Reactive Programming
with Scala, Lagom, Spark, Akka and Play

Interview with Scala creator Martin Odersky
The state of Scala

The Lagom Framework
Lagom gives the developer a clear path

DevOpsCon 2016: Our mission statement
This is how we interpret modern DevOps
“We believe that a coherent approach to systems architecture is needed, and we believe that all necessary aspects are already recognized individually: we want systems that are Responsive, Resilient, Elastic and Message Driven. We call these Reactive Systems.” – The Reactive Manifesto

Why should anyone adopt reactive programming? Because it allows you to make code more concise and focus on important aspects such as the interdependence of events which describe the business logic. Reactive programming means different things to different people and we are not trying to reinvent the wheel or define this concept. Instead we are allowing our authors to prove how Scala, Lagom, Spark, Akka and Play co-exist and work together to create a reactive universe.

If the definition “stream of events” does not satisfy your thirst for knowledge, get ready to find out what reactive programming means to our experts in Scala, Lagom, Spark, Akka and Play. Plus, we talked to Scala creator Martin Odersky about the impending Scala 2.12, the current state of this programming language and the technical innovations that await us.

Thirsty for more? Open the magazine and see what we have prepared for you.

Gabriela Motroc, Editor
Hot or Not

**Docker fork: Will it happen?**
 Discussions about a possible Docker fork first appeared in late August, when Google’s Craig McLuckie took to Twitter to reveal his thoughts about the need for “a container runtime and format standard to emerge beyond the (current) scope of OCI”, but he was not the only one to throw a rock at Docker. Google Kubernetes evangelist Kelsey Hightower took to Twitter to wonder whether “Docker deserves to be the stewards of an open container standard.” Meanwhile, Benjamin Wootton, co-founder of Contino, opined that “the Docker fork talk is a storm in a teacup” and Viktor Farcic, Senior Consultant at CloudBees, declared that “Docker 1.12 is probably the most important release since 1.0.”

**Java 9 leaves us hanging**
 Mark Reinhold, the Chief Architect of the Java Platform Group at Oracle, announced in a message on the OpenJDK mailing list in mid-September that the schedule for Java 9 may be delayed to July 2017. A few days later he told the JavaOne audience that Java 9 “will be coming soon – but not as soon as you’d probably hoped”. However, he also pointed out that Java 9 is not only about Jigsaw, although it is the headline of the release. Reinhold also mentioned in the message on the OpenJDK mailing list that even though the team has made significant progress on Project Jigsaw, it “needs more time”. If the JDK 9 schedule does not receive another extension, Java 9 should be launched in July 2017.

**Angular 2 is shy no more**
 It’s been two years since Google announced plans for Angular 2 and now the full-platform successor to Angular 1 is finally here. Angular 2.0.0 was released on September 14, 2016, one day after the team launched the last release candidate. According to the official statement, “Angular 1 first solved the problem of how to develop for an emerging web. Six years later, the challenges faced by today’s application developers, and the sophistication of the devices that applications must support, have both changed immensely”. Angular is now “ready for the world”, said Google’s Julie Kremer.

**BOSH 2.0 – Happy as a clam**
 Those who attended the Cloud Foundry Summit Europe 2016 in late September may have noticed that a certain project was mentioned over and over again: BOSH, an open source tool chain for release engineering, deployment and lifecycle management of large-scale distributed services. Version 2.0 (the logo of the project is a clam) is coming soon and will include the following: property validation, availability zones, global networking, cloud config, config server and links, according to Alexander Lomov, Cloud Foundry Engineer at Altoros.

**NetBeans is packing its bags and moving to Apache**
 NetBeans is moving to Apache in the hope that a change of scenery will help boost the number of contributions from various organizations. NetBeans Dream Team members reacted differently to the news, with one saying that Apache should not be the final destination of NetBeans. According to Bill Pataky, Oracle’s Vice President of Mobile Development Program and Developer Tools, Oracle developers will be involved in at least two Apache NetBeans releases.

www.JAXenter.com | October 2016
James Strachan, the creator of Groovy, once said that if somebody had shown him Martin Odersky’s “Programming in Scala” book back in 2003, he would have probably never created Groovy. Scala’s star is still shining bright – we invited Martin Odersky, the creator of Scala, to talk about the impending 2.12 version, the current state of this programming language and the technical innovations that await us.

Martin Odersky designed Scala in 2001. Now version 2.12 is in development and version 3.0 is just around the corner. Even James Gosling, the creator of Java once said that if he were to choose “a language to use today other than Java, it would be Scala.” We talked to the creator of Scala about the current state of this programming language, what to expect from version 2.12 and more.

JAX Magazine: It’s been 15 years since the work on Scala began at the École polytechnique fédérale de Lausanne (EPFL). If we look back at how Scala looked like more than a decade ago, one cannot help but wonder: What is the difference between the original concept and design of Scala and the current state of this programming language?

Martin Odersky: Scala was designed to show that a fusion of functional and object-oriented programming is possible and practical. That’s still its primary role. What has changed is that at the time it came out FP was regarded as an academic niche. So it was difficult to make the case that FP should be needed in a mainstream language. XML literals were added to the language as a specific use case, because we knew that traditional OO techniques had a hard time dealing with XML trees. Nowadays the tide has turned. Functional programming has become respectable, in some areas even mainstream, and it’s sometimes harder to make the case for good object-oriented design.

In terms of language features I believe the biggest development has been the refinements we did to implicit parameter search. Implicits evolved from a somewhat ad-hoc feature into the cornerstone of most Scala libraries. I believe they will become even more important in the future.

JAXmag: In your opinion, what are the most important technical milestones for this programming language?

Odersky: The most important step was no doubt Scala 2.8, which came out in 2010. With 2.8 we had for the first time a uniform collections library, refined rules for implicit resolution, and type inference. After that, Scala 2.10 was also a big step because it introduced meta-programming in Scala.

JAXmag: Were there also milestones in relation to the dissemination of the language? The community around Scala was formed rather fast and important projects and companies adopted it quickly. What contributed to the expansion of this language?

Odersky: Adoption has by and large steadily increased for the last eight years, but there were nevertheless a few drivers which led to rapid adoption in certain areas. The first leap was adoption of Scala by Twitter and other new web companies, starting in 2008. The second leap was widespread adoption of reactive programming, notably around Akka and Play, starting around 2011. The third leap was the success of Scala in big and fast data scenarios as well as data science, driven by Spark, Kafka, and many other frameworks.

Portrait

Martin Odersky created the Scala programming language and is a professor in the programming research group at EPFL, the leading technical university in Switzerland. He authored “Programming in Scala”, the best-selling book on Scala. Previously he has held positions at IBM Research, Yale University, University of Karlsruhe and University of South Australia, after having obtained his doctorate from ETH Zürich as a student of Niklaus Wirth, the creator of Pascal.
JAXmag: An entire stack was formed around Scala; it consists of Akka, Play, Lagom, Apache Spark and others. What role do this stack – and the so-called reactive programming paradigm – play in the Scala community and especially in Lightbend?

