several ways to deploy a Lambda feature, either manually through the AWS console or automatically through a CI server. For this sample project, the automated path was chosen using Travis CI.

Travis is a cloud-based CI server that supports different languages and target platforms. The great advantage of Travis is that it can be connected to your own GitHub account within a matter of seconds. What do we need to do this? Well, Travis is configured through a file called *.travis.yaml* . In our case, we need Maven for the build as well as for the connection to our AWS-hosted Lambda function for the deployment.

As you can see from the configuration in Listing 3, you need the following configuration parameters for a successful deployment:

- *Provider:* In Travis, this is the destination provider to deploy to in the deploy step
- *Runtime:* The runtime needed to execute the deployment
- *Handler name:* Here we have to specify our request handler, including package, class and method names
- *Amazon Credentials:* Last but not least, we need to enter the credentials so that the build can deploy the function. This can be done by using the following commands:
  - AccessKey*: travis encrypt "Your AccessKey" –add deploy.access_key*
  - Secret_AccessKey: *travis encrypt "Your SecretAccessKey" –add deploy.secret_access_key*

If these credentials are not passed through the configuration, Travis looks for the environment variables *AWS_ACCESS_KEY* and *AWS_SECRET_ACCESS_KEY*

Interim conclusion: The effort needed to provide a Lambda function is relatively low. The interfaces to the Lambda implementation are clear and easy to understand. Because AWS provides Lambda Runtime as SaaS, we don't have to worry about installation and configuration, and on top of that, we receive certain services right out of the box (such as logging and monitoring). Of course, this implies a very strong connection to AWS. So if for example, you want to switch to Microsoft Azure or to Google Cloud for some reason, you need to migrate the feature and adjust it accordingly in your code.

Listing 1: Core dependency for AWS Lambda

```
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-lambda-java-core</artifactId>
  <version>1.2.0</version>
</dependency>
```

Listing 2: Handler class for requests

```
public class LambdaMethodHandler implements RequestHandler<String, String>
{

  public String handleRequest(String input, Context context) {
    context.getLogger().log("Input: " + input);
    return "Hello World " + input;
  }
}
```

Listing 3: Configuration of "travis.yml"

```
language: java
jdk:
- openjdk8
script: mvn clean install
deploy:
  provider: lambda
  function_name: SimpleFunction
  region: eu-central-1
  role: arn:aws:iam::001843237652:role/lambda_basic_execution
  runtime: java8
  andler_name: de.developerpat.handler.LambdaMethodHandler::handleRequest
  access_key_id:
    secure: {Your_Access_Key}
  secret_access_key:
    secure: {Your Secret Access Key}
```

FaaS with Knative

The open-source Knative project was launched in 2018. The founding fathers were Google and Pivotal, but all the IT celebrities such as IBM and Red Hat have since joined the game. The Knative framework is based on Kubernetes and Istio, which provide the application environment (container-based) and advanced network routing. Knative expands Kubernetes with a suite of middleware components essential for designing modern container-based applications. Since Knative is based on Kubernetes, it is possible to host the applications locally, in the cloud or in a third-party data center.

Pre-Steps: Whether it's AWS, Microsoft, IBM or Google – nowadays every big cloud provider offers a "managed Kubernetes". For test purposes, you can also simply use a local MiniCube or Minishift. For my use case, I use the MiniCube supplied with Docker for Windows, which can be easily activated via the Docker UI (**Fig. 1**).

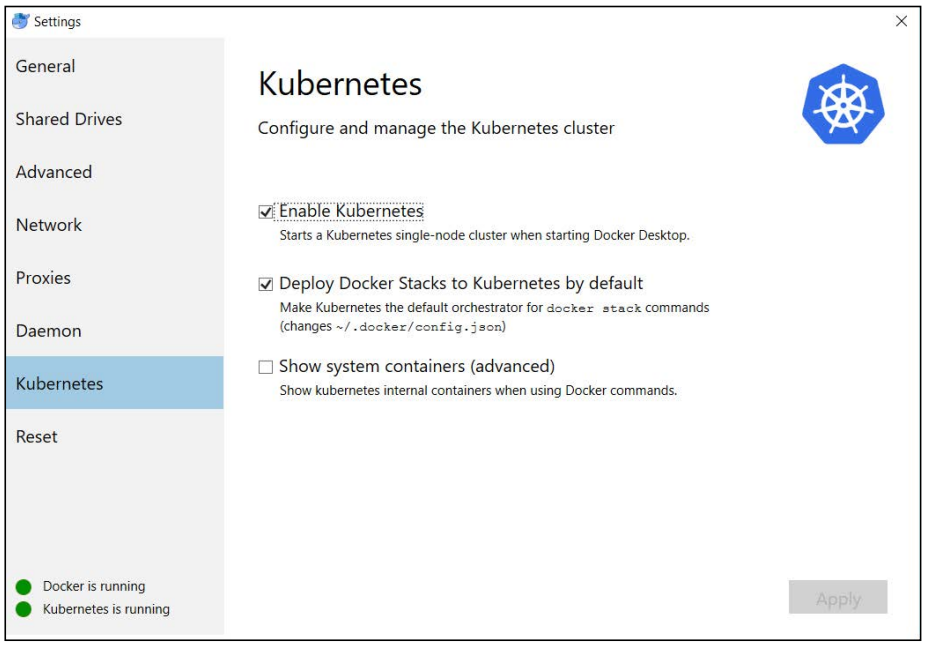


Fig. 1: Activation of Kubernetes via Docker UI

Unfortunately, the standard MiniCube alone does not give us anything. So how do we install Knative now? Before we can install Knative, we need Istio first. There are two ways to install Istio and Knative - an automated method and a manual one.

Individual installation steps for different cloud providers are provided in the Knative documentation. It should be noted here that the part specifically referring to the respective cloud provider is limited to the provisioning of a Kubernetes cluster. Once you install Istio, the procedure is the same for all cloud providers. The manual steps required for this can be found in the Knative documentation on GitHub [1].

For the automated method, Pivotal has released a framework called riff. We will run into this later on as well during development. riff is designed to simplify the development of Knative applications and support all core components. At the time this article is being written, it is available in version 0.2.0, assuming the use of a kubectl configured for the correct cluster. If you have this, you can use the command from Listing 4 to install Istio and Knative via the CLI.

```
>>riff system install
```

Warning! If you are installing Knative locally, the parameter *--node-port* needs to be added. After you run this command,

Listing 4: "spring-cloud-starter-function-web" dependency

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-function-web</artifactId>
</dependency>
```

the message *riff system install completed successfully* should appear within a few minutes.

Now we have our application environment with Kubernetes, Istio and Knative set up. Relatively fast and easy thanks to riff - in my opinion at least.

Development of a function: We now have a number of supported programming languages to develop a function that will then be hosted on Knative. A few sample projects can be found on the GitHub presence [2] of Knative. For my use case, I decided to use the Spring Cloud Function project. This specifically helps deliver business value as a function, while bringing all the benefits of the Spring universe with it (autoconfiguration, dependency injection, metrics, etc.).

As a first step, you add the Dependency described in Listing 4 to your *pom.xml*.

Once we have done that, we can write our little function. To make the Lambda function and the Spring function comparable, we will implement the same logic. Since our logic is very minimal, I will implement it myself in *Spring Boat Application*. Of course, you can also use the Dependency injection as you normally would with Spring. We use the *java.util.function.Function* class to implement the function (Listing 5). This will be returned as an object of our *Hello* method. It is important that the method with the *@Bean* annotation is used, otherwise the end point will not be released.

Knative Deployment: To make sure we can provide our function, we need to first initialize our namespace with riff. Here the user name and the Secret of our Docker registry are stored so that the image can be pushed when creating the function. Since we do not have to write the image ourselves, riff practically takes this over for us. For this purpose, a few CloudFoundry build packs are used. In my case, I use the Docker Hub as a Docker registry. Initialization can be run using the following command:

```
>>riff namespace init default --dockerhub $DOCKER_ID
```

If you want to initialize a namespace other than *default*, just change the label. But now, we want to deploy our function. Of course, we have checked everything into our GitHub repository and we now want to build this state and subsequently deploy it. Once riff has built our Maven project, a Docker image should be created and pushed into our Docker Hub repository. The following command takes care of this for us:

```
>>riff function create springknative --git-repo
https://github.com/developerpat/spring-cloud-function.git --image
                developerpat/springknative:v1 --verbose
```

If we want to do all that from a local path, we swap *--git-repo* with *--local path* and add the corresponding path. If you now take a look at how the command runs, you recognize that the project is analyzed, created with the correct build pack, and the finished Docker image is pushed and deployed at the end.

Calling the function: Now we would like to test - which is also relatively easy to do with AWS via the console - whether the call works against our function. We can do that very simply as follows thanks to our riff CLIs:

```
>>riff service invoke springknative --text -- -w '\n' -d Patrick
curl http://localhost:32380/ -H 'Host: springknative.default.example.com' -H
'Content-Type: text/plain' -w '\n' -d Patrick
Hello Patrick
```

Using the *invoke* command, you can generate and execute a *curl* as desired. As you can see now, our function works flawlessly.

Can Knative compete with an integrated Amazon Lambda?

Due to the fact that Knative is based on Kubernetes and Istio, some features are already available natively:

- **Kubernetes**
- Scaling
- Redundancies
- Rolling out and back
- Health checks
- Service discovery
- Config and Secrets
- Resilience
- **Istio:**
- Logging
- Tracing
- Metrics
- Failover
- Circuit Breaker
- Traffic Flow
- Fault Injection

This range of functionality comes very close to the functional scope of a Function-as-a-Service solution like AWS Lambda. There is one shortcoming, however: There are no UIs to use the functionality. If you want to visualize the monitoring information in a dashboard, you need to set up a solution yourself for it (such as Prometheus). A few tutorials and support documents are available in the Knative documentation (/*TODO*/). With AWS Lambda, you get those UIs innately and you don't need to worry about anything anymore.

Multi-cloud capacity

All major cloud providers now offer a managed Kubernetes. Since Knative uses Kubernetes as the base environment, it is completely independent of the Kubernetes provider. So it's no issue to migrate your applications from one environment to another in a very short amount of time. The biggest effort here is to change the target environment in the configuration during deployment. Based on these facts, simple migration and exit strategies can be developed.

Conclusion

Knative does not fall in line with all aspects of the FaaS manifest. So is it an FaaS at all then? From my personal point of view, by all means yes. For me, the most important item in the FaaS manifesto is that no machines, servers, VMs or containers can be visible in the programming model. This item is fulfilled by Knative in conjunction with the riff CLI.

Listing 5: Implementation of our function

```
package de.developerpat.springKnative;

import java.util.function.Function;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class SpringKnativeApplication {

  @Bean
  public Function<String, String> hello(){
    return value -> new StringBuilder("Input: " + value).toString();
  }

  public static void main(String[] args) {
    SpringApplication.run(SpringKnativeApplication.class, args);
  }
}
```

Patrick Arnold is an IT Consultant at Pentasys AG. Technology to him is like a toy for little kids - he's addicted to new things. Having gone through old-school education in the mainframe area, he switched to the decentralized world. His technological focus is on modern architectural and development approaches such as the Cloud, API-Management, Continuous Delivery, DevOps and Microservices.

References

[1] Knative documentation on GitHub: https://github.com/knative/docs/blob/master/docs/install/README.md

[2] https://github.com/knative/docs/tree/master/docs/serving/samples

Tutorial: an introduction to Ansible

Multi-tier deployment with Ansible

Ansible, by Red Hat, provides automated configurations as well as orchestrations of machine landscapes. The lightweight and flexible tool is increasingly used in DevOps toolchains and cloud computing. This article shows how to get a complex multi-tier setup on the server with it.

by Daniel Stender

As a configuration and provisioning framework for machines, *Ansible* provides automated and reproducible management for IT landscapes. The relatively young but already quite technically mature tool belongs to the application class Infrastructure-as-Code (IaC) and is in competition with solutions such as Puppet, Chef and Salt/Saltstack, some of which have already been established for some time.

The software was acquired by Red Hat in 2015 and it is currently positioned in the market as part of a comprehensive open source software stack for application provisioning offered by this company.

The CLI-based tools of Ansible, which are also available also outside of the commercial offering, are referred to as the Ansible Engine. The provisioner offers a clearly structured and well-arranged component model consisting of a machine inventory, modules, playbooks and roles. It deliberately follows a lightweight approach and can be used successfully after a relatively short training period.

Ansible is mainly used for the controls of Linux, BSD and Unix machines and doesn't require an agent on the target systems. It also doesn't require a central server and only needs to be installed on the user's control node resp. workstation and kept up-to-date there. For an operation on a target machine, a SSH access and an installed Python interpreter (this 2.7 by default) are required there, so that Ansible starts at a higher level than working with "bare metal" systems that don't have an installed operating system, and thus cannot provide this.

Ansible projects are used to move one or more machines from specific states (usually the basic installation of an OS is assumed) to another specific state (for example, to function as a web or database server). The software has an imperative approach (*how* to achieve something). Supporters of this method point to its advantages, such as easier debugging and full control. However, the Ansible user always needs to know the initial state of a machine in very much detail, just like an administrator, and he also needs to know how to implement the required procedures ("plays") with the required individual steps (tasks) in the correct order in Ansible. And this can become quite complex rather fast.

Ansible

Ansible utilizes the user-friendly YAML format for scripting procedures in workflows (playbooks) and other elements. It also offers the integrated powerful template language Jinja 2 as its killer feature, which was developed by the Pocoo group as a stand-alone-project and is used also in other Python projects. For the implementation of different administrative operations on targeted systems, Ansible provides a comprehensive toolbox with hundreds of supplied modules. A profound knowledge of Ansible is to a large extent means knowing what modules are included for which purpose and how they behave in detail.

A couple of modules form a manageable basic set for fundamental administrative tasks on a target system. Tasks such as uploading files or importing them from the network, installing packages, creating users and directories, attaching lines to configuration files or changing them, as well as restarting services. Some standard modules, such as *ping*, are particularly suitable for being set-off ad hoc without a playbook but with the provided CLI tool *ansible*. Furthermore, there are many special modules available that can be used with MySQL servers, for example, to set up users and databases. The additional Python libraries that are required for the application of some modules on the target system (e.g. for MySQL) can then simply be installed as a first step in the same playbook.

Partially, there are alternative modules for specific purposes and sometimes there are also alternative ways how certain things can be achieved. Additionally, modules can also be used creatively, like the *subversion* module. This can be employed in such a way that it imports individual directories directly from GitHub projects onto the target machines. The supplied CLI tool *ansible-doc* offers a fast access to the documentation

of all provided modules on the control node similar to Linux Manpages. The infrastructure and cloud computing modules, which are now fully established in Ansible and only indirectly related to machine provisioning procedures, play a major role in the more recent versions of the provisioner. They make use of Ansible's procedural principle for their own purposes, e.g. for pulling up cloud infrastructure and remotely controlling network hardware.

Roles in Ansible function in the sense of "the role that a machine takes" as an organizational structure for the components of projects. A complex Ansible project can contain multiple playbooks and a single role offers a set of fixed defined directories (Listing 1): *tasks/* is for the playbooks, *vars/* and *defaults/* are for the definition of variables, and *handlers/* for the definition of switches, they hold YAML configuration files, and in the directories *files/* and *templates/* you can provide arbitrary files and templates for the installation during the Ansible run. You can create the skeleton for a new role with the included CLI tool *ansbile-galaxy*, where the unneeded generated directories with included templates (always *main.yml*) can easily be deleted; *meta/* , for example, is mainly intended to distribute the role via the official repository of Ansible – the galaxy.

In Ansible projects the user can define any custom variables and evaluate them together with the built-in variables, which all start with *ansible_*, if required. The so-called "facts" do play a special role in this regard. This is comprehensive information that Ansible collects from all connected machines when running a playbook and which can be completely outputted for development purposes with the *setup* module. For example, Ansible researches the IP addresses and hostnames of all connected machines during the run and the user can evaluate these in templates for configuration files for complex setups in which nodes must be able to reach each other.

Example

We will use a classic multi-tier setup, which was implemented with open source software, and consists of three components as an example for a slightly more complex Ansible project with three roles (Listing 1). The MySQL-Fork *MariaDB* is a relational database server running on a backend machine on which the test database *test_db* is installed, which is well-known among MySQL developers. It is a fictitious personnel database of a non-existent large corporation with six tables which contain around 300 000 persons and several million salary data entries. To use this database, a micro service written with the Python web framework Flask [1] is installed on frontend machines, which queries this database when called and returns a JSON object. This contains the current top earner of one of the nine company departments that are included in this personnel data. This web application is based

Listing 1: The example project

```
|-- group_vars
|   |-- all.yml
|-- hosts
|-- roles
|   |-- flask
|   |   |-- files
|   |   |   |-- querier.conf
|   |   |   |-- querier.wsgi
|   |   |-- handlers
|   |   |   |-- main.yml
|   |   |-- tasks
|   |   |   |-- main.yml
|   |   |-- templates
|   |   |       |-- querier.py.j2
|   |-- haproxy
|   |   |-- handlers
|   |   |   |-- main.yml
|   |   |-- tasks
|   |   |   |-- main.yml
|   |   |-- templates
|   |   |   |-- haproxy.cfg.j2
|   |   |-- vars
|   |   |       |-- main.yml
|   |-- mariadb
|       |-- handlers
|       |   |-- main.yml
|       |-- tasks
|           |-- main.yml
|-- site.yml
```

Listing 2: "roles/mariadb/tasks/main.yml"

```
- name: MariaDB und benötigte Pakete installieren
  apt:
    name: "{{ item }}"
    state: latest
    update_cache: yes
  with_items:
    - mariadb-server
    - python-mysqldb
    - unzip

- name: Datenbank "employees" anlegen
  mysql_db:
    name: employees
    state: present

- name: SQL-Benutzer "employees" anlegen
  mysql_user:
    name: employees
    host: "%"
    password: "{{ employees_password }}"
    priv: "employees.*:ALL"
    state: present

- name: check ob test_db bereits importiert ist
  stat:
    path: /var/lib/mysql/employees/employees.frm
  register: testdb_imported

- name: test_db von Github einspielen
  unarchive:
    src: https://github.com/datacharmer/test_db/archive/master.zip
    dest: /tmp
    remote_src: yes
  when: testdb_imported.stat.exists == false

- name: Pfade in Importskript anpassen
  replace:
    path: /tmp/test_db-master/employees.sql
    regexp: "source"
    replace: "source /tmp/test_db-master/"
  when: testdb_imported.stat.exists == false

- name: test_db importieren
  mysql_db:
    name: all
    state: import
    target: /tmp/test_db-master/employees.sql
  when: testdb_imported.stat.exists == false

- name: MariaDB für Fernzugriff freischalten
  lineinfile:
    dest: /etc/mysql/mariadb.conf.d/50-server.cnf
    regexp: "^bind-address(.+)127.0.0.1"
    line: "bind-address = 0.0.0.0"
    backrefs: yes
  notify: restart mariadb
```

on an Apache 2 web server with WSGI extension, which Flask needs as an interface to communicate with the server.

The load balancer *HAProxy* is installed on another node. For the purpose of reliability and load-sharing, it distributes requests from the outside network to any number of dedicated frontend nodes with the same Flask application, all of which access the same backend with the database in parallel. HAProxy is a powerful enterprise-level software solution that is used by many well-known providers such as Twitter and GitHub. This application makes only limited sense, even beyond the fictitious personal data, because the data does not change at all and the queries always return the same results. But this is, nevertheless, an overall-setup, which is much common in practice and appears oftenly in variations. Ansible is suitable for completely automatically pulling up this entire structure with the push of a button and thereby deploying the required components on machines (hosts). The playbooks are written for the basic installations of Debian 9 and contain several customized details, such as the used package names.

Backend

To install the MariaDB server you only need a playbook (Listing 2), but no configuration templates. Plus, files to be placed on the control node for uploading can be omitted, because the sample database can be imported directly from the net.

At first, the playbook installs the package *mariadb-server* from the official Debian archive with the module *apt* for the package manager. For later operations with Ansible, two more packages are needed on this host, which are *python-mysqldb* for the MySQL modules of Ansible and *unzip* to unpack the downloaded example database. There is nothing wrong with installing all three packages in one step. Ansible contains the variable *item*, which can be used in conjunction with *with_items* to construct you own iterators, as it is shown here in the example.

In Ansible, or respectively Jinja2, variables are always expanded with doubled curly brackets and the syntax in the examples (Ansible provides an alternative syntax which isn't YAML conform) also requires it to put these constructs in quotation marks. The database server is already activated after the installation of the package, the playbook then creates a new empty database and a user for it in two further steps with the modules *mysql_db* and *mysql_user*. The password-preset in the user-defined variable *employees_password* is also required for the Flask application. Therefore, it is a good idea not to define it in both roles in parallel in *vars/*, but centrally on a higher level in *group_vars/* (Listing 3).

The test database installation is the next step. But it is best to first create a checking mechanism in order to prevent this from happening at every new Ansible run, because this creates an unnecessary overhead. The *stat* module is well suited

Listing 3: "group_vars/all.yml"

```
employees_password: fbfad90d99d0b4
```

Listing 4: "roles/mariadb/handlers/main.yml"

```
- name: restart mariadb
  service:
    name: mariadb
    state: restarted
    enabled: true
```

Listing 5: "roles/flask/tasks/main.yml"

```
- name: Apache und benötigte Pakete installieren
  apt:
    name: "{{ item }}"
    state: latest
    update_cache: yes
  with_items:
    - apache2
    - libapache2-mod-wsgi
    - python-flask
    - python-mysqldb

- name: WSGI-Starter aufspielen
  copy:
    src: querier.wsgi
    dest: /var/www/querier/

- name: Pseudo-User für WSGI-Prozesss anlegen
  user:
    name: wsgi
    shell: /bin/false
    state: present

- name: Applikation aufspielen
  template:
    src: querier.py.j2
    dest: /var/www/querier/querier.py
    owner: wsgi
    mode: 0600
  notify: reload apache

- name: Konfiguration für virtuellen Host aufspielen
  copy:
    src: querier.conf
    dest: /etc/apache2/sites-available/
  notify: reload apache

- name: virtuellen Host enablen
  file:
    src: /etc/apache2/sites-available/querier.conf
    dest: /etc/apache2/sites-enabled/querier.conf
    state: link

- name: Default-Startseite disablen
  file:
    path: /etc/apache2/sites-enabled/000-default.conf
    state: absent
```

for this task with which you can check whether the *employees.frm* file already exists (which is the case if the database has already been installed), and the return of the module can be incorporated in a variable (here in *testdb_imported*) with *register*. In the next step the *unarchive* module imports the database as a ZIP file from GitHub into */tmp* and unpacks it. But this only happens (*when*) if the return value of *testdb_imported.stat.exists* is negative. The *replace* module will then adjust some paths in the import script from the ZIP archive and here, as well as in the next steps, the playbook will set with *when* the same condition for the execution. The next step uses the *mysql_db* module again to install the unpacked personnel database to the MariaDB server by using the import script which is shipped with it.

To enable the database server for access from the network it is necessary to change a line in a configuration file under */etc/myslq*, which can be done with the module *lineinfile*. The *backrefs* option for this module prevents the same line from being rewritten when this playbook is run again: Otherwise it will be appended again and again at every run of this role when the *regexp* expression is not found (anymore) in this file. This step activates the switch (a handler) *restart mariadb*, which is defined in Listing 4 for the *service* module with *notify* if required (if the module returns *changed* as result). The handler ensures that the MariaDB server re-reads its configuration files, with *enabled* it is simultaneously specified that the associated *Systemd* service should be active again after a reboot of the target machine (the module switches the service unit if that isn't the case).

By using handlers instead of fixed steps the *service* module can be prevented from reloading or restarting the Systemd service repeatedly every time the playbook runs again, which is then just not necessary to happen every time when this part isn't further developed, but remains the same. Ansible's imperative method requires you to think carefully when writing procedures and to always keep an eye also on the repeated run of a playbook.

Listing 6: "roles/flask/files/querier.wsgi"

```
import sys
sys.path.insert(0, '/var/www/querier')
from querier import app as application
```

Listing 7: "roles/flask/tasks/main.yml"

```
from flask import Flask
import json
import MySQLdb as mysqldb

app = Flask(__name__)

ipv4 = '{{ ansible_eth0.ipv4.address }}'
hostname = '{{ ansible_hostname }}'

mydb = mysqldb.connect(user = 'employees',
    host = '{{ hostvars[groups.datenbank.0].ansible_default_ipv4.address }}',
    passwd = '{{ employees_password }}',
    db = 'employees')

@app.route("/&lt;abteilung&gt;")
def topearners(abteilung):
    cursor = mydb.cursor()

    command = cursor.execute("""SELECT e.last_name, e.first_name, d.dept_no,
        max(s.salary) as max_sal FROM employees e
        JOIN salaries s ON e.emp_no = s.emp_no AND s.to_date &gt; now()
        JOIN dept_emp d ON e.emp_no = d.emp_no
        WHERE d.dept_no = 'd%s'
        GROUP BY e.emp_no ORDER BY max_sal desc limit 1;""" % abteilung)

    results = cursor.fetchall()
    daten = (results[0])
    (nachname, vorname, abteilung, gehalt) = daten
    resultsx = (abteilung, vorname, nachname, gehalt, ipv4, hostname)
    return json.dumps(resultsx)
```

Frontend

The playbook for the setup of the Flask application (Listing 5) is a bit shorter, but several files have to be provided in this role so that Ansible can upload them.

Initially some packages are installed again in one step: the web server *apache2*, the corresponding WSGI extension, Flask, and the same Python library for MySQL again – this time it is not for an Ansible module but for the Flask application. You then install a WSGI launch script (Listing 6) to the target machine with *copy*. This module automatically searches in the *files/* folder of this role so you don't have to specify a source path here. The target path is automatically created by the module if it doesn't exist already. The WSGI process requires a pseudo-user on the target system to avoid running with root privileges. The playbook then creates this easily with the module *user*.

With the next step the actual application *querier.py* (Listing 7) can be loaded.

The Python script is set up as an Ansible template (with the extension *.j2*) and must be processed accordingly with the module *template* instead of *copy*. It has to be made available in *templates/* for this purpose.

During the installation on the target system the variables in the template are expanded from the facts, and afterwards the IP address (*ansible_eth0_ipv4.address*) and the host name (*ansible_hostname*) of the respective target system could be found hard-coded here in place to appear in the output when the script runs. The flask application returns a JSON object with the result of the database query, whereby the accompanying IP address and the hostname just serves to check where the return of the load balancer actually comes from.

With *hostvars* you also get the IP address of the database backend from the inventory (Listing 8), which the application needs for the query of the database over the network. Just like in the case of the backend, *employees_password* is evaluated for the access password (Listing 3). This step is linked to the

handler *reload apache* (Listing 9), which is triggered when you make changes to the template and then run the playbook through again to re-deploy that new version. In the next step, the playbook loads the configuration for the virtual host for the Apache webserver (Listing 10) on which the Flask app will run.

Two further steps are activating the virtual host with the common method for Apache. The first step is the creation of a softlink below */etc/apache2* with the module *file*. Then you delete the softlink which is already there for the Apache default starter page with the same module. Since it is not to be expected that these steps will undergo any change in the future (that is unless Apache happens to be changing) there is no need to supply these steps also with the *reload apache* handler. This doesn't represent a problem for the first run of the playbook (like the configuration changes aren't recognized by Apache which is already running) because this handler is already triggered by previous steps, and Ansible always sets off handlers at the end of a run. Thus, these changes to the default Apache configuration are ensured to take effect. Another subtle fact is that the *reloaded* switch for the *service* module not just reloads but also always starts a Systemd service if it is not already running. This for example is the case after installing the *apache2* package.

Listing 8: "hosts"

```
[datenbank]
167.99.242.69

[applikation]
167.99.242.84
167.99.242.179
167.99.242.237

[load-balancer]
167.99.250.42

[all:vars]
ansible_user=root
ansible_ssh_private_key_file=
                    ~/.ssh/id_digitalocean
```

Listing 9: "roles/flask/handlers/main.yml"

```
- name: reload apache
  service:
    name: apache2
    state: reloaded
    enabled: yes
```

Listing 10: "roles/flask/files/querier.conf"

```
<VirtualHost *:80>
 WSGIDaemonProcess querier user=wsgi group=wsgi threads=5
 WSGIScriptAlias / /var/www/querier/querier.wsgi
 <Directory /var/www/querier>
  WSGIProcessGroup querier
  WSGIApplicationGroup %{GLOBAL}
  Order allow,deny
  Allow from all
 </Directory>
</VirtualHost>
```

Listing 11: "roles/haproxy/tasks/main.yml"