Odersky: Developing and supporting this technology stack is at the core of what Lightbend does. The stack covers everything from web frameworks to big data backends, with special emphasis on reactive programming and microservices in distributed scenarios. That’s where these technologies are very successful. A large part of the Scala community uses this stack, but there are of course also many other frameworks to choose from.

JAXmag: Right now Scala 2.12 is in development. What is this release all about?

Odersky: The main purpose of 2.12 is to optimize Scala for use on Java 8. Java 8 introduced lambdas and default methods for interfaces. Both are very useful to streamline Scala code generation and make the code size smaller.

JAXmag: Scala 3.0 is fast approaching. Can you offer us some insight into the technical innovations that await us?

Odersky: It’s too early to talk about that yet. The next major version will be Scala 2.13, which will concentrate on modernizing and modularizing some of the core libraries.

JAXmag: Will Scala 3.0 differ greatly from the 2.x line of development, as it is often the case with major versions or will it be a natural evolution of 2.x?

Odersky: Scala has been quite stable over the last five years. We hope that by the time Scala 3.0 comes out we will have the necessary technologies in place to make some bigger changes without too much disruption. For one, we are working on sophisticated rewrite tools that allow code to evolve to new standards. We are also planning to use TASTY, a platform independent interchange format, to avoid binary compatibility problems through automatic code adaption to specific platforms and versions.

JAXmag: Is there already a release roadmap for Scala 3.0?

Odersky: No, not yet. We want to keep some flexibility in deciding what to ship before we determine when to ship it.

JAXmag: Scala was explicitly designed as a language which should be adopted in research, but should also have an industrial use. Is the gap between industry and research really that big? There has been criticism with regard to the fact that a developer/architect’s job has little to do with what is taught in universities.

Odersky: I can’t speak for other universities. But I find that our students at EPFL have no problems finding interesting work in which they can apply what they have learned during their studies. Scala certainly helps since it has a healthy job market.

JAXmag: What area of research is still insufficiently taken into consideration in today’s mainstream languages?

Odersky: Functional programming is just catching up in industry. I think one of the next big developments will be the refinement of type systems to describe ever more precise properties of programs. This has been researched for a while, and more progress is needed to make this really practical. But I predict that this development end will pick up speed in the future.
Do you ever get the impression that our industry is one of extremes? Sometimes I get that impression. Many seem to be polarized in their opinions about various aspects of how software should be written or how it should work, whereas I think we could find a better balance based on the problem space that we face.

One example that I am thinking of is the common opinion that data must be persisted using ACID transactions. Don’t get me wrong. It’s not that I think that ACID transactions are not useful. They are useful, but they are often overused.

Should we find a better balance here? Consider the case when data that is spread across different subsystems must reflect some kind of harmony. Some hold to the strong opinion that all such dependent data must be transactionally consistent; that is, up-to-date across all subsystems all at once (Figure 1).

What about you? Do you agree with that? Do some of your software systems use global transactions in this way to bring data on multiple subsystems into harmony all at once? If so, why? Is it the business stakeholders who have insisted on this approach? Or was it a university professor who lectured you on why global transactional consistency is necessary? Or was it a database vendor who sold you on this service-level agreement? The most important opinion of these three is that of the business. It should be real business drivers that determine when data must be transactionally consistent. So, when was the last time you asked the business about its data’s consistency requirements, and what did the business say?

If we all took a look back to the time before most businesses were run by computers, we would find a very different picture of data consistency. Back then almost nothing was consistent. It could take hours or even days for some business transactions to be carried out to completion. Paper-based systems required physically carrying forms from one person’s desk or work area to another. In those days even a reasonable degree of data consistency was completely impossible. What is more, it didn’t result in problems. You know what? If today you got an authentic business answer to the question of necessary data consistency, you would very likely learn that a lot of data doesn’t require up-to-the-millisecond consistency. And yet, extreme viewpoints in our industry will still push many developers to strive for transactional consistency everywhere.
If you want to use microservices, you are going to need to embrace eventual consistency. Understand that the business will likely not be opposed, even if your university professor and your database sales representative will be. I explain how this works later in this article.

There’s another extreme that has cropped up recently. It’s service size, and specifically with regard to the use of “microservices”. What is a microservice anyway, and how does “micro” qualify the size of a service? Some guidance says: 100, 400, and 1,000. What? Yes, I have seen and heard that a microservice should be no more than 100 lines of code. Others say no more than 400 lines of code. And some are a bit more liberal at 1,000 lines of code. So, who is correct?

I am not sure that any of those extremes provide good guidance as to the size of a microservice. For example, if someone agrees with the rule that a microservice should be no more than 400 lines of code, and I create a microservice that is 537 lines of code, does that mean that my microservice is too big? If so, why? Is there something magical about 400 lines of code that makes a service just right?

On the other hand, some will take the opposite extreme view that software systems should be developed as monoliths. In such a case you could expect that nearly every subsystem that is required to support the core of the system will be housed in a single code base. When you deploy this system you must deploy all subsystems with it. If even one subsystem changes in a non-intrusive way to the other subsystems, the whole system must be deployed for the users to see the effects of one isolated change. Clearly there seem to be disadvantages with this approach, but is 400 lines of code the answer?

I think that the emphasis on the word “micro” should convey more the meaning that your team should not be creating a monolith, but instead a number of smaller independent services that work together to accomplish some significant business goal. But to say that a service that is of “micro” status must be no more than 400 lines of code also seems imbalanced. Take for example segmenting an existing monolith system into components of no more than 400-lines each and deploying all of them. At 400 lines of code, you are probably talking in terms of just one entity type per microservice. You could actually end up with hundreds or even thousands of these very small microservice components. What kind of problems could your teams face in trying to administer all those microservices? The hardware and network infrastructure alone would be very complex. It could be quite difficult to create and maintain such as system, say the least.

Still, if we strike a balance in terms of “micro” we might find a “measurement” of service size that is not only logical, but actually business driven. Since I am a big proponent of the Domain-Driven Design (DDD) approach to developing software, what I am suggesting here is to use two of the most important concepts of DDD to determine the size of a microservice: Bounded Context and the Ubiquitous Language.

If you have read the book “Building Microservices” by Sam Newman, you already know that Sam says that a microservice should be defined by a Bounded Context. Sam and I are in agreement. But actually saying that a microservice is a Bounded Context doesn’t exactly state what the size should be. That’s because the size of a Bounded Context is determined by its Ubiquitous Language. Thus, if you want to know the size of your Bounded Context (and microservice), you have to ask your Ubiquitous Language. It is the combination of the business experts and software developers working together that determines the Language of the Bounded Context. It’s important to understand that using business language drivers means that all linguistically cohesive components will live in the same Bounded Context. Those that are not linguistically cohesive with one Bounded Context thus belong in another.
What I find is a Bounded Context that is truly constrained by an actual business-driven Ubiquitous Language is quite small. It’s typically bigger than one entity (although it may be just one), but also much, much smaller than a monolith. It’s impossible to state an exact number, because the unique business domain drives the full scope of the Ubiquitous Language. But to throw out a number, many Bounded Contexts could be between 5 and 10 entity types total. Possibly 20 entity types is a large Bounded Context. So, a Bounded Context is small, yes “micro” in size, especially when compared with a monolith. Now, if you think about dividing a monolith system into a number of Bounded Contexts, the deployment topology isn’t so extreme.

At a minimum I think that this is a good place to start if your team is determined to break up a legacy monolith into a number of microservices; that is, use Bounded Context and Ubiquitous Language as your first step into microservices and you will go a long way away from the monolith.

Reactive software is ...

Reactive software is defined as having these four key characteristics:

- Responsive
- Resilient
- Elastic
- Message driven

A responsive system is one that responds to user requests and background integrations in an impressive way. Using the Lightbend platform, you and your users will be impressed by the responsiveness that can be achieved. Often a well-designed microservice can handle write-based single requests in 20 milliseconds or less, and even roughly half that time is not uncommon.

Systems based on the Actor model using Akka can be designed with incredible resilience. Using supervisor hierarchies means that the parental chain of components are responsible for detecting and correcting failures, leaving clients to be concerned only about what service they require. Unlike code that is written in Java that throws exceptions, clients of actor-based services never have to concern themselves with dealing with failures from the actor from which they are requesting service. Instead clients only have to understand the request-response contract that they have with a given service, and possibly retry requests if no response if given in some time frame.

An elastic microservices platform is one that can scale up and down and out and in as demands require. One example is an Akka cluster that scaled to 2,400 nodes without degradation. Yet, elastic also means that when you don’t need 2,400 nodes, only what you currently need is allocated. You will probably find that when running Akka and other components of the Lightbend reactive platform, you will become accustomed to using far fewer servers than you would with other platforms (e.g. JEE). This is because Akka’s concurrency capabilities enable your microservices to make non-blocking use of each server’s full computing resources at all times.
The Actor model with Akka is message driven to the core. To request a service of an actor, you send it a message that is delivered to it asynchronously. To respond to a request from a client, the service actor sends a message to the client, which again is delivered asynchronously. With the Lightbend platform, even the web components operate in an asynchronous way. When saving actor persistent state, the persistence mechanism is generally designed as an asynchronous component, meaning that even database interactions are either completely asynchronous or block only minimally. In fact, the point is that the actor that has requested persistence will not cause its thread to block while waiting for the database to do its thing. Asynchronous messaging makes all of this possible.

Reactive components
At each logical layer of the architecture, expect to find the following Lightbend platform components and microservice components that your team employs or develops (Figure 6).

For example, an Akka-based persistent actor looks like this (Listing 1).

This is a component based on event sourcing. The Product entity receives commands and emits events. The events are used to constitute the state of the entity, both as the commands are processed and when the entity is stopped and removed from memory and then recovered from its past events.

Now with those events we can derive other benefits. First we can project the events into the views that users need to see, by creating use-case-optimized queries. Secondly, the events can be published to the broader range of microservices that must react to and integrate with the event originating microservice.

What I am describing here is a event-driven architecture, one that is entirely reactive. The actors within each microservice make the one reactive, and the microservices that consume events from the others are also reactive. This is where eventual consistency in separate transactions come into play. When other microservices see the events they create and/or modify the state that they own in their context, making the whole system agree in time.

I have written three books on developing these kinds microservices based on Domain-Driven Design, and I also teach two workshops on these topics. I encourage you to read my books and contact me for more information on how you can put this practical microservices architecture to work in your enterprise.

Vaughn Vernon is a veteran software craftsman, with more than 30 years of experience in software design, development, and architecture. He is a thought leader in simplifying software design and implementation using innovative methods. Vaughn is the author of the books “Implementing Domain-Driven Design”, “Reactive Messaging Patterns with the Actor Model”, and “Domain-Driven Design Distilled”, all published by Addison-Wesley. Vaughn speaks and presents at conferences internationally, he consults, and has taught his “IDDD” Workshop and “Go Reactive with Akka” workshop around the globe to hundreds of developers. He is also the founder of “For Comprehension”, a training and consulting company, found at http://ForComprehension.com. You may follow him on Twitter: @VaughnVernon.
All you need to know about development and direction

What’s new in Akka?

Last year, Akka won the JAX Innovation Award for Most Innovative Open Source Tech – we’re still very proud of this recognition! – and one thing I can say is that winning the award did not make us pause or slow down. A lot is happening in and around Akka as we will see in this whirlwind tour through the ecosystem.

by Dr. Roland Kuhn

Before we begin, a personal note: On March 7 this year, Patrik Nordwall took over the Akka Tech Lead duties because I co-founded Actyx, a startup that helps small and mid-sized manufacturing companies to reap the benefits of digital technologies – sometimes classified as Industry 4.0. I remain very interested in Akka and I continue to develop some parts as we’ll see in this article, but I no longer speak for Lightbend.

The biggest development within Akka since 2.3.0 (March 2014) has for sure been the Reactive Streams [1] initiative and Akka Streams. The former is now slated for inclusion in Java 9 under the name Flow [2] and it forms the basis for the inner workings of the latter: Akka Streams is fully Reactive Streams-compliant. Streaming data pipelines are becoming ever more common and satisfy a rising demand of properly rate-controlled processing facilities – gone are the days of OutOfMemoryErrors resulting from one overwhelmed actor’s mailbox.

Streams were first developed on a very long-lived feature branch and versioned separately. Their experimental phase ended when they were released as part of Akka 2.4.2 in February 2016. In parallel, Spray – the lean and expressive HTTP library – has been rewritten on top of Akka Streams, a new foundation for reliable and efficient processing of large payloads. The resulting Akka HTTP modules present one of the most modern HTTP stacks available today, which is perfectly positioned for implementing the HTTP/2 standard that is swiftly gaining popularity: HTTP/2 connections are optimized for streaming several independent requests and responses at the same time. Konrad Malawski (maintainer of the Akka HTTP modules) and Johannes Rudolph (co-author of Spray and now member of the Akka core team at Lightbend) have started a proof-of-concept implementation for HTTP/2 support in September and I’m excited to see the results.