```
- name: haproxy-Paket installieren
  apt:
    name: haproxy
    state: latest
    update_cache: yes

- name: Konfiguration einspielen
  template:
    src: haproxy.cfg.j2
    dest: /etc/haproxy/haproxy.cfg
      backup: yes
    notify: reload haproxy
```

Listing 12: "roles/haproxy/handlers/main.yml"

```
- name: reload haproxy
  service:
    name: haproxy
    state: reloaded
    enabled: yes
```

Load Balancer

For the installation of the load balancer you'll need another handler to reload the corresponding Systemd service (Listing 12) and only two steps in the playbook (Listing 11), because everything crucial is happening in the template for the configuration data for the HAProxy (Listing 13). Under *backend* (meaning the backend of the Load Balancer) the necessary nodes have to be integrated. This is done by using a *for* loop provided by Jinja2 to iterate over the nodes in the *applikation* group in the inventory (Listing 8) and using *hostvars* to write their IP addresses and hostnames from the facts into this configuration file.

The programming elements of Jinja2 such as *if* and *for* are always within simple brackets with percentage signs, like shown above in this file also at the *if statspage*: This block for the HAProxy dashboard will only be outputted if *statspage* is set to true, as in the example the master playbook shows

Listing 13: "roles/haproxy/templates/haproxy.cfg.j2"

```
global
  daemon
  maxconn 256

{% if statspage %}
listen stats
  bind 0.0.0.0:9000
  mode http
  stats enable
  stats uri /haproxy?stats
  stats refresh 15s
{% endif %}

defaults
  mode http
  timeout connect 10s
  timeout client 1m
  timeout server 1m

frontend http
  bind {{ balancer_listen_address }}:
       {{ balancer_listen_port|default(
                                  '80') }}
  default_backend querier

backend querier
{% for host in groups['applikation']
%}
  server {{ hostvars[host].ansible_
       hostname }} {{ hostvars[host].
       ansible_default_ipv4.address }}:
                                  80 check
{% endfor %}
```

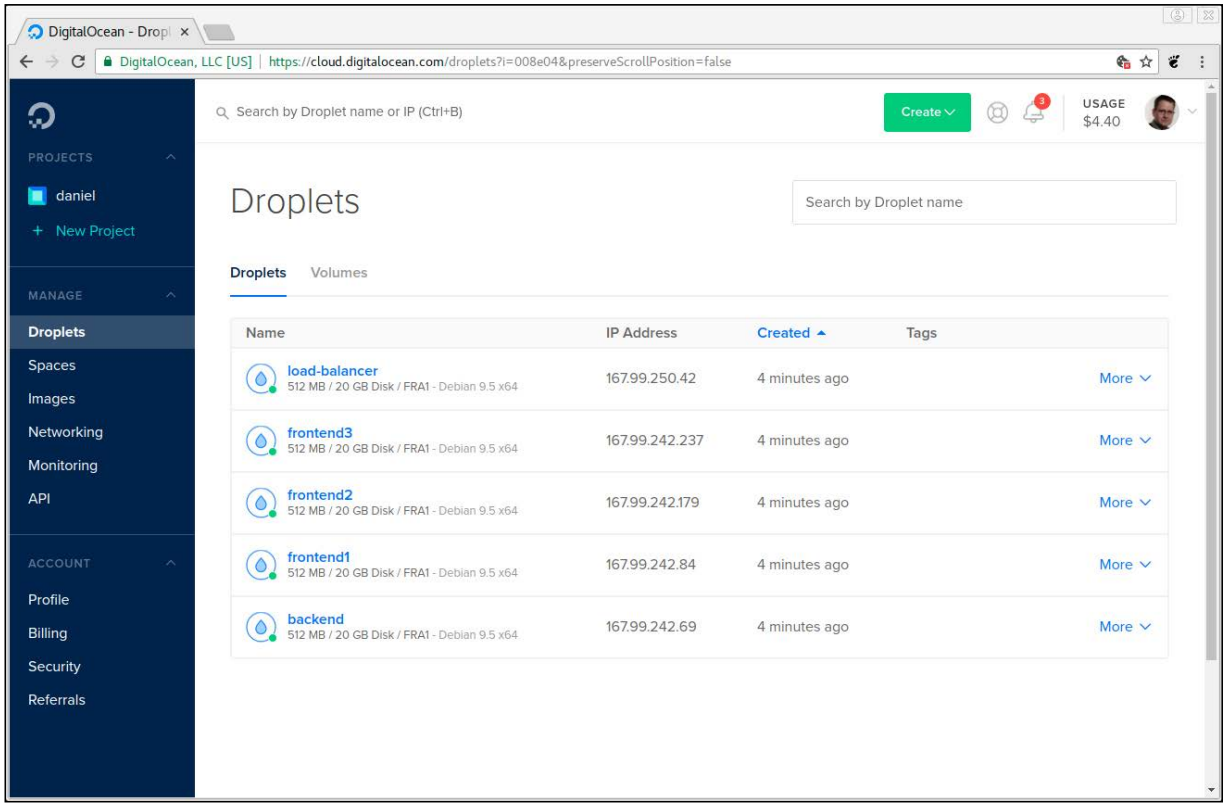


Fig. 1: The user dashboard of DigitalOcean

when calling the role (Listing 14). Under *frontend* (meaning the frontend of the load balancer), two additional self-defined variables are evaluated. These are only valid for this role, which is why they are best defined under *vars/* (Listing 15). For the port on which HAProxy awaits requests, the preset is the number 80 here with a special variable filter (*default*). If you want to change this, you just set *balancer_listen_port* accordingly in the variables files (Listing 15).

Deployment

To utilize a role two additional components are needed within an Ansible project. These components are an inventory resp. an inventory file (with any name, but it is often *hosts* or *inventory*) and a master playbook (quite often *site.yml*). Ansible inventories are written in simple *INI* format; they take IP addresses or DNS hostnames in, and can group the machine inventory associated therein with the project as desired (Listing 8).

The database group contains only one machine. Multiple entries would be also possible and they are assembled homogeneously, but the application only considers the first node (*groups.database.0* in Listing 7). Beneath the applikation nodes you can scale arbitrarily and go beyond the three servers of the example if an extreme request load is to be expected because, just like it was said before, the template for the HAProxy iterates over all the nodes that are registered here. You can simply add more machines later, if it is necessary. Ansible must be run again to do this in order to populate the newly added nodes and to update the load balancer accordingly.

Other *load-balancer* nodes are of course also possible and work in the same way, but the use of HAProxy actually is supposed to overcome alternative accesses. Frontend nodes that are out-of-order are automatically caught by the remaining ones. However, if you want to compensate the failure of the database or the load balancer, you need an even more complex setup with built-in monitoring. The example used in this article uses virtual servers from *DigitalOcean* (**Fig. 1**). The username (*ansible_user*) and path to the private SSH key on the workstation (*ansible_ssh_private_key_file*) can be given directly in the inventory.

The master playbook (Listing 14) links the groups from the inventory (hosts) with the roles in the project and determines the order in which the deployment should take place. To do this, simply trigger *ansible-playbook -i hosts site.yml* and Ansible will work through the entire project. During the run, a log file is output which lists the individual steps and shows whether changes have taken place (which is the case every-

Listing 14: "site.yml"

```
- hosts: datenbank
  roles:
    - mariadb

- hosts: applikation
  roles:
    - flask

- hosts: load-balancer
  roles:
    - {role: haproxy, statspage: true}
```

Listing 15: "roles/haproxy/vars/main.yml"

roles/haproxy/vars/main.yml

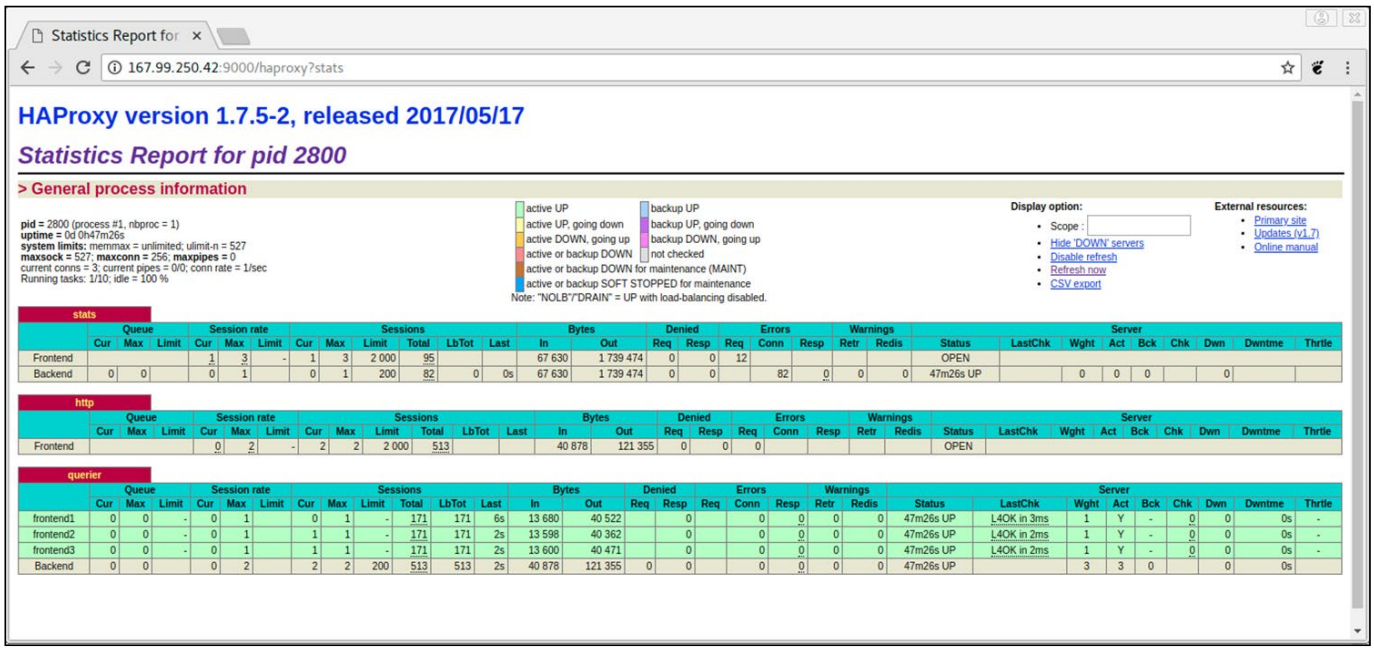


Fig. 2: The statistics page of HAProxy

where during the first run). The entire setup is installed after a few minutes. Then, the load balancer just has to be addressed. For this purpose the nine departments of the fictitious group are available as endpoints (001-009):

```
$ curl 167.99.250.42/001

["d001", "Akemi", "Warwick", 145128, "167.99.242.179", "frontend2"]

$ curl 167.99.250.42/002

["d002", "Lunjin", "Swick", 142395, "167.99.242.237", "frontend3"]

$ curl 167.99.250.42/003

["d003", "Yinlin", "Flowers", 141953, "167.99.242.84", "frontend1"]
```

The returned JSON object always contains first and foremost the queried department, the first and last name of the respective top earner, the current annual salary of this person, as well as the IPv4 address and the hostname of the frontend from which the return originates, like explained. If you call the HAProxy dashboard (**Fig. 2**), you can follow the load balancer doing its work. It always takes a few seconds until the result of a request arrives because the backend has to work through about 160 MB of data each time. Should the request become larger, then it is better to provide the MariaDB server with a more powerful hardware and also to take more in-depth tuning measures.

Conclusion

Ansible is a relevant tool for deploying applications on servers. Not only single-node setups but also multi-tier setups can be implemented with it. Ansible unfolds its full strength and offers the roles of a proven means of structuring complex projects. The example has shown how to use this provisioner to deploy a setup with three communicating components to five nodes. It also showed how playbooks and the built-in modules are used to perform procedurally ordered steps on the target machines.

The example setup is not suitable for production and for potential attackers it is rather a challenge at the primary school level: The database server, for example, is not secured (no root password is set and anonymous access is possible), the frontends are individually addressable, the internal connections are unsecured (you would rather use a private network for this), the load balancer is not addressable via HTTPS, etc.

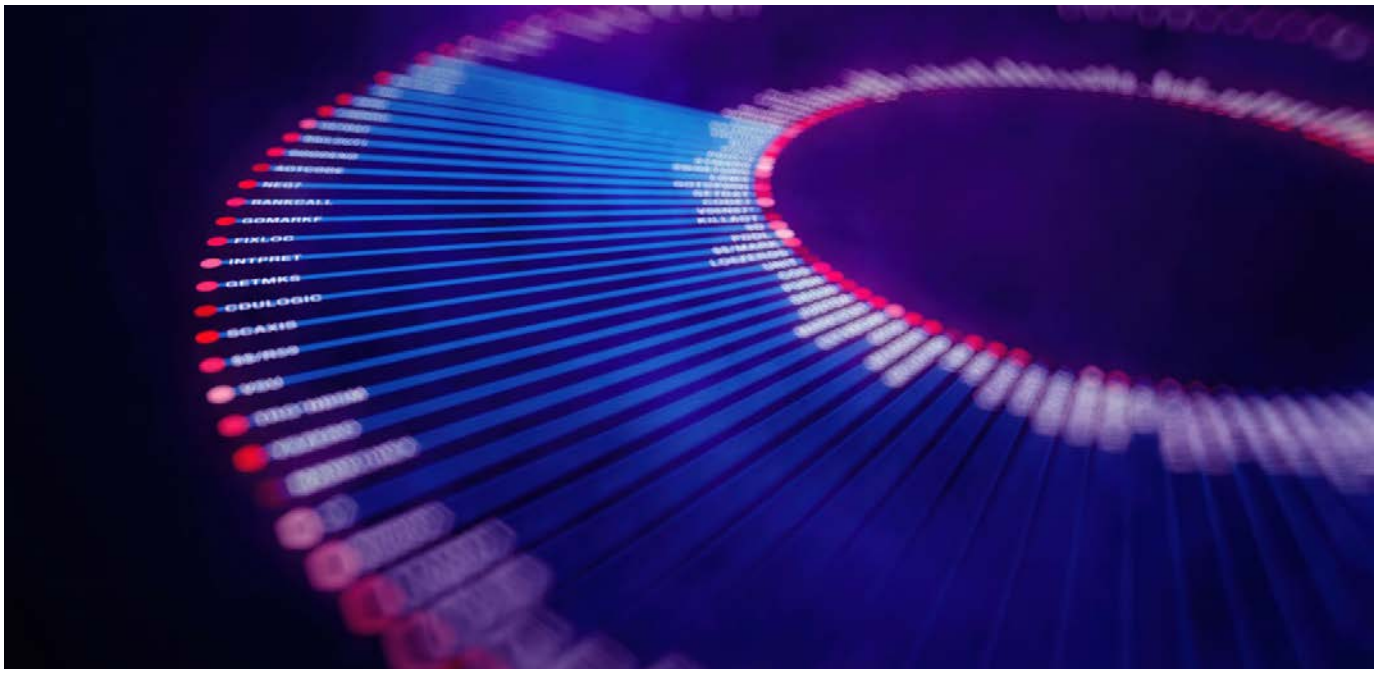
Hardening the MariaDB server would also be done with Ansible tools according to the script *mysql_secure_installation*, which is included in the package. Of course, the setup itself should not be in the foreground here, but rather how to get such a construct installed automatically with Ansible and offer starting points for a more engaged and detailed activity with this tool. But be careful, like many other DevOps tools, Ansible has a certain addiction potential.

Daniel Stender is a DevOps Engineer and works for the ITP Nord, Ratbacher and Aventes as technicial advisor for key customers in the financial sector.

https://danielstender.com stender@debian.org

References

[1] Stender, Daniel: „Mit Flask Webapplikationen in Python entwickeln“, in: Entwickler Magazin 6.2017, S. 68-75.



© HandMadeFont.com/Shutterstock.com

A brief discussion on the importance of service meshes

Do we need a service mesh?

Over the past year, service mesh has come to the fore as a crucial component of the cloud native stack. Giants like Ticketmaster and PayPal have all added a service mesh to their production applications. But what is a service mesh, exactly? Is it even relevant? And why?

by Anton Weiss

If I had to choose one concept that has really captured my attention in the last couple of years it would probably be 'liquid software' (**Fig. 1**). Coined by Fred Simon, Baruch Sadogursky and Yoav Landman of JFrog it refers to "high quality applications that securely flow to end-users with zero downtime". In the book with the same name they describe the world of a near future, where code updates are being poured continuously into all kinds of deployment targets – web servers, mainframes, mobile phones, connected cars, or IoT devices. I must admit I've always been abnormally excited by this notion of information flowing freely, without restrictions and borders. This is probably impacted by the fact I grew up in the Soviet Union, where information was held hostage, passed around in whispers. Where it was a bomb waiting to explode. And then it did all blow up – exactly as I was turning into a teenager. Perestroika arrived to set the information free, and my young heart on fire. So yes, I'm an information freedom junkie, don't blame me for that.

The platform for distributed trust

But personal biases aside – the information revolution has changed how our world operates. In this increasingly connected world, we can no longer rely on central, institutional authorities to get our information. Rachel Botsman wrote a great book called 'Who Can you Trust' which talks about the evolution of trust in human society (**Fig. 2**). In the book she shows how trust has graduallydeveloped from local – as

Fig. 1: Liquid Software

in prehistoric societies – to institutionalized and centralized. And how we are witnessing the fading of centralized trust as it is replaced by the distributed model. A model in which we trust an AirBnB house owner on the other side of the globe just because there are a couple dozen more folks on the internet who put the same trust in her.

Guess what enables this distribution of trust? You're right, information technology, of course! And yes, that information technology must be both up to date and working at any given moment, please. 24/7 – 365 days a year.

There's a problem, though. Not only are modern execution platforms increasingly distributed, they are also fragmented and heterogenous. We've only grown somewhat accustomed to the complex, fluid, ephemeral world of microservice architectures, and now we also need to support the event-driven paradigms of serverless and edge computing. How can we roll out continuous updates with certainty when there's a scary, chaotic world full of unexpected events waiting for our code out there?

Guess what enables the distribution of trust? You're right, information technology!

Progressive delivery

But let's take a step back and remind ourselves why we need all this complexity in the first place.

There are three main driving forces behind splitting our monolithic applications into microservices, and further into ephemeral functions and edge deployments: **scalability**, **resilience,** and **continuous delivery**. Large monoliths are quite bad at all 3 of them, while service-oriented systems supposedly promise to give us all three. I say supposedly because this trinity also exists in eternal mutual tension. Continuous delivery and scaling are relentlessly changing the logic and configuration of the system – putting its resilience under constant stress. On one hand, scalability can be seen as a feature of resilience, on the other, it makes a system more complex and thus more brittle. Features that provide us robustness (robustness and resilience aren't the same, but they do share several qualities) can also cause resistance to change. And what is especially important for today's subject is that horizontal scalability makes all our systems – no matter if they run in the cloud – increasingly dependent on the **network tissue**, which has now become the main integration point. This is where our components meet and also where they meet the external services they interact with… I'm not sure if this distinction even matters anymore. True resilience requires treating all services like strangers bound with a clear official contract – no matter whether they are internal or external. And the resulting crazy interdependence matrix turns continuous delivery into an equation with a multitude of unknowns.

The necessity to provide ongoing updates under these unpredictable conditions is making release safety practices such as blue-green deployments, canary releases, dark launches, and traffic mirroring ever more relevant. We need them both to keep our systems running and to keep our engineers sane. Thanks to James Governor there's now even a new umbrella definition for all these techniques: *progressive delivery*. I.e. the type of delivery that allows us to gradually pour new freshly brewed liquid software into the data pools, lakes, and seas.

But it suddenly becomes very hard to implement progressive delivery when our main integration point is this dumb, unreliable, inflexible network tissue.

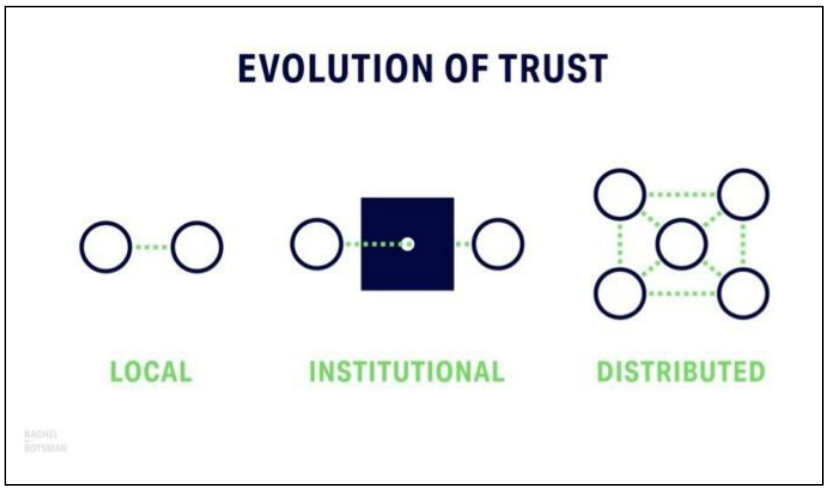


Fig. 2: Evolution of trust

Service mesh to the rescue

Service mesh is a relatively new software architecture pattern that allows us to insert smarts into this integration layer. Pioneered by Linkerd and further developed and enhanced by Istio, it involves infiltrating the network with a mesh of smart proxies that intercept all data paths. There's been some debate as to whether making the network smart goes against one of the main principles of microservices: smart endpoints, dumb pipes. But the general agreement seems to be that it's ok to make the network more intelligent as long as we don't put business logic in it (**Fig. 3**).

And well – it's hard to withstand the temptation. Smart networks provide some significant perks. Increased centralized ob-

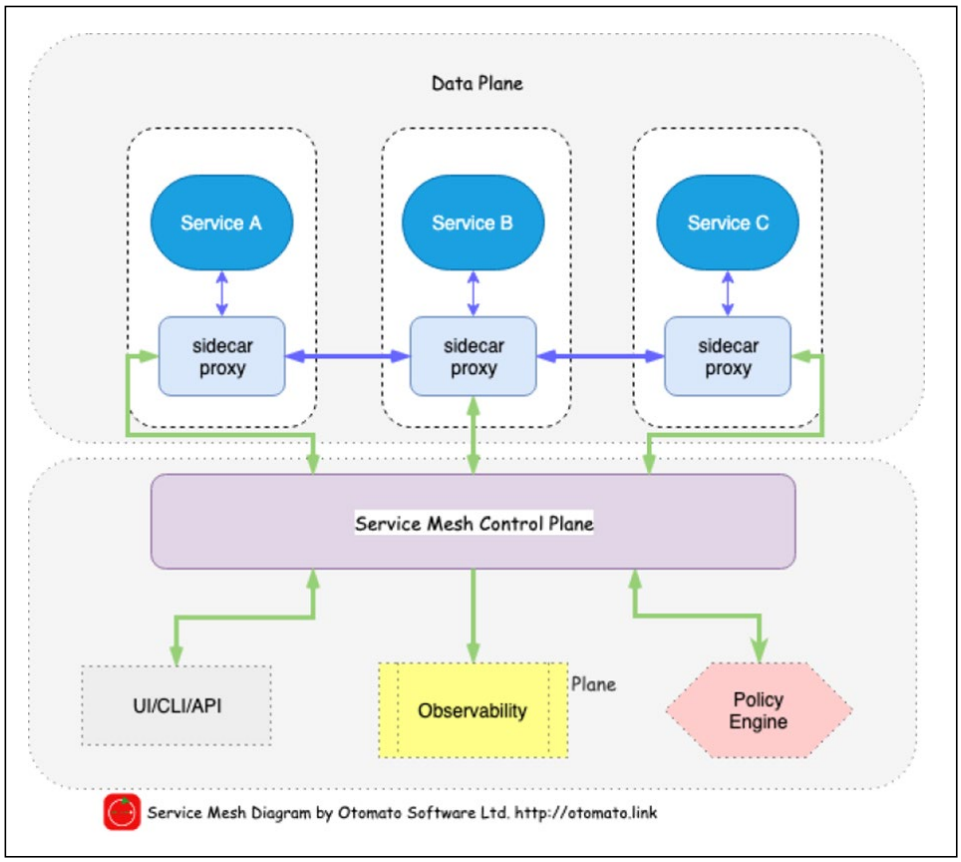


Fig. 3: Service mesh diagram. © Otomato Software Ltd. (http://otomato.link).

servability of the communication layer, distributed tracing, end-to-end mutual TLS, and yes – all the advanced intelligent routing and error injection techniques one might need for implementing true progressive delivery all across the stack. Some of these require virtually no effort. E.g. Istio, Linkerd and, of course, all cloud providers' offerings have an option to include the observability stack out of the box. At the click of a button everybody gets what they want: operators – visibility and control, developers – freedom from the fallacies of distributed systems, and the business – agility, stability, and continuous change.

Do I need a service mesh?

Service mesh sounds wonderful, doesn't it? But you know what came to my mind when I first heard of it? "Oh, gosh, we haven't passed through all nine circles of Kubernetes yet, what do we need another level of complexity for?" A similar emotion is quite well illustrated on twitter by Sam Kottler (**Fig. 4**).

Fig. 4: Tweet by Sam Kottler.

And I've talked to quite a bunch of folks sharing the same sentiment. Yes, a while later, when we dove deeper into the technology we've found a number of use cases where Istio has provided tremendous value. Especially in all that pertains to progressive delivery. But that may be because effective software delivery is our main focus. I'm sure security and compliance folks will love meshes for all the different reasons.

So do you need a service mesh? I'll give you the usual, annoying "it depends". You will probably benefit from it if you're running a service-oriented system but have hard time reaping its benefits: resilience, scalability and continuous delivery. If your bottlenecks are somewhere else (and they often are) – service mesh will only make things more complex. The mesh takes some control from the hands of developers and puts it in the hands of operators. So if you're trying to run a no-ops kind of shop, you may be better off sticking with libraries (like Finagle or Hystrix) instead. And certainly ask yourself if you're happy with your organization's ability to release changes to production and run experiments. If you feel like you should be doing better – service mesh may be just the way to achieve canaries, a/b testing or error injection and allow for stress-free releases. Service meshes are all the buzz. Our Istio workshop at DevOpsCon Berlin was sold out pretty fast, but if you'd like to see Istio in action – we can probably arrange running a workshop at DevOpsCon Munich later this year.

See you soon at the conference and happy progressive delivering.

Anton Weiss, the co-owner of Otomato Software technological consulting, has got over fifteen years of experience in cutting-edge technologies. He's an expert in technical coaching and the initiator and co-author of the first Israeli DevOps certification course.

Development, training and deployment of neural networks

Neural networks with PyTorch

PyTorch [1] is currently one of the most popular frameworks for the development and training of neural networks. It is characterized above all by its high flexibility and the ability to use standard Python debuggers. And you don't have to compromise on the training performance.

by Chi Nhan Nguyen

Because of the features mentioned above, PyTorch is popular above all with deep learning researchers and Natural Language Processing (NLP) developers. Significant innovations were also introduced in the last version, the first official release 1.0, in the area of integration and deployment as well.

Tensors

The elementary data structure for representing and processing data in PyTorch is *torch.Tensor*, The mathematical term tensor stands for a generalization of vectors and matrices. Tensors in the form of multidimensional arrays are implemented in PyTorch. Here a vector is nothing more than a one-dimensional tensor (or a tensor with rank 1) the elements of which can be numbers of a certain data type (such as *torch.float64* or *torch.int32*) could be. A matrix is thus a two-dimensional tensor (rank 2) and a scalar is a zero-dimensional tensor (rank 0). Tensors of even higher dimensions do not have any special names (**Fig. 1**).

The interface for PyTorch tensors strongly relies on the design of multidimensional arrays in NumPy [2]. Like NumPy, PyTorch provides predefined methods which can be used to manipulate tensors and perform linear algebra operations. Some examples are shown in Listing 1.

The use of optimized libraries such as BLAS [3], LAPACK [4] and MKL [5] allows for high-performance execution of tensor operations on the CPU (especially with Intel processors). In addition, PyTorch (unlike NumPy) also supports the execution of operations on NVIDIA graphic cards using the CUDA toolkit [6] and the CuDNN library [7]. Listing 2 shows an example of how to move tensor objects to the memory of the graphic card to perform optimized tensor operations there.

Since NumPy arrays are more or less considered to be standard data structures in the Python data science community, frequent conversion from PyTorch to NumPy and back is necessary in practice. These conversions can be done easily and efficiently (Listing 3) because the same memory area is shared and no copying of memory content is required.

Network modules

The torch.nn library contains many tools and predefined modules for generating neural network architectures. In practice, you define your own networks by deriving the abstract *torch.nn.Module* class, Listing 4 shows the implementation of a simple feed-forward network with a hidden layer and one *tanh* activation listed.

In the process, a network class of the abstract *nn.Module* class is derived. The *__init__()* and *forward()* methods must be defined at the same time. In *__init__ ()*, we need to instantiate and initiate all the required elements that make up the entire network. In our case, we generate three elements:

1. *fc1* - using *nn.Linear (input_dim, hidden_dim)* we generate a fully connected layer with an input dimension of *input_dim* and an output dimension of *hidden_dim*
2. *act1* - a Tanh activation function
3. *fc2* - another fully connected layer with an input dimension of *hidden_dim* and an output dimension of *output_dim*.

The sequence in *__init()__* basically does not matter, but for stylistic reasons you should generate them in the order in which they are called in the *forward()* method. The sequence in the *forward()* method is decisive for processing - this is where you determine the sequences of the forward run. At this point, you can even build in all kinds of conditional queries and branches, since a calculation graph is dynamically

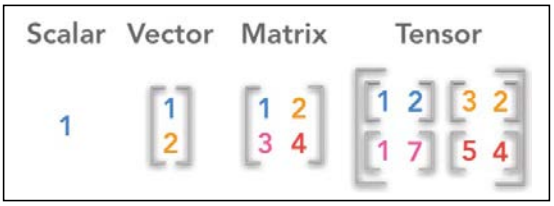


Figure 1: Tensors

generated on each run (Listing 5). This is useful if for example you want to work with varying batch sizes or experiment with complex branches. In particular, the processing of sequences of different lengths as input - as is often the case with many NLP problems - is much easier to realize with dynamic graphs than with static ones.

"Autograd" and dynamic graphs

PyTorch uses the *torch.autograd* package to dynamically generate a directed acyclic graph (DAG) on each forward run. In contrast, in the case of static generation, the graph is completely constructed initially and is then no longer changed. The static graph is filled and executed at each iteration with the new data. Dynamic graphs have some advantages in terms of flexibility, as I had already explained in the previous section. The disadvantages concern optimization capabilities, distributed (parallel) training and deployment of the models.

Through the definition of the forward path, *torch.autograd* generates a graph; the nodes of the graph represent the tensors and the edges represent the elementary tensor operations. With the help of this information, the gradients of all tensors can be determined automatically at runtime and thus back

Listing 1