The 2.4.9 release has brought its performance up to the level of Spray or even beyond, and the Java and Scala APIs are merely being polished a bit more to straighten out a few remaining wrinkles. The surface area covered by an HTTP implementation as complete and expressive as this one is enormous: the user-facing API has more than 26,000 lines of Scala code, defining all those media types, encodings and other headers as well as the directives for routing requests and formulating responses. In order to allow more people to contribute efficiently – and in order to allow the HTTP modules to evolve at a faster pace than the very stable Akka core – the high-level HTTP modules are moving out of the core Akka repository into a separate new home under the Akka organization on github. For more details see the 2.4.9 release notes and the discussion on the akka-meta repository [3].

Apropos contributions: since the addition of Streams & HTTP, we have seen a marked increase in the breadth of our contributor base. Every minor release contains the work of circa 40 community members. Contributing has never been easier, not only because the high-level features of Streams & HTTP are more accessible, I think it is also due to the use of Gitter for nearly all team communication.

There is also vivid exchange with authors of integration libraries that connect Akka Streams to data sources and sinks like message brokers, files, databases, etc. In order to foster this growing ecosystem, an umbrella project named Alpakka [4] has been created, similar in spirit but of course currently a lot less powerful than Apache Camel. Everyone is invited to think up and implement new integrations – this is your chance to give back to the community. These are currently best targeted at the Akka Stream Contrib repository, but the team is thinking about opening a special Alpakka dedicated repository as well.

Another project that will have deep impact on how you use Akka is going on under the hood and driven by the core team – its codename is Artery. This new remote messaging stack is based on the Aeron [5] library and on Akka Streams and delivers vastly improved throughput and latency for sending messages across the network – just be sure to not use Java Serialization, which is just too slow to keep up. The goal of this rewrite of the basis for clustering is not only improved performance, the main impetus behind it is to use the expressive power and safety of Streams to simplify the implementation and eventually provide verified cross-version stability and compatibility. The current status (as of Sep 19, 2016) is that milestone 4 is available for you to try out and give feedback on [6]. Within the next few months Artery will replace the old remoting implementation and become the default.

The last project to talk about is the one that I am deeply involved in. Akka Typed has been included since version 2.4.0 as a sketch of how to achieve type-safety in actor interactions.
in a fashion that is simple to understand, without compiler magic, and inherently built for great and easy test coverage. In a nutshell, it consists of a generic ActorRef that only accepts a certain well-specified type of messages, plus a generic Behavior (i.e. actor implementation) that only expects inputs of a given type. It is the third attempt in this direction within Akka and I firmly believe that this is the one that works. Recently, also due to the intention to use this at Actyx, I have created an entirely new implementation of the actor mechanics for the Akka Typed package, making it available alongside the adapter that offers the new features as an emulation layer on top of the untyped akka-actor package.

Based on the experience collected over the past six years from developing Akka, we decided to remove features that greatly complicate the internal code while not delivering sufficient benefit to all users. The full list has been discussed on akka-meta [7]; highlights are the configurability of mailboxes and remote deployment. Together with the relocation of the execution of supervisor strategies into the child actors themselves this has led to a huge reduction in code size, while the user-facing feature set has only been trimmed slightly.

Special focus has been devoted to pervasive testability, both of the implementation and of user-level code. The main difference to untyped actors in this regard is that behavior and execution are fully decoupled now, you can create your actor behaviors directly within tests and exercise them synchronously without using a special TestActorRef or CallingThreadDispatcher. This allows convenient validation of internal logic as well as deterministic stimulation with external inputs. A mock ActorContext can be used to check that child actors are created correctly, all other effects like watching/unwatching are accessible as well.

The new implementation is quite a bit faster than the old one, reducing actor messaging overhead by 20–30 percent even though the mailbox implementation is not yet optimized. Once the full potential has been realized, it will be possible to achieve allocation-free actor messaging. The current breadth of the implementation covers only local actors without persistence, the plan is to add a custom Materializer for Akka Streams and reuse Artery in order to have a new basis for the existing clustering code.

One aspect that I find extremely exciting about Akka Typed is that it opens up actor interactions to even more static verification than is afforded by a type-selective ActorRef. Current research within the ABCD group [8] has the potential to enable an actor-based implementation of a verified communication protocol to be checked for conformance to the specification by the Scala compiler (here it might be that Java’s type system is not expressive enough to achieve the full feature set). This would mean that many bugs would be caught without even running a single test, the compiler will tell you that you forgot to send a certain message or that you expected to receive a message that will never arrive.

Enabling the compiler to see these actions means representing them in the type-system and lifting the description of what an actor does into a sequence of actions that can be inspected. This is a powerful tool even without the protocol verification, it allows actor behaviors to be composed in ways that were very cumbersome or impossible before. Think about an actor that speaks some protocol with its clients, in the simplest case request–response. In order to compute the response the actor might have to converse with a handful of other actors in a multi-message exchange. Previously you would implement that by manually stashing all unexpected messages and switching between different behaviors using context.behave(). With the upcoming Akka Typed DSL [9] it becomes possible to write down a sequence of send and receive actions using a Process abstraction, where processes can be composed to run in sequence or in parallel within the same actor. This greatly simplifies the code structure because it disentangles the different aspects of an actor’s behavior and sorts them into cohesive strands that each fulfill their separate purposes.

All this is currently still in the research phase, so feedback is very much welcome and needed to get it ready soon.

Oh, and one last thing: at Actyx we are currently working on an improved version of distributed data structures (for the curious: an implementation of 6CRDTs) that can be used independently of Akka Cluster. If that turns out to be successful, we will open-source this as a cousin to Akka Distributed Data.

Dr. Roland Kuhn is CTO and co-founder of Actyx, author of Reactive Design Patterns, a co-author of the Reactive Manifesto, co-teacher of the Coursera course “Principles of Reactive Programming”, and a passionate open-source haker. Previously he led the Akka project at Lightbend.

References

[5] https://github.com/real-logic/Aeron
[6] https://groups.google.com/forum/#!topic/akka-user/V0DH_e8w7M0
[9] See my presentation at scala.world, slides at http://www.slideshare.net/rolandkuhn/distributed-systems-vs-compositionality
Akka: The Swiss army knife of reactive systems on the JVM

Dedicated to doing the right thing

by Manuel Bernhardt

Akka is in many ways a very innovative and yet robust technology. Now that it has become generally accepted that microservice-oriented architectures are a good idea, Akka will play an even more important role in the reactive system at large and already serves as a foundation for other reactive technologies such as the Play Framework and Lagom.

Akka is one of the most mature technologies available on the JVM for building reactive applications. It builds on top of the actor-based concurrency model and is inspired by Erlang (in fact, the initial name of the project was Scala OTP). The influence of the Akka project on the reactive ecosystem is rather significant: Jonas Bonér (Akka’s creator) and Roland Kuhn (Akka’s former project lead at Lightbend) are the co-authors of the Reactive Manifesto. The Reactive Streams initiative has been significantly supported by the Akka team.