```
# Generation of a one-dimensional tensor with
# 8 (uninitialized) elements (float32)
x = torch.Tensor(8)
x.double()  # Conversion to float64 tensor
x.int()     # Conversion to int32 data type

# 2D long tensor preinitialized with zeros
x = torch.zeros([2, 2])

# 2D long tensor preinitialized with ones
# and subsequent conversion to int64
y = torch.ones([2, 3]).long()

# Merge two tensors along dimension 1
x = torch.cat([x, y], 1)

x.sum()  # Sum of all elements
x.mean() # Average of all elements

# Matrix multiplication
x.mm(y)

# Transpose
x.t()

# Inner product of two tensors
torch.dot(x, y)

# Calculates intrinsic values and vectors
torch.eig (x)

# Returns tensor with the sine of the elements
torch.sin(x)
```

Listing 2

```
# 1D Tensors
x = torch.ones(1)
y = torch.zeros(1)

# Move tensors to the GPU memory
x = x.cuda()
y = y.cuda()

# or:
device = torch.device("cuda")
x = x.to(device)
y = y.to(device)

# The addition operation is now performed on the GPU
x + y  # like torch.add(x, y)
# Copy back to the CPU
x = x.cpu()
y = y.cpu()
```

Listing 3

```
# Conversion to NumPy
x = x.numpy()

# Conversion back as PyTorch tensor
y = torch.from_numpy(x)
# y now points to the same memory area as x
# a change of y changes x at the same time
```

Listing 4

```
import torch
import torch.nn as nn

class Net(nn.Module):

  def __init__(self, input_dim, hidden_dim, output_dim):
    super(Net, self).__init__()
    # Here you create instances of all submodules of the network

    self.fc1 = nn.Linear(input_dim, hidden_dim)
    self.act1 = nn.Tanh()
    self.fc2 = nn.Linear(hidden_dim, output_dim)

  def forward(self, x):
    # Here you define the forward sequence
    # torch.autograd dynamically generates a graph on each run

    x = self.fc1(x)
    x = self.act1(x)
    x = self.fc2(x)
    return x
```

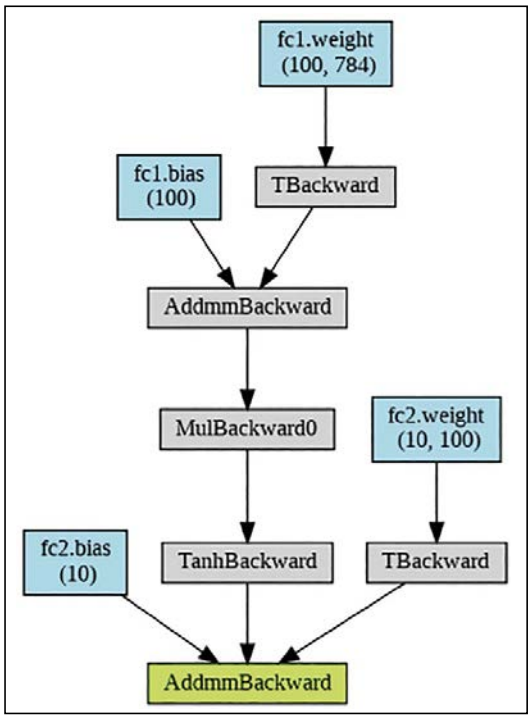


Figure 2: Example of a DAG generated using „torch.autograd“

propagation can be carried out efficiently. An example graph is shown in **Figure 2**.

Debugger

The biggest advantage in the implementation of dynamic graphs rather than static graphs is the possibility of debugging. Within the *forward ()* method, you can make any printout or set breakpoints, which in turn can be analyzed, for example with the help of the *pdb* standard debugger. This feature is not readily available with static graphs, because you do not have direct access to the objects of the network at runtime.

Training

The torchvision package contains many useful tools, pretrained models and datasets for image processing. In Listing 6, the *FashionMNIST* dataset is loaded. It consists of a training and validation dataset containing 60,000 or 10,000 icons from the fashion industry.

The icons are grayscale images of 28x28 pixels divided into ten classes (0-9): 0. T-Shirt, 1. Trousers, 2. Sweater, 3. Dress, 4. Coat, 5. Sandals, 6. Shirt, 7. Sneakers, 8. Bag, 9. Ankle Boot (**Fig. 3**). The *dataset* class represents a dataset that can be partitioned arbitrarily and applied to various transformations. In this example, the NumPy arrays are converted to Torch tensors. In addition though, quite a few other transformations are offered to augment and normalize the data (such as snippets, rotations, reflections, etc.). *Data Loader* is an iterator class that generates individual batches of the dataset and loads them into memory, so you do not have to completely load large data sets. Optionally, you can choose whether you want to start multiple threads (*num_workers*) or whether the dataset should be remixed before each epoch (*shuffle*).

In Listing 7, we first generate an instance of our model and transfer the entire graph to the GPU. PyTorch offers various loss functions [8] and optimization algorithms [9]. For a multi-label classification problem, *CrossEntropyLoss ()* can be for example chosen as a loss function, and Stochastic Gradient Descent (SGD) as the optimization algorithm. The parameters of the network that should be optimized are transferred to the *SGD()* method. An optional parameter is the learning rate (*lr*).

The *train()* function in Listing 8 performs a training iteration. At the beginning, all gradients of the network graph are reset (*zero_grad()*). A forward run through the graph is performed afterwards. The loss value is determined by comparing the network output and the label tensor. The gradients are calculated using *backward()* and finally, the weights of the network are updated through back propagation using *optimizer.step()*. The *valid()* validation iteration is a variant of the training iteration, and all back propagation work steps are omitted in the process.

The full iteration over multiple epochs is shown in Listing 9. For the application of the *Net()* feed forward model, the icon tensors with the dimensions *(batch_size, 1, 28, 28)* must be transformed to *(batch_size, 784)*. The call of train_loop() should thus be executed with the *'flatten'* argument:

```
train_loop(model, trainloader, validloader, 10, 200, 'flatten').
```

Listing 5

```
class Net(nn.Module):
  ...
  def forward(self, x, a, b):

    x = self.fc1(x)

    # Conditional application of the activation function
    if a > b:
      x = self.act1(x)

    x = self.fc2(x)
    return x
```

Listing 6

```
transform = transforms.Compose([transforms.ToTensor()])
# more examples of transformations:
# transforms.RandomSizedCrop()
# transforms.RandomHorizontalFlip()
# transforms.Normalize()

# Download and load the training data set (60,000)
trainset = datasets.FashionMNIST('./FashionMNIST/', download=True, train=True,
                transform=transform)  # Object from torch.utils.data.class dataset
trainloader = DataLoader(trainset, batch_size=batch_size, shuffle=True, num_
workers=4)

# Download and load the validation dataset (10,000)
validset = datasets.FashionMNIST('./FashionMNIST/', download=True, train=False,
                transform=transform)  # Object from torch.utils.data.class dataset
validloader = DataLoader(validset, batch_size=batch_size, shuffle=True, num_
                                                                workers=4)
```

Save and load trained weights

To be able to use the models later for inference in an application, it is possible to save the trained weights in the form of serialized Python Dictionary objects. The Python package pickle is used for this. If you want to continue training the model later, you should also save the last state of the optimizer. Listing 9 stores the model weights and current state of the optimizer after each epoch. Listing 0 shows how one of these pickle files can be loaded.

Network modules

PyTorch offers many more predefined modules for building Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), or even more complex architectures such as encoder-decoder systems. The *Net()* model could for example be extended with a dropout layer (Listing 11).

Listing 12 shows an example of a CNN consisting of two Convolutional Layers with Batch Normalization, each with ReLU activation and a Max Pooling Layer. The training call could look like this:

Listing 7

```
import torch.optim as optim
# Use the GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Define model
input_dim = 784
hidden_dim = 100
output_dim = 10
model = Net(input_dim, hidden_dim, output_dim)
model.to(device)  # move all elements of the graph to the current device

# Define optimizer algorithm and associate with model parameters
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Define loss function: CrossEntropy for classification
loss_function = nn.CrossEntropyLoss()
```

Listing 8

```
# Training a batch
def train(model, images, label, train=True):
  if train:
    model.zero_grad() # Reset the gradients

  x_out = model(images)
  loss = loss_function(x_out, label)  # Determine loss value

  if train:
    loss.backward()  # Calculate all gradients
    optimizer.step() # Update the weights
  return loss

# Validation: Only forward run without back propagation
def valid(model, images, label):
  return train(model, images, label, train=False)
```

```
model = CNN(10).to(device)
optimizer = optim.SGD(model.parameters(), lr=0.01)
train_loop(model, trainloader, validloader, 10, 200, None)
```

An example of an LSTM network which has been optimized using the Adam Optimizer is shown in Listing 13. The pixels of the images from the FashionMNIST dataset are interpreted as sequences of 28 elements, each with 28 features and preprocessed accordingly.

Listing 9

```
import numpy as np

def train_loop(model, trainloader, validloader=None, num_epochs = 20, print_
                    every = 200, input_mode='flatten', save_checkpoints=False):
  for epoch in range(num_epochs):

    # Training loop
    train_losses = []
    for i, (images, labels) in enumerate(trainloader):
      images = images.to(device)
      if input_mode == 'flatten':
        images = images.view(images.size(0), -1)  # flattening of the Image
      elif input_mode == 'sequence':
        images = images.view(images.size(0), 28, 28)  # Sequence of 28
                                                      elements with 28 features

      labels = labels.to(device)
      loss = train(model, images, labels)
      train_losses.append(loss.item())
      if (i+1) % print_every == 0:
        print('Training', epoch+1, i+1, loss.item())

    if validloader is None:
      continue

    # Validation loop
    val_losses = []
    for i, (images, labels) in enumerate(validloader):
      images = images.to(device)
      if input_mode == 'flatten':
        images = images.view(images.size(0), -1)  # flattening of the Image
      elif input_mode == 'sequence':
        images = images.view(images.size(0), 28, 28)  # Sequence of 28 elements
                                                      with 28 features
      labels = labels.to(device)
      loss = valid(model, images, labels)
      val_losses.append(loss.item())
      if (i+1) % print_every == 0:
        print('Validation', epoch+1, i+1, loss.item())

    print('--- Epoch, Train-Loss, Valid-Loss:', epoch, np.mean(train_losses),
                                                      np.mean(val_losses))

    if save_checkpoints:
      model_filename = 'checkpoint_ep'+str(epoch+1)+'.pth'
      torch.save({
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
      },  model_filename)
```

The torchvision package also allows you to load known architectures or even pre-trained models that you can use as a basis for your own applications or for transfer learning. For example, a pre-trained VGG model with 19 layers can be loaded as follows:

```
from torchvision import models
vgg = models.vgg19(pretrained=True)
```

Deployment

The integration of PyTorch models into applications has always been a challenge, as the opportunities to use the trained models in production systems had been relatively limited. One commonly used method is the development of a REST service, using flask [10], for example. This REST service can run locally or within a Docker image in the cloud. The three major providers of cloud services (AWS, GCE, Azure) now also offer predefined configurations with PyTorch.

An alternative is conversion to the ONNX format [11]. ONNX (Open Neural Network Exchange Format) is an open format for the exchange of neural network models, which are also supported by MxNet [12] and Caffe [13], for example. These are machine learning frameworks that are used productively by Amazon and Facebook. Listing 14 shows an example of how to export a trained model to the ONNX format.

Listing 10

```
model = Net(input_dim, hidden_dim, output_dim)

checkpoint = torch.load('checkpoint_ep2.pth')
model.load_state_dict(checkpoint['model_state_dict'])

optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
```

Listing 11

```
class Net(nn.Module):

  def __init__(self, input_dim, hidden_dim, output_dim):
    super(Net, self).__init__()

    self.fc1 = nn.Linear(input_dim, hidden_dim)
    self.dropout = nn.Dropout(0.5) # Dropout layer with probability 50 percent
    self.act1 = nn.Tanh()
    self.fc2 = nn.Linear(hidden_dim, output_dim)

  def forward(self, x):
    x = self.fc1(x)
    x = self.dropout(x) # Dropout after the first FC layer
    x = self.act1(x)
    x = self.fc2(x)

    return x
```

Listing 12

```
class CNN(nn.Module):

  def __init__(self, num_classes=10):
    super(CNN, self).__init__()

    self.layer1 = nn.Sequential(
      nn.Conv2d(1, 16, kernel_size=5, padding=2),
      nn.ReLU(),
      nn.MaxPool2d(2)
    )
    self.layer2 = nn.Sequential(
      nn.Conv2d(16, 32, kernel_size=5, padding=2),
      nn.ReLU(),
      nn.MaxPool2d(2))
    self.fc = nn.Linear(7*7*32, 10)

  def forward(self, x):
    out = self.layer1(x)
    out = self.layer2(out)
    out = out.view(out.size(0), -1)  # Flattening for FC input
    out = self.fc(out)
    return out
```

Listing 13

```
# Recurrent Neural Network
class RNN(nn.Module):
  def __init__(self, input_size, hidden_size, num_layers, num_classes):
    super(RNN, self).__init__()
    self.hidden_size = hidden_size
    self.num_layers = num_layers
    self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
    self.fc = nn.Linear(hidden_size, num_classes)

  def forward(self, x):
    # Initialize Hidden and Cell States
    h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(device)
    c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(device)

    out, _ = self.lstm(x, (h0, c0))

    out = self.fc(out[:, -1, :]) # last hidden state
    return out

sequence_length = 28
input_size = 28
hidden_size = 128
num_layers = 1

model = LSTM(input_size, hidden_size, num_layers, output_dim).to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
train_loop(model, trainloader, validloader, 10, 200, 'sequence')
```

TorchScript and C++

As of version 1.0, PyTorch has also been offering the possibility to save models in LLVM-IR format [14]. This can be done completely independently of Python. The tool for this is TorchScript, which implements its own JIT compiler and special optimizations (static data types, optimized implementation of tensor operations).

You can create the TorchScript format in two ways. For one, by tracing an existing PyTorch model (Listing 15) or through direct implementation as a script module (Listing 16). In script mode, an optimized static graph is generated. This not only offers the advantages for deployment mentioned earlier, but could, also be used for distributed training, for example.

Listing 14

```
model = CNN(output_dim)

# Any input sensor for tracing
dummy_input = torch.randn(1, 1, 28, 28)

# Conversion to ONNX is done by tracing a dummy input
torch.onnx.export(model, dummy_input, "onnx_model_name.onnx")
```

Listing 15

```
# Any input sensor for tracing
dummy_input = torch.randn(1, 1, 28, 28)

traced_model = torch.jit.trace(model, dummy_input)
traced_model.save('jit_traced_model.pth')
```

Listing 16

```
from torch.jit import trace

class Net_script(torch.jit.ScriptModule):

  def __init__(self, input_dim, hidden_dim, output_dim):
    super(Net_script, self).__init__()

    self.fc1 = trace(nn.Linear(input_dim, hidden_dim), torch.randn(1, 784))
    self.fc2 = trace(nn.Linear(hidden_dim, output_dim), torch.randn(1, 100))

  @torch.jit.script_method
  def forward(self, x):
    x = self.fc1(x)
    x = torch.tanh(x)
    x = self.fc2(x)

    return x

model = Net_script(input_dim, hidden_dim, output_dim)

model.save('jit_model.pth')
```

The TorchScript model can now be integrated into any C ++ application using the C ++ front-end library (LibTorch). This enables high-performance execution of the inference independent of Python and in many different production environments, such as on mobile devices.

Conclusion

With PyTorch, you can efficiently and elegantly develop and train both simple and very complex neural networks. By implementing dynamic graphs, you can experiment with very flexible architectures and use standard debugging tools with no problem. The seamless connection to Python allows for speedy development of prototypes. These features currently make PyTorch the most popular framework for researchers and experiment-happy developers. The latest version also provides the ability to integrate PyTorch models into C ++ applications to achieve better integration in production systems. This is significant progress when compared to the earlier versions. However, other frameworks, especially TensorFlow [15], still have a clear lead in this category. With TF Extended (TFX), TF Serving, and TF Lite, the Google framework provides much more application-friendly and robust tools for creating production-ready models. It will be interesting to see what new developments in this area we will see from PyTorch.

Chi Nhan Nguyen works as a Senior Data Scientist Consultant at itemis. As a high-energy physicist, he has spent years doing research on facilities such as CERN, Fermilab, and DESY and has implemented machine learning techniques for data analysis. He is currently working as a data scientist in the private sector, where he focuses on the development of probabilistic models and methods of deep learning.

References

[1] https://pytorch.org
[2] http://www.numpy.org
[3] http://www.netlib.org/blas/
[4] http://www.netlib.org/lapack/
[5] https://software.intel.com/mkl/
[6] https://developer.nvidia.com/cuda-toolkit
[7] https://developer.nvidia.com/cudnn
[8] https://pytorch.org/docs/stable/nn.html#loss-functions
[9] https://pytorch.org/docs/stable/optim.html
[10] http://flask.pocoo.org
[11] https://onnx.ai
[12] https://mxnet.apache.org
[13] https://caffe.berkeleyvision.org
[14] https://llvm.org
[15] https://www.tensorflow.org

© irin-k/Shutterstock.com

The perfect addition?

Machine learning with the Apache Kafka ecosystem

Machine Learning (ML) allows applications to obtain hidden knowledge without the need to explicitly program what needs to be considered in the process of knowledge discovery. This way, unstructured data can be analyzed, image and speech recognition can be improved and well-informed decisions can be made. In this article we will in particular discuss new trends and innovations surrounding Apache Kafka and Machine Learning [1].

by Kai Wähner

Machine Learning and the Apache Kafka Ecosystem are an excellent combination for training and deploying scalable analytical models. Here, Kafka becomes the central nervous system in the ML architecture, feeding analytical models with data, training them, using them for forecasting and monitoring them. This yields enormous advantages:

- Data pipelines are simplified
- The implementation of analytical models is separated from their maintenance
- Real time or batch can be used as needed
- Analytical models can be applied in a high-performance, scalable and business-critical environment

Since customers expect information in real time today, the challenge for companies is to respond to customer inquiries and react to critical moments before it's too late. Batch processing is no longer sufficient here; the reaction must be immediate, or better yet - proactive. This is the only way you can stand out from the competition. Machine learning can improve existing business processes and automate data-driven decisions. Examples of such use cases include fraud de-

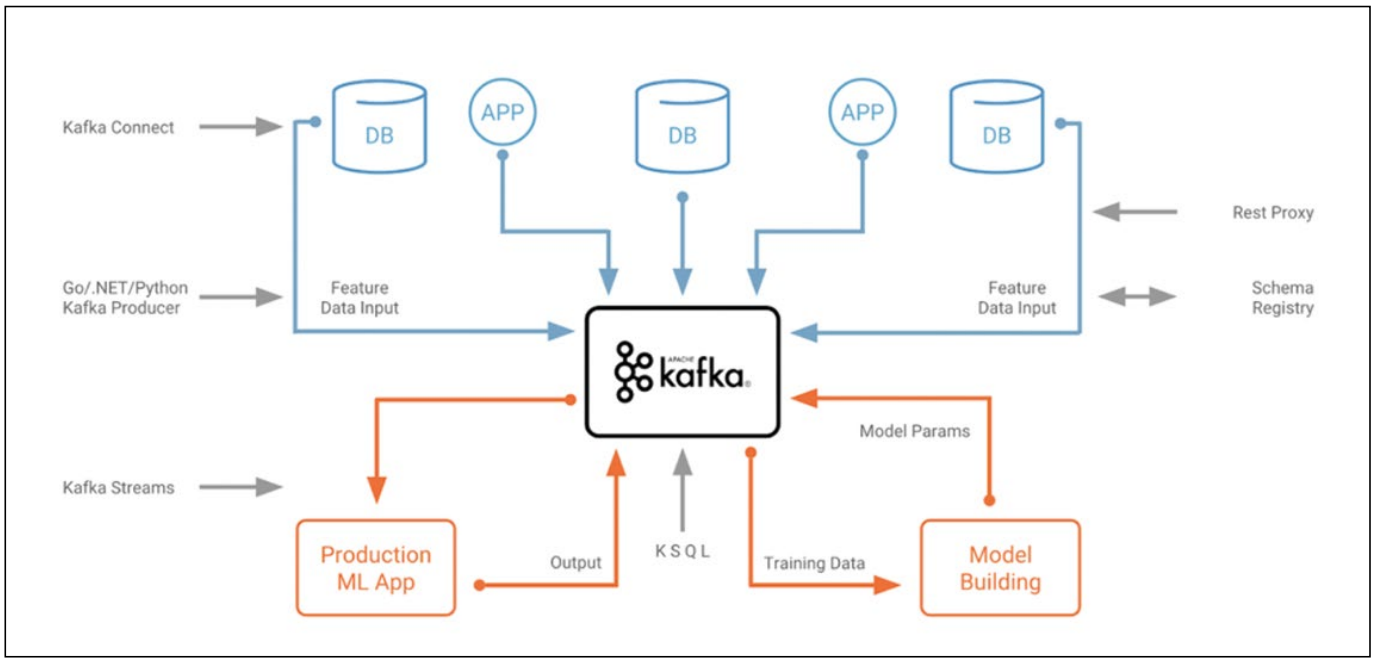


Fig. 1: An example of mission-critical real-time applications

tection, cross-selling or the predictive maintenance of IoT devices. The architecture of such business-critical real-time applications which use the Apache Kafka Ecosystem as a scalable and reliable central nervous system for your data can be presented as shown in **Figure 1.**

Deployment of analytical models

Generally speaking, a machine learning life cycle consists of two parts:

- **Training the model:** In this step, we load historic data into an algorithm to learn patterns from the past. The result is an analytical model.

- **Creating forecasts/predictions:** In this step, we use the analytical model to prepare forecasts about new events based on the learned pattern.

Machine learning is a continuous process in which the analytical model is constantly improved and redeployed over time. Forecasts can be deployed within an application or microservice in various ways. One possibility is to embed an analytical model directly into a stream processing application, such as an application that uses Kafka streams. For example, you can use TensorFlow for Java API to load models and apply them in real time (**Fig. 2**).

Alternatively, you can deploy the analytical models to a dedicated model server (such as TensorFlow Serving) and use remote procedure calls (RPC) from the streaming application for the service (for example, with HTTP or gRPC) (**Fig. 3**).

Both options have their advantages and disadvantages. The advantages of a dedicated mail server include:

- Easy integration into existing technologies and business processes
- The concept is easier to understand if you're coming from the „non-streaming world"
- Later migration to „real" streaming is possible
- Model management incorporates the use of various models, including versioning, A/B testing, etc.

On the other hand, here are some disadvantages of a dedicated model server:

- Often linked to specific ML technologies
- Often linked to a specific cloud provider (vendor lock-in)
- Higher latency
- More complex security concepts (remote communication via firewalls and authorization management)

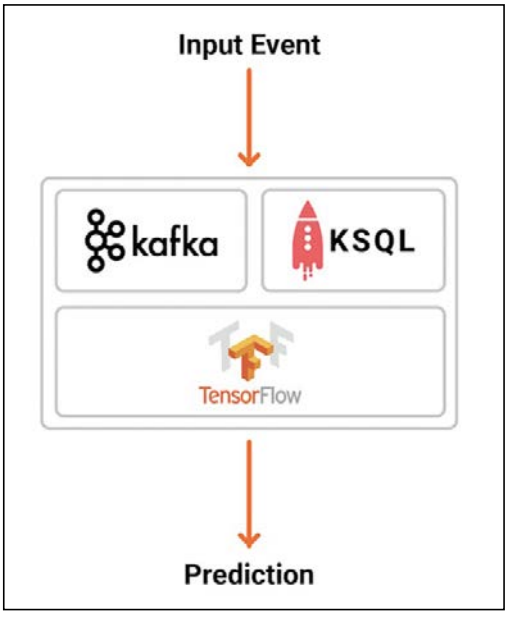


Fig. 2: Example of the use of TensorFlow for Java API

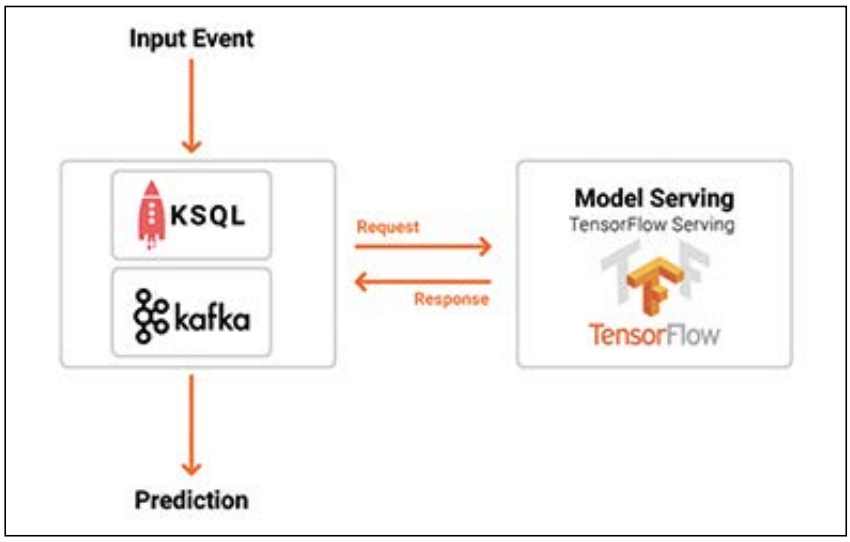


Fig. 3: Deployment of analytical models on a dedicated model server

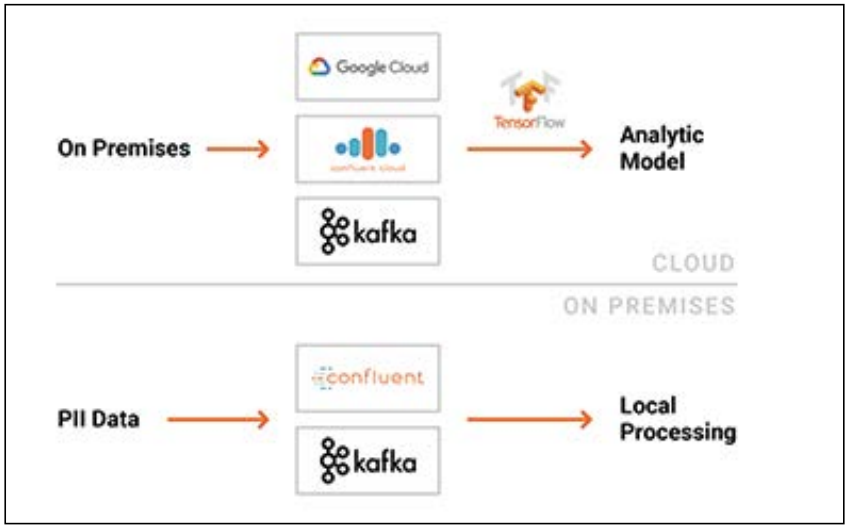


Fig. 4: Example of a hybrid Kafka infrastructure

- No offline inference (devices, edge processing, etc.)
- Combines the availability, scalability, and latency/throughput of your stream processing application with the SLAs of the RPC interface
- Side effects (e.g. in the event of a failure during network issues) that are not covered by Kafka processing (such as exactly-once processing)

How analytical models are deployed is a matter of individual decisions for each scenario, including latency and security considerations. Some of the trends discussed below, such as hybrid architectures or low-latency requirements, also require model deployment considerations: For example, do you use models locally in edge components such as sensors or mobile devices to process personal information, or integrate external AutoML services to take advantage of the benefits and scalability of cloud services? To make the best choice for your use case and your architecture, it is very important to understand both options and the associated trade-offs.

Model deployment in Kafka applications

Kafka applications are event-based and use event streams to continuously process incoming data. If you use Kafka, you can natively embed an analytical model in a Kafka Streams or KSQL application. There are several examples of Kafka Streams microservices that incorporate models natively created with TensorFlow, H2O, or Deeplearning4j [2].

Due to reasons of an architectural, safety or organizational nature, it is not always possible or feasible to embed analytical models directly. You can also use RPC to perform model inferences from your Kafka application (while considering the pros and cons described above).

Hybrid cloud and on-premise architectures

Hybrid cloud architectures are often the technology of choice for machine learning infrastructure. Training can be conducted with large amounts of historic data in the public cloud or in a central data lake within an in-house data center. The model deployment needed to prepare forecasts can be done anywhere.

In many scenarios it makes perfect sense to use the scalability and adaptability of public clouds. So for example, new large calculation instances can be created to train a neural network for a few days, after which they can be stopped easily. Pay-as-you-go is a perfect model, especially for deep learning.

In the cloud, you can use specific processing units that would be costly to run in your own data center and often go unused. For example, Google's TPU (Tensor Processing Unit), an application-specific integrated circuit (ASIC) engineered from the ground up for machine learning, is a specific processor designed for deep learning only. TPUs can do one thing well: Matrix multiplication - the heart of deep learning - for the training of neural networks.

If you need to keep data out of the public cloud, or if you want to build your own ML infrastructure for larger teams or departments in your own data center, you can buy specially designed hardware/software combinations for deep learning, such as the Nvidia DGX platforms.

Wherever you need it, forecasts can be made using the analytical model regardless of model training: in the public cloud, on-site in your data centers or on edge devices such as the Internet of Things (IoT) or mobile devices. However, edge devices often have higher latency, limited bandwidth, or poor connectivity.

Building hybrid cloud architectures with Kafka

Apache Kafka allows you to build a cloud-independent infrastructure, including both multi-cloud and hybrid architec-