Actors as the unit of computation

Akka applications are comprised of actors that are arranged in a hierarchy and send messages to each other. An actor’s behavior describes what happens when an actor receives a message: it can alter its state, create a new child actor, forward the message to another actor, reply to the sender or ignore it altogether. It is through this set of interactions between actors that advanced concurrent applications can be written without the headaches brought in by thread-based concurrency such as unwanted race conditions, data races, deadlocks and livelocks which are very hard to reason about even for seasoned developers. Akka explicitly takes these pains away by providing an abstraction at a higher level – but not high enough to hide the fact that the application is concurrent, a trend seen in many application servers and frameworks and which has turned out to be hurtful rather than helpful.

The actor hierarchy is the secret ingredient that makes Akka applications resilient to failure: each parent actor supervises their children, being responsible for what happens in case of crash. The supervisor decides what happens to the failing child actor by specifying a supervisor strategy. Depending on the type of exception, the failing actor (and if necessary, all of its siblings) is then either resumed, restarted or stopped – and in some cases, the failure is escalated higher in the hierarchy until someone knows what to do. This separation of failure handling logic and business logic is a key concept in Akka’s design: failure is embraced and treated as a first-class citizen, rather than an afterthought.

Designed from the ground up for distributed systems

Akka actors are designed in such a way that the physical location of an actor should not influence how it can be interacted with. This concept, known as location transparency, is of paramount importance when it comes to building reactive applications which need to be distributed in order to be resilient through redundancy and capable to scale out when the load demands it. Paradoxically then, distribution itself is a necessary evil which lies at the root of many failures since networks are shaky constructs that fail all the time.

Akka promotes a few simple principles to embrace network-induced failure: distributed systems are prone to failure due to their very nature and Akka does not try to hide this behavior. In fact, the Akka team’s mantra is “No Magic”, always making it explicit to its users which guarantees Akka can give and what they need to be aware of. First of all, messages should be sent in the asynchronous fire-and-forget mode in which the sender does not explicitly wait for an answer but later reacts to an answer, using correlation identifiers when necessary. Second, first-class failure handling is provided through the previously described actor supervision. Third, Akka is explicit as to what guarantees it can give in terms of message delivery. It provides best mechanism to col-
lect locally undelivered messages, the so-called “dead letters”, providing a means to inspect why certain messages do not make it to their sender, at least locally. However, this mechanism does not work across network boundaries where the use of acknowledgements is required to guarantee at-least-once delivery semantics. In order to build distributed applications, Akka offers some very useful extensions.

**Akka persistence, Akka cluster and Akka HTTP**

Akka Persistence allows actors to recover their state after a crash. Persistent actors have a journal that allow them to replay events after a crash; they can also make use of snapshots to speed up the recovery. Journal entries and snapshots can be stored in a variety of backends such as Cassandra, Kafka, Redis, Couchbase and many more.

Akka Cluster lets an actor system run on several nodes and handles basic concerns such as node lifecycle and location-independent message routing. In combination with Akka Persistence, it provides at-least-once delivery semantics for messages sent across the wire. It uses a lightweight gossip protocol for detecting when nodes are failing. Lightbend’s commercial offering also adds the Split Brain Resolver (SBR) extension that allows to handle correctly in the face of network partitions, where it may not be trivial to decide which nodes should be removed and which ones should survive.

Akka HTTP offers client and server capabilities for HTTP and WebSockets including DSLs for describing URI-based routing.

**One level higher and an interface to the world: Akka Streams**

After several years of building and maintaining actor-based systems with Akka, it became clear that actors may sometimes still be too low-level of a concept, especially when it comes to describing advanced scenarios including control flow and failure handling. Akka Streams represent an answer to this insofar as they allow to describe sophisticated flow manipulation “machines” through a rich set of combinators.

Rather than limiting itself to the Akka universe, Akka streams implement the Reactive Streams API for non-blocking asynchronous stream manipulation on the JVM. Thanks to this, applications built with Akka Streams can seamlessly interoperate with other technologies implementing the Reactive Streams API.

**A drive for quality and innovation**

Akka is in many ways a very innovative and yet robust technology. Now that it has become generally accepted that microservice-oriented architectures are a good idea – better, at least, than the vast, monolithic and unmaintainable enterprise systems that so many companies are stuck with (and because of), Akka will play an even more important role in the reactive system at large and already serves as a foundation for other reactive technologies such as the Play Framework and Lagom.

In my opinion, it is this dedication to “doing the right thing” that makes Akka such an exciting project and great technology to work with. The Akka team does not fear to experiment first with APIs and implementations (Akka Streams, for example, has seen as many as six complete rewrites over the source of three years before it was deemed good enough). This is also why, when working with Akka, you should always be mindful of extensions tagged as experimental in the documentation: there is a real chance that the APIs will change significantly over time, which is not necessarily a bad thing in itself but something to be aware of nonetheless.

Last but not least, Akka has a very active community and an excellent documentation – so good, in fact, that it is rather difficult to do better when writing a book about it. I can only recommend downloading the PDF and reading the documentation as a whole when getting started with the project to get a sense of what pieces are already provided by the toolkit and which concepts to be aware of. Happy hAkking!

---

**Manuel Bernhardt** is a passionate engineer, author, speaker and consultant who has a keen interest in the science of building and operating networked applications that run smoothly despite their distributed nature. Since 2008, he has guided and trained enterprise teams on the transformation to distributed computing. In recent years he is focusing primarily on production systems that embrace the reactive application architecture, using Scala, Play Framework and Akka to this end. Manuel likes to travel and is a frequent speaker at international conferences. He lives in Vienna where he is a co-organizer of the Scala Vienna User Group. Next to thinking, talking about and fiddling with computers he likes to spend time with his family, run, scuba-dive and read. You can find out more about Manuel’s recent work at [http://manuel.bernhardt.io](http://manuel.bernhardt.io).
by Marius Soutier

The Play framework used to have a lot of global state, most and for all the current running application that contains things such as the configuration, access to plugins etc. Another strange thing was the Global object, a singleton used not only to register startup and shutdown hooks but also to override error handlers. All of this leaked into the application lifecycle and the plugin system. For example, you had to define a plugin priority and make sure external components are instantiated lazily and closed after an app restart in development mode. Lastly, global state makes it harder to test your application in a clean way.

Play introduced Dependency Injection (DI) in version 2.4 (Note: this article is up-to-date with Play 2.5) to reduce (Note: global state is not completely gone yet, but will be gradually removed in the next versions) global state, and make it easier to write isolated, reusable, and testable code by instantiating components on each reload and by providing stop hooks.

Runtime DI
The runtime DI mechanism defined by Java Specification Request 330, with its reference implementation Guice, has been selected as the default DI framework in Play. This makes it rather straightforward to start writing new Play applications without worrying too much about DI, as components are injected automatically by using the @Inject annotation. The official documentation on runtime DI is pretty complete so I will not go into detail about it.

Compile-time DI
Guice as a standard mechanism makes sense for a framework that offers a first-class Java API.

As Scala developers, however, we don’t like to rely on reflection, runtime instantiation or to worry about eager bindings. Luckily, Play also provides a compile-time DI mechanism.

In its simplest form, compile-time DI, or any DI for that matter, means passing dependencies to a constructor or a function. All of your dependencies are declared as constructor parameters in your classes, be it controllers, actors, or any other class. When your Play app launches the Application Loader, which replaces the Global object, it prepares all dependencies and instantiates your classes by passing their dependencies into their constructor. The link from controllers to route handling is made by creating a generated router that maps routes to instantiated controllers – again, via their constructors. From that, the Application Loader assembles and provides the current Application.

All parts of Play, now called components, support compile-time DI by providing a trait which ends with Components (Note: a lightweight version of the cake pattern) and builds on three basic dependencies: ApplicationLifecycle, Configuration, Environment.

ApplicationLifecycle allows your component to register a stop hook. Configuration is the TypesafeConfig that you would previously get from a running application. Environment is a collection of methods that revolve around your running Play app, such as the classloader, the current mode (development, test, or production), and obtaining classpath resources. By extending BuiltInComponentsFromContext, you get these three provided by Play.

Let’s say we have a small app with one Application controller that needs to make a web service call. Now instead of importing Play’s WS which required the current application, we mix in a trait at the application loader level and pass a concrete WS instance to the controller.
The controller in turn depends on WSClient – which is just a trait:

class Application(ws: WSClient) extends Controller {...}

Finally, we need to tell Play to use our Application Loader in application.conf:

play.application.loader="com.mariussoutier.example.AppLoader"

While we now have more boilerplate to write, this way of building our application has several advantages:

- Dependencies on components are more explicit
- We avoid using the current Application
- We can easily switch to a mocked WS implementation when writing tests
- When we refer to a new controller in the routes file, the compiler will tell us that Routes is missing a dependency

Testing

Since we control how our application components are assembled, it’s easy to create an ApplicationLoader for our tests.

ScalaTest supports compile-time DI with its OneAppPerSuite trait.

Plugins

Play plugins are now deprecated and replaced by DI component, which means the entire plugin system is gone, along with its configuration file and priorities.

You just provide a component which ideally should be compatible with both runtime and compile-time DI. For this, you typically write a generic API trait and implement it using a Module or a JSR-330 Provider class, and a components trait for compile-time DI.

A basic example to get started is the ActorSystemProvider in Play itself which is also used in compile-time DI via Akka-Components.

Conclusion

By using compile-time dependency injection we gain more control over how our application is assembled, making it more testable. Writing isolated and testable components is now straightforward and no longer requires an elaborate plugin system. Plus, we don’t have to worry about referring to an application too early.

You can find a full example in my PlayBasics repository under https://github.com/mariussoutier/PlayBasics/tree/master/DependencyInjection.

The controller in turn depends on WSClient – which is just a trait:

class Application(ws: WSClient) extends Controller {...}

Finally, we need to tell Play to use our Application Loader in application.conf:

play.application.loader="com.mariussoutier.example.AppLoader"

While we now have more boilerplate to write, this way of building our application has several advantages:

- Dependencies on components are more explicit
- We avoid using the current Application
- We can easily switch to a mocked WS implementation when writing tests
- When we refer to a new controller in the routes file, the compiler will tell us that Routes is missing a dependency

Testing

Since we control how our application components are assembled, it’s easy to create an ApplicationLoader for our tests.

ScalaTest supports compile-time DI with its OneAppPerSuite trait.

Plugins

Play plugins are now deprecated and replaced by DI component, which means the entire plugin system is gone, along with its configuration file and priorities.

You just provide a component which ideally should be compatible with both runtime and compile-time DI. For this, you typically write a generic API trait and implement it using a Module or a JSR-330 Provider class, and a components trait for compile-time DI.

A basic example to get started is the ActorSystemProvider in Play itself which is also used in compile-time DI via Akka-Components.

Conclusion

By using compile-time dependency injection we gain more control over how our application is assembled, making it more testable. Writing isolated and testable components is now straightforward and no longer requires an elaborate plugin system. Plus, we don’t have to worry about referring to an application too early.

You can find a full example in my PlayBasics repository under https://github.com/mariussoutier/PlayBasics/tree/master/DependencyInjection.
We speak your language.

Do you speak ours?

We are the leading planning and performance management platform.

Whatever language you speak—in person and in code—we want to talk.

Check out our company page on Stack Overflow or visit anaplan.com
Lagom gives the developer a clear path

The Lagom Framework

Radically different, but nonetheless easy – that is the dichotomy the new open source microservice framework Lagom is trying to create. What are the features that differentiate it from other frameworks? How easy is it to handle? What does the name actually mean?

by Lutz Hühnken

The question regarding the meaning of the name is not easy to answer since one cannot literally translate the Swedish idiom Lagom. According to Wikipedia, the meaning is: “Enough, sufficient, adequate, just right.” In our case, this is not supposed to be a self-praise but a critical statement to the concept of microservices. Instead of focusing on “micro” and stubbonly following a “the less code, the better” concept, Lagom suggests that we think of a concept of “Bounded Context” from the Domain-Driven Design to find the boundaries for a service. The conceptual proximity of domain driven design and microservices can be found in different locations in the Lagom framework.

Getting started with Lagom

The easiest way to develop an application with Lagom is with the help of a Maven project template:

```bash
$ mvn archetype:generate -DarchetypeGroupId=com.lightbend.lagom \
-DarchetypeArtifactId=maven-archetype-lagom-java \
-DarchetypeVersion=1.1.0
```

After the questions regarding names have been answered and you switch into the newly-created directory, you will find the directory structure as displayed in the Listing 1.

As it should be for microservices, not one, but already two services were generated. After all, the interaction and communication between services are at least as important as the implementation of a single one (and frequently the bigger challenge). Here are the services “hello” and “stream”; each implementation is divided into two subprojects (“api” and “impl”). To launch the application, a simple `mvn lagom:runAll` is enough. After a few downloads, should be running at Port 9000. This can be easily checked with a command line tool like HTTPIE (Listing 2).

One particularity that all components needed in the development have is that they – the project’s services, a service registry, an API gateway, and even the database Cassandra (in the embedded version) – are launched through the Maven plug-in. It is not necessary to set up services or a database outside of the project. Lagom stresses the importance to offer the developer an environment which feels interactive – check out the project and get going. This includes the fact that code changes will come into effect right after a reload, without the need for a `build/deploy/restart cycle`.