```
CREATE STREAM car_sensor AS
   SELECT car_id, event_id, car_model_id, sensor_input
   FROM car_sensor c
   LEFT JOIN car_models m ON c.car_model_id = m.car_model_id
   WHERE m.car_model_type ='Audi_A8';
```

gistfile1.txt hosted with ♥ by GitHub view raw

Fig. 5: Example of filtering sensor data for further processing or analysis in real time

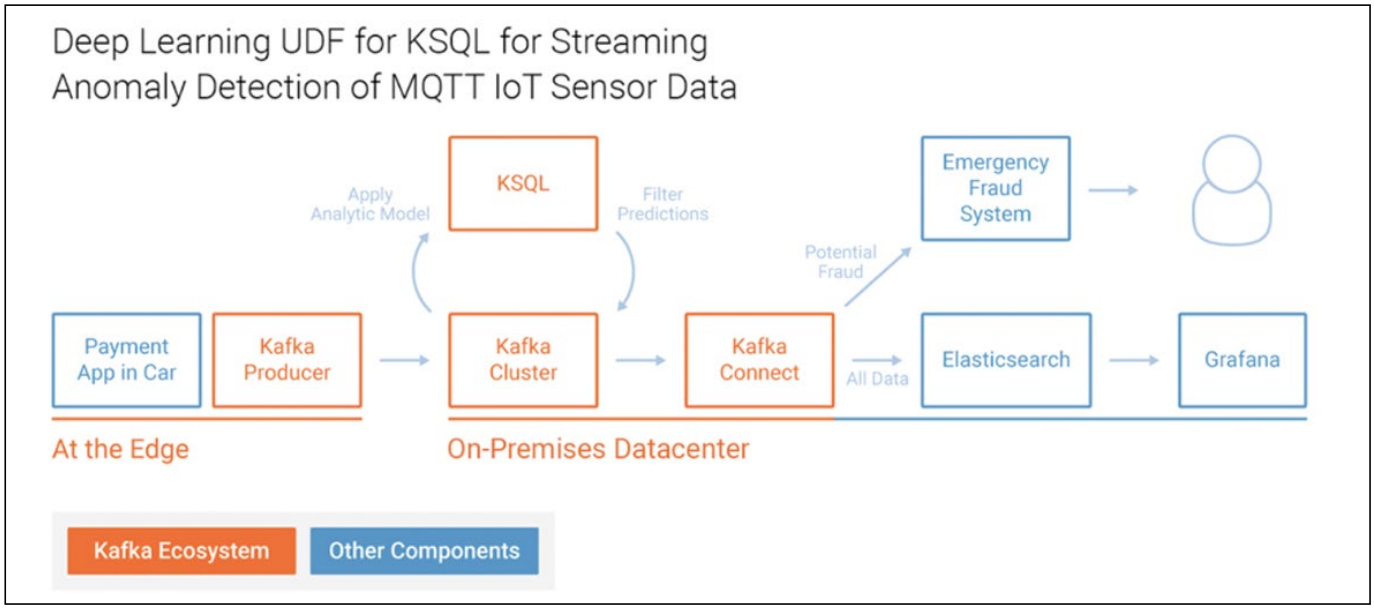


Fig. 6: Application of a neural network for sensor analysis

tures, without being tied to specific cloud APIs or proprietary products. **Figure 4** shows an example of a hybrid Kafka infrastructure used to train and deploy analytical models.

Apache Kafka components such as Kafka Connect can be used for data feed, while Kafka Streams or KSQL are good for preprocessing data. Model deployment can also be done within a Kafka client like Java, .NET, Python, Kafka Streams or KSQL. Quite often, the entire monitoring of the ML infrastructure is carried out with Apache Kafka. This includes technical metrics such as latency and project-related information such as model accuracy.

Data streams frequently come from other data centers or clouds. A common scenario is to use a Kafka replication tool, such as MirrorMaker or Confluent Replicator, to replicate the data from the source Kafka clusters in a reliable and scalable manner to the analysis environment.

The development, configuration and operation of a reliable and scalable Kafka cluster always requires a solid understanding of distributed systems and experience in dealing with them. Therefore, you can alternatively use a cloud service like Confluent Cloud, which offers Kafka-as-a-Service. Here, only the Kafka client (such as Kafka's Java API, Kafka Streams or KSQL) is created by the developer and the Kafka server side is used as an API. In in-house data centers, tools such as Confluent Operator for operating on Kubernetes distributions and Confluent Control Center for the monitoring, control and management of Kafka clusters are helpful.

Streaming analysis in real time

A primary requirement in many use cases is to process information while it is still current and relevant. This is relevant for all areas of an ML infrastructure:

- Training and optimization of analytical models with up-to-date information
- Forecasting new events in real time (often within milliseconds or seconds)
- Monitoring of the entire infrastructure for model accuracy, infrastructure errors, etc.
- Security-relevant tracking information such as access control, auditing or origin

The simplest and most reliable way to process data in a timely manner is to create real-time streaming analyzes native to Apache Kafka using Kafka clients such as Java, .NET, Go, or Python.

In addition to using a Kafka Client API (i.e. Kafka Producer and Consumer), you should also consider Kafka Streams, the stream processing framework for Apache Kafka. This is a Java library that enables simple and complex stream processing within Java applications.

For those of you who do not want to write Java or Scala or cannot do so, there is KSQL, the streaming SQL engine for Apache Kafka. It can be used to create stream-processing applications that are expressed in SQL. KSQL is available as an open source download at [3].

```
1   SELECT car_id, event_id, ANOMALY(sensor_input) FROM car_sensor;
```

gistfile1.txt hosted with ♥ by GitHub view raw

Fig. 7: KSQL query using ANOMALY UDF

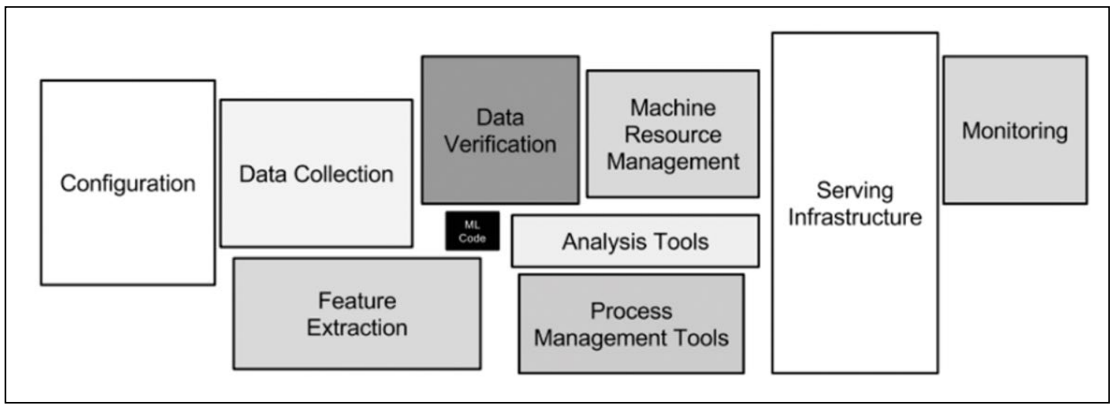


Fig. 8: The deployment of an analytical model is no uncomplicated matter.

KSQL and machine learning for preprocessing and model deployment

Processing streaming data with KSQL makes data preparation for machine learning easy and scalable. SQL statements can be used to perform filtering, enrichment, transformation, feature engineering or other tasks. **Figure 5** shows only one example on how to filter sensor data from multiple vehicle types for a particular vehicle model to be used for further processing or analysis in real time.

ML models can easily be embedded in KSQL by creating a user-defined function (UDF). In the detailed example "Deep Learning UDF for KSQL for Streaming Anomaly Detection of MQTT IoT Sensor Data"[4], a neural network is used for sensor analysis, or more specifically as an auto-encoder, to detect anomalies (**Fig. 6**).

In this example, KSQL continuously processes millions of events from networked vehicles through MQTT integration to the Kafka cluster. MQTT is a publish/subscribe messaging protocol which was developed for restricted devices and unreliable networks. It is often used in combination with Apache Kafka to integrate IoT devices with the rest of the company. The auto-encoder is used for predictive maintenance.

Real-time analysis in combination with this vehicle sensor data makes it possible to send anomalies to a warning or emergency system to be able to react before the engine fails. Other use cases for intelligent networked car include optimized route guidance and logistics planning, the sale of new features and functions for a better digital driving experience, and loyalty programs that correspond directly with restaurants and roadside businesses.

Even if a KSQL UDF requires a bit of code, it only needs to be written by the developer once. After that, the end user can easily use the UDF within their KSQL statements like any other integrated function.

In **Figure 7**, you can see the KSQL query from our example using the ANOMALY UDF, which uses the TensorFlow model in the background.

Depending on preferences and requirements, both KSQL and Kafka streams are perfect for ML infrastructures when it comes to preprocessing streaming data and performing model inferences. KSQL lowers the entry-level barrier and allows you to implement streaming applications with simple SQL statements, rather than having to write source code.

Scalable and flexible platforms for machine learning

Technology giants are typically several years ahead of traditional companies. They have already built what others (have to) build today or tomorrow. Machine learning platforms are no exception here.

The "Hidden Technical Debt in Machine Learning Systems" paper [5] explains why creating and deploying an analytical model is far more complicated than just writing some machine learning code using technologies like Python and TensorFlow. You also have to take care of things like data collection, feature extraction, infrastructure deployment, monitoring and other tasks - all this with the help of a scalable and reliable infrastructure (**Fig. 8**).

In addition, technology giants are showing that a single machine learning/deep learning framework such as TensorFlow is not sufficient for their use cases, and that machine learning is a fast-growing field. A flexible ML architecture must support different technologies and frameworks. It also needs to be fully scalable and reliable when used for essential business processes. That's why many technology giants have developed their own machine learning platforms, such as Michelangelo by Uber, Meson by Netflix, and the fraud detection platform by PayPal. These platforms enable businesses to build and monitor powerful, scalable analytical models, while remaining flexible in choosing the right technology for each application.

Apache Kafka as a central nervous system

One of the reasons for Apache Kafka's success is its rapid adoption and acceptance by many technology companies. Almost all major Silicon Valley companies such as LinkedIn,

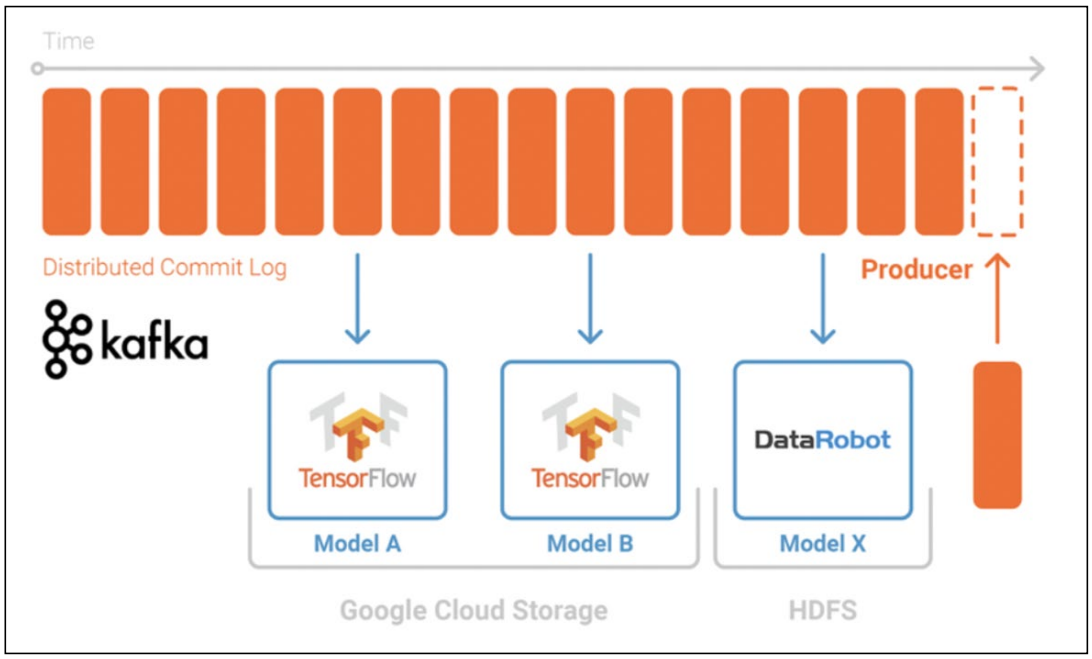


Fig. 9: Data can be consumed and processed again and again from the distributed commit log

Netflix, Uber or eBay blog and speak of their use of Kafka as an event-driven central nervous system for their business-critical applications. Many of them concentrate on the distributed streaming platform for messaging, but components such as Kafka Connect, Kafka Streams, REST Proxy, Schema Registry and KSQL are being used more and more.

As I had already explained, Kafka is a logical addition to an ML platform: Training, monitoring, deployment, inferencing, configuration, A/B testing, etc. That's probably why Uber, Netflix, and many others already use Kafka as a key component in their machine learning infrastructure.

Kafka makes deploying machine learning tasks easy and uncomplicated, without the need for another large data cluster. And yet it is flexible in terms of integration with other systems. If you rely on Hadoop for your batch data processing or or want to do your ML Processing using Spark or AWS Sagemaker for example, all you need to do is connect them to Kafka via Kafka Connect. With the Kafka ecosystem as the basis of an ML infrastructure, nobody is forced to use only one specific technology (**Fig. 9**).

You can use different technologies for analytical model training, deploy models for any Kafka native or external client application, develop a central monitoring and auditing system, and finally be prepared and open for future innovations in machine learning. Everything is scalable, reliable and fault-tolerant, since the decoupled systems are loosely linked to Apache Kafka as the nervous system.

In addition to TensorFlow, the above example also uses the DataRobot AutoML framework to automatically train different analytical models and deploy the model with optimal accuracy. AutoML is an emerging field, as it automates many complex steps in model training such as hyperparameter tuning or algorithm selection. In terms of integration with the Kafka ecosystem, there are no differences with other ML frameworks, both work well together.

Conclusion

Apache Kafka can be viewed as the key to a flexible and sustainable infrastructure for modern machine learning. The ecosystem surrounding Apache Kafka is the perfect complement to an ML architecture. Choose the components you need to build a scalable, reliable platform that is independent of any specific on-premise/cloud infrastructure or ML technology.

You can build an integration pipeline for modeling training and use the analytical models for real-time forecasting and monitoring within Kafka applications. This does not change with new trends such as hybrid architectures or AutoML. Quite the contrary: with Kafka as the central but distributed and scalable layer, you remain flexible and ready for new technologies and concepts of the future.

Kai Wähner is an IT Consultant at Maiborn Wolff et al. His focus is on JEE, SOA and Cloud Computing. He also writes about his experiences with new technologies on his blog.

www.kai-waehner.de/blog **kai.waehner@mwea.de**

@KaiWaehner

References

[1] More information on Apache Kafka and Machine Learning: https://www.confluent.io/blog/build-deploy-scalable-machine-learning-production-apache-kafka/

[2] Examples of Kafka Streams Microservices: https://github.com/kaiwaehner/kafka-streams-machine-learning-examples

[3] https://hub.docker.com/r/confluentinc/cp-ksql-cli/

[4] Deep Learning UDF for KSQL for Streaming Anomaly Detection of MQTT IoT Sensor Data: https://github.com/kaiwaehner/ksql-udf-deep-learning-mqtt-iot

[5] Paper Hidden Technical Debt in Machine Learning System: https://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems.pdf

Interview with Oz Tiram

Python: "We see a trend towards adding more and more typing support"

The Python programming language is enjoying increasing popularity. Python is especially popular in the field of machine learning. We talked to Oz Tiram, senior IT architect at noris network AG and speaker at the Python Summit 2019, about the reasons for the popularity of Python and the current state of Python language development.

Jax Magazine: In your workshop you will cover Python's Iterables, Iterators and Generators. Can you briefly explain what's special about it?

Oz Tiram: Well, nothing actually! Almost all programming languages have similar constructs, and if they don't have them built in, they are easy to implement. And that is the cool thing about Python – it comes with batteries included. That means, in order to get things done, you don't need to implement them. Any Python object that contains other objects is iterable. Obviously, other languages have this, too (it's no surprise, here).

Computer programs are there to do repetitive tasks for us, hence any programming language must have some way to repeat an action until there is no longer need to repeat it. One such way is to repeat an action on a set of items. Speaking a human language, this can be formulated as:

For each item in BagOfItems, do something with item.

In Python this sentence is written as:

```
>>> for item in BagOfItems:
...   do_something(item)
```

So, in this case *BagOfItems* is an iterable. In the workshop, we look at how *BagOfItems* is built and how Python knows when to stop when *BagOfItems* has no more items in it.

Iterators are a special kind of *BagOfItems*, in which we can access items one at a time. Usually, when we start a task we would like to finish it, but sometimes we can't or don't want to process all items at once. For example, if *BagOfItems* was *BagOfBalls* and a human needed to find all the red balls in this bag, they could take a ball out of the bag, inspect its color and put it in a basket of red balls if the color is red, while balls of another color would go in another basket.

A human can stop this task, to take a coffee break, for example, and then carry on afterwards. The bag is an iterator because we can take one item at a time. An iterable only allows all of the balls to be inspected without pausing. So, while doing a task multiple times, real world tasks sometime require us to take a break.

Put simply, a **generator** is a special type of function that can help us turn iterable items into an iterator. Generators can also generate iterators without having a local data store, which makes them useful for processing large amounts of data without consuming all the memory. Because generator functions are so useful, they have their own special keyword, which is *yield*. Generators are functions that do not have a *return* statement, instead they can have one or more yield statements in their body:

Portrait

Oz Tiram studied Applied Geology in Tübingen. For 11 years, Oz has been programming with Python in various roles: as Data and System Engineer, Backend Developer, DevOps and finally as IT Architect at Noris Network AG. He loves Python as a language for data analysis and visualization, automation and web development.

```
>>> def odd_counter(max):
...     x = 0
...     while x <= max:
...         if not x%2 == 0:
...             yield x
...         x += 1
...
>>> odd_counter(10)
<generator object odd_counter at 0x7f771f800258>
```

Calling a generator creates a generator instance instead of running the generator. To run a generator, we call the next item in the generator:

```
>>> c = odd_counter(10)
>>> next(c)
1
>>> next(c)
3
...
>>> next(c)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

We can keep doing this until the code is exhausted. At this point the generator raises a *StopIteration* exception.

JaxMag: Can you give an example how asynchronous programming can be realized with the described language features?

Tiram: Because generators aren't immediately invoked, we can create many of them and use some method to schedule them to run when possible. If a coroutine can make use of the operating system's resources, it will block other coroutines from running. There are, however, existing mechanisms to switch between blocking coroutines when a blocking statement must be executed. This actually an old feature that has been around since Python 2.5, but it was not widely adopted until the introduction of the keyword *async* and *await* in later versions of Python. To read more about this I recommend looking at PEP 342 and especially the implementation of such a scheduler.

JaxMag: How do coroutines work in Python?

Tiram: Coroutines are a strange beast in Python. Like generators, they also have the keyword *yield* in their body. However, in a coroutine the *yield* appears right of the assignment operator. Coroutines are a follow-up step in the evolution of generators. Every time the execution of the generator code reaches a *yield* expression, the value of the *yield* is delivered to the caller, but it also suspends the execution of the generator until the *next* call. So the caller can now execute other tasks.

A coroutine can not only produce values, it can also receive values from the caller, which uses *.send(DATA)* to feed the coroutine with values. The caller can also stop the execution using *.close()*, or even to throw exceptions inside the coroutine with *.throw*. Here is a simple example of coroutine definition, priming, and usage:

```
>>> def hello_coro(subject):
...     print("Searching for: %s" % subject)
...     while True:
...         message = (yield)
...         if subject in message:
...             print("Found %s in %s" % (subject, message))
...

>>> # calling the coroutine does not run it
>>> coro = hello_coro("Python Summit")
>>> # advance the coroutine to the next yield statement (priming)
>>> next(coro)
Searching for: Python Summit
>>> coro.send("This will do nothing")
>>> coro.send("The Python Summit 2019 will be awesome")
Found Python Summit in The Python Summit 2019 will be awesome
```

JaxMag: Python is very popular in the machine learning environment. Why is that so? Is there a lot of semantic help in this language? Is it because of the available libraries? Or are there other reasons?

Tiram: For a start, Python is very popular in general. It's currently on the top in Stack Overflow, and whether it's the Red Monk popularity index or TIOBE's, it is always among the top 3 languages, surpassed only by Java and JavaScript. In general, Python isn't the fastest – its reference implementation cPython, which is the most widely used Python implementation, is the one I'm refering to – because it is an interpreted language and not a compiled one.

"Python is fast enough most of the time, with development speed more important than execution speed."

Python takes the approach that this is fast enough most of the time, and that development speed is more important than execution speed. When code needs to be executed fast, it can be optimized using the C language. cPython is written in C and extending the Python with C-based modules isn't hard.

In machine learning and in data science in learning, a lot of code is written for prototypes in exploration mode, so it makes no sense to optimize analysis code that is only executed once. As such, most of the time it does not matter that the code is slow. It does matter when you try to solve large partial differential equations or some other heavy CPU-based algebraic calculations. This is why most machine learning code isn't written in Python, but rather as extensions in C. Numpy, SciPy, Pandas, and TensorFlow are C-based libraries, which can be called directly from Python. So, one gets the comfort of Python with the speed of C.

The one thing that made Python popular among scientists, and therefore also in machine learning, is that the language is easy to read and write. In fact, for a couple of years now, it

has been the most widely used language when teaching computer programming in the USA. It's no surprise, then, that its popularity is booming. Python has no semantic help for machine learning, but its popularity and ease of use is what made it desirable in the field of machine learning.

One anecdote about data science with Python is that the popularity of Numpy, which is used for Matrix calculations and various algebraic operations, has pushed Python's developers to introduce a new operator into the language in version 3.5. This is the matrix multiplication operator marked as @. There are no built-ins in Python that use this operator, but Numpy uses it heavily.

JaxMag: Where does Python language development stand right now? What innovations are under discussion?
Tiram: Python is definitely developing, but the spirit of staying easy to learn is still there. Beginning with Python 3, we see a trend towards adding more and more typing support for this dynamic language. Python version 3 added optional function annotations:

```
>>> def foo(a: int, b: float=5.0) -> int:
...     pass
...
```

Which continued with type annotation for variables introduced in Python 3.6:

```
primes: List[int] = []

captain: str  # Note: no initial value!

class Starship:
    stats: Dict[str, int] = {}
```

Python 3.8 brings another form of typed dictionaries:

```
from typing import TypedDict

class Movie(TypedDict):
    name: str
    year: int
```

This all means that Python is starting to look a little like Java or other compiled languages. However, it's only a resemblance, because all the typing and hinting is optional. This means that for the one-time script you'll throw away, you can ignore these features, but for larger software projects where typing is desired, you can make use of them. My guess is that this will only increase Python's popularity and help reduce its image as "a scripting language".

One interesting project that builds on top of these developments is the static analysis tool MyPy. I'll simply quote the documentation on what it does:

"Mypy is a static type checker for Python 3 and Python 2.7. If you sprinkle your code with type annotations, mypy can type check your code and find common bugs."

I also believe that there will be a growing ecosystem of Python compilers to native code. Currently, the most known one is Cython, which allows one to write code in Python and then compile it to machine code via translation to C code. A similar project written from scratch is called nuitka, and I believe others will come.

Thank you for the interview!

Interview questions by Hartmut Schlosser

Imprint

Publisher
Software & Support Media GmbH

Editorial Office Address
Software & Support Media
Schwedlerstraße 8
60314 Frankfurt, Germany
www.jaxenter.com

Editor in Chief: Sebastian Meyen
Editors: Chris Stewart, Hartmut Schlosser, Sarah Schlothauer, Dominik Mohilo
Authors: Patrick Arnold, Thilo Frotscher, Dirk Lemmermann, Mike Milinkovich, Chi Nhan Nguyen, Falk Sippach, Daniel Stender, Jeff Sussna, Oz Tiram, Johan Vos, Kai Wähner, Anton Weiss, Tim Zöller
Copy Editors: Jonas Bergmeister, Dr. Anne Lorenz, Frauke Pesch
Creative Director: Jens Mainz
Layout: Dominique Kalbassi

Sales Clerk:
Anika Stock
+49 (0) 69 630089-22
anika.stock@sandsmedia.com

Entire contents copyright © 2019 Software & Support Media GmbH. All rights reserved. No part of this publication may be reproduced, redistributed, posted online, or reused by any means in any form, including print, electronic, photocopy, internal network, Web or any other method, without prior written permission of Software & Support Media GmbH.

The views expressed are solely those of the authors and do not reflect the views or position of their firm, any of their clients, or Publisher. Regarding the information, Publisher disclaims all warranties as to the accuracy, completeness, or adequacy of any information, and is not responsible for any errors, omissions, inadequacies, misuse, or the consequences of using any information provided by Publisher. Rights of disposal of rewarded articles belong to Publisher. All mentioned trademarks and service marks are copyrighted by their respective owners.

COMING SOON!

DevOpsCon

APRIL 21 – 24, 2020

Expo: APRIL 22–23, 2020

PARK PLAZA VICTORIA | LONDON

The Conference for Continuous Delivery, Microservices, Containers, Cloud & Lean Business

devopscon.io/london