The services API — typesafe and asynchronously

As it can be seen from the folder structure, every service is divided into an implementation (“-”) and an API definition (“-”). The latter defines the HTTP interface of the service programmatically, as shown in Listing 3. With the help of a builder, the service description will be created, in which the requested path will be mapped on a method call.

This interface is not only the template for implementation; Lagom also generates an appropriate client library. In other Lagom services, this can be injected via dependency injection with Google’s Guice. This way, a type-safe interface is provided when the respective service is selected. The manual construction of an HTML request and the direct use of a generic http client can be omitted.

Still, it is not mandatory to use the client library because the framework maps

```
Listing 1
cassandra-config
hello-api
hello-impl
integration-tests
pom.xml
stream-api
stream-impl
```

```
Listing 2
$ http localhost:9000/api/hello/
Lagom
HTTP/1.1 200 OK
Content-Type: text/plain
Hello, Lagom!
```

```
Listing 3
public interface HelloService extends Service {
    ServiceCall&ltNotUsed, String&gt hello(String id);
    default Descriptor descriptor() {
        return named("hello").withCalls(
            pathCall("/api/hello/:id", this::hello),
            );
    }
}
```
the method calls on HTTP calls, which may also be called directly, especially by non-Lagom-services.

By the way, our little “hello” method doesn’t deliver the response directly, but a ServiceCall. This is a functional interface. That is to say we do not create a simple object but a function – the function which shall be executed by the corresponding request. We deliver the types as type parameters for the request (since user GET call doesn’t submit any data, in this case “NotUsed”) and the response (in our case a simple String). The processing of the request is always asynchronous – the outcome of our function must be a CompletionStage. Lagom extensively uses Java 8 features. A simple implementation would look like this (Listing 4).

For a simple GET request, the gain of the service descriptors is limited. It gets more interesting when we want to send events between services asynchronously. We can achieve this in Lagom by choosing different type parameters for the ServiceCall. If our request and response types are defined as source (a type from the Akka streams library), as shown in Listing 5, the framework will initialize a WebSocket link. Here the service abstraction can score since it simplifies working with the WebSockets. As far as future versions are concerned, there are plans to support the additional “publish/subscribe” pattern so that messages can be placed on a bus and other services can subscribe to it.

Circuit breaker built-in

Let us assume that our service requests information per HTTP request at another service. This doesn’t respond within the expected timeframe, which means there will be a timeout. Requests to this server shouldn’t be repeated constantly because we ask an unnecessary amount of idle time from our application: If it’s likely that we won’t be getting a response, why should we wait for a timeout? Furthermore, there would be requests accumulating to the service. As soon as it becomes available again, it will be bombarded with pending requests to such an extent that it will be brought to its knees immediately. A reliable solution for this problem is the circuit breaker pattern. A circuit breaker knows three states:

- As long as everything is running without errors, it is closed.
- If a defined limit of errors (timeouts, exceptions) is reached, it will be open for a defined period of time. Additional requests will fail with a “CircuitBreakerException”. For the client there won’t be additional waiting time and the external service won’t even notice the request.
- As soon as the set time period runs out, the circuit breaker will switch into the state “half open”. Now there will be one request passed through. If it is successful, the circuit breaker will be closed- the external system seems to be available again. If it fails, the next round with the state “open” begins.

Such circuit breakers are already integrated into the Lagom service client. The parameters are adjustable with the configuration file.

Lagom persistence

One aspect which proves that Lagom is very different from other micro frameworks is the integration of a framework for Event Sourcing and CQRS. For many developers, working with a relational database is still the “default case”, possibly in connection with an ORM tool. Even this can be implemented in Lagom, but the user is steered into another direction. The standard in Lagom is the use of “Persistent Entities” (corresponding to “Aggregate Roots” in Domain-Driven design). These Persistent Entities receive messages (commands).

Listing 6 shows exactly how this is presented in the code. Our quite simple entity allows us to change the welcome text. For type-saving purposes, one can go back to commands, events and states from the library Immutables. To save yourself some keystrokes, you can leverage a library such as Immutables for your commands, events and states.

The way our entity responds to commands is defined by a behavior. This can change at runtime. This way the entities can implement finite-state machines – the replacement of one behavior with another at the runtime correlates with the transition of the machines into another state. The framework obtains the initial behavior via initialBehavior. To construct this, we will make use of the builder pattern.

First, we define a CommandHandler as our command. If a command is valid and demands the entity to be changed, for example, in case it sets an attribute to a new value, the change
won’t occur immediately. Instead, an event will be created, saved and emitted. The EventHandler of the persistent entity which we also added with the builder to the behavior, reacts to the event and executes the actual change.

A significant difference to an “Update” in a relational database is that the current state of the persistent entity does not necessarily have to be saved. This will be merely held in memory (Memory Image). In case it becomes necessary to restore the state, e.g. after a restart of the application, this will be reconstructed through a playback of the events. The optional saving of the current state in called “Snapshot” in the model and does not replace the Event history, but solely represents a “pre-processing”. If an entity experienced thousands of changes of state during its lifetime, there is no need to play back all the events from the very beginning. It is possible to shortcut by starting with the latest snapshot and repeating only the following events.

The strict specifications that Lagom gives for the types and the structure of the behavior are meant to ease the conversion to this principle, called Event Sourcing, for developers. The idea is that I am forced to specify a clear protocol for each entity: Which commands can be processed, which events can be triggered and which values define the state of my class?

**Clustering included**

The number of Persistent Entities that I can use is not limited by the main memory of a single server. Rather, every Lagom application can be used as a distributed application. During the start of an additional instance I only have to add the address of an already running instance, after that it will register there and form a cluster with the present instances. The Persistent Entities are administered by the framework and will be distributed automatically within the cluster (Cluster Sharding). If nodes are added to or removed from the cluster, the framework will redistribute the instances. Likewise, it can restore instances which were removed from the memory (Passivation).

By the way, the built-in feature to keep the application state in the memory this way and also to scale this hasn’t been developed for Lagom originally. For this, Lagom relies on Akka. This has definitely been used in mission-critical applications, therefore any concerns regarding the reliability of the young framework are not well-founded.

**Separate writing and reading**

While it is easy in SQL databases to request any information from the data model, it is impossible in the case of Event Sourcing. We can only access our entity and request the state with the primary key. Since we only have an Event Log and not a relational data model, queries through secondary indices are impossible to make.

To enable this, the CQRS architecture (Command Query Responsibility Segregation, for further reading: A CQRS Journey, https://msdn.microsoft.com/en-us/library/jj554200.aspx) is applied. The basic principle here is that different data models are used for reading and writing. In our case this means that our Event Log is the write side... It can be used to reconstruct our entities, but we won’t perform any queries on this. Instead, we also generate a read side from the events. Lagom is already offering an ReadSideProcessor. Every event which occurs in combination with a class of PersistentEntities will also be processed and used to create the read side. This is optimized for reading and doesn’t allow for direct writing.

This architectural approach does not only offer technical advantages, since in many application cases the read and writing frequency are very different and they are scaled independently with this method. It also enables some new possibilities. As a consequence of never deleting the saved events, it is possible to add new structures on the read side, the so-called projections. These can be filled with the historical events and thus can give information not only in the future but also from the past.

CQRS allows the use of different technologies on the read side, adjusted to the Use Case. It is conceivable while not supported by Lagom yet, that one can build an SQL read side and continue the use of available tooling, but simultaneously feeding an ElasticSearch database for the quick search and to send the events for analysis to Spark Streaming. It is important to keep in mind that the read side will be refreshed asynchronously, with latency (“Eventual Consistency” between the write and the read side). Strong consistency is only available in this model on the level of the PersistentEntity.

Finally, it is also possible to code Lagom without Lagom Persistence. It is not mandatory to use Event Sourcing; the development of “stateless” – Services, or “CRUD” applications (Create, Read, Update, Delete) with a SQL database in the

```java
Listing 6
public class HelloEntity extends PersistentEntity&amp;
ltHelloCommand, HelloEvent, HelloState&amp gt {

@Override
public Behavior initialBehavior(Optional&amp;
ltHelloState&amp snapshotState) {

/*
 * The behavior defines how the entity reacts on
 * commands.
 */
BehaviorBuilder b = newBehaviorBuilder(

snapshotState.orElse(new HelloState("Hello",
LocalDateTime.now().toString()));

/*
 * Event handler for GreetingMessageChanged.
 */
b.setEventHandler(GreetingMessageChanged.class,
vit -&gt new HelloState(vit.message,
LocalDateTime.now().toString()));

/*
 * Command handler for UseGreetingMessage.
 */
b.setCommandHandler(UseGreetingMessage.class,
(cmd, ctx) -&gt
tctx.thenPersist(new
GreetingMessageChanged(cmd.message),
evit -&gt vit.ctx.reply(Done.getInstance())));

return b.build();
}
}
```
backend is also possible. But if someone is interested in Event Sourcing and CQRS, in scalable, distributed systems, Lagom can help them gain access into the topic.

**Immutable values — Immutables**

As mentioned earlier, the single commands, events and instances of the state must be immutable. Immutable data structures are an important concept from the functional coding, especially in the area of concurrency. Let us assume a method gets passed a list of numbers. The result is a value that is calculated from the list (maybe a meridian of the numbers of the list). By reasoning about this or maybe in some cases even through mathematical proof, you may state that a function is correct and will always deliver the same output for the same input.

But what if the delivered list is e.g. an ArrayList — how can we be sure? Fix is only the reference that is delivered. But what if another part of the program that is executed in parallel has the same reference? And adds some values to the list? In asynchronous systems that are based on sending the commands, it is essential that a command must not be changed after it has been sent. To rely on the fact that the developer will be careful would be negligent.

Lagom uses third party libraries for this. For the commands it binds Immutables, for collections pCollections. If I add a value to a collection from this library, the original collection will remain unchanged and I will receive a new instance with an additional value.

**Deployment**

Microservices provide a challenge not just for the developer but also for the ongoing operation. In many companies the deployment processes are still set up for the installation of .war or .ear files for application servers. But microservices are running standalone and are often packed into (Docker) containers and administered by the so-called service orchestration tools like Kubernetes or Docker Swarm.

Lagom requires such an environment, too. But it does not depend on a certain container standard (like Docker). It requires the runtime environment to have a registry which is searchable through other services. To be accessible, it must make an implementation of the Lagom ServiceLocator API available.

Unfortunately, at the moment it is only available for the commercial closed-source product ConductR. The open source community is working on the implementation for Kubernetes and Consul. Alternatively, a ServiceLocator based on static configuration can be used, but this is not recommended for production use.

**Conclusion**

Lagom follows an interesting path and is a remarkable framework. It’s fundamentally different in its technical base: Everything is asynchronous, it is based on sending commands and persisting is done per Event Sourcing. This brings tremendous advantages for the scalability of services — but for most developers (including everybody from the Java EE area), this means rethinking. With the change of a programming language always comes the fear of a temporary decrease in productivity because developers cannot revert to familiar practices and resources. It is the same in our case.

Lagom is trying to prevent this by giving the developer a clear path. If I follow the documentation of the textbook approach for service implementation and persistence in Lagom, I will be able to build a reactive system — completely based on messaging and being able to cluster, maybe even without realizing it.

In the relatively new area of microservices, standards are yet to be established. We will have to see which frameworks can stand the test of time. In contrast with old acquaintances from Java EE and Spring, Lagom instills new life into this and is putting a whole different architecture in the balance. Those who wish to try something new and are interested in scalable distributed systems will find Lagom helpful.

Lutz Hühnken is Solutions Architect at Lightbend. He is an experienced software architect, project manager and team leader, with a history of successful projects in systems integration, internet/e-commerce and server-side application development in general.
SMACK – Next generation Big Data

Big Data becomes Fast Data

Big Data is changing. Buzzwords such as Hadoop, Storm, Pig and Hive are not the darlings of the industry anymore – they are being replaced by a powerful duo: Fast Data and SMACK. Such a fast change in such a (relatively) young ecosystem begs the following question: What is wrong with the current approach? What is the difference between Fast and Big Data? And what is SMACK?

by Jochen Mader

Google retired MapReduce at I/O 2014: By then, one had already switched to the new Dataflow framework and removed the existing MapReduce jobs. This announcement caused a stir since Hadoop and its ecosystem were still seen as ‘innovative’. After a few apocalyptic blog posts and some vigorous debate, calm was restored. Many companies dipped their toes into the Big Data universe but learned a valuable lesson, namely that the limits of many technologies are too restricted for the desired periods of analysis. A new concept was needed. The following article will show you how Big Data (on Hadoop) became Fast Data (with SMACK).

At the beginning there was Lambda

Over the years, the big data world evolved into a confusing zoo of interwoven frameworks and infrastructure components. HDFS, Ceph, ZooKeeper, HBase, Storm, Kafka, Big, Hive etc. Many of these components are highly specialized and depict only a subset of the intended functionality. Only their – not very unproblematic – combination allows for the execution of more complicated use cases. Over time, it has been shown that many frameworks can be divided into two different groups: On the one side we have the frameworks that respond immediately (passage: “Real Time”). This category contains Storm, Samza, different CEP engines, but also reactive frameworks like Akka, Vert.x or Quasar. The other group consists of frameworks that require some time to respond. Here everything is based upon MapReduce, e.g. Pig or Hive. Since both groups usually appeared together, a certain style of architecture came into being. This triggered the name Lambda Architecture [1] by Nathan Marz (Figure 1).

Real Time

Unfortunately, most people understand something totally differently when they hear the name Real Time. This terms refers to the capability of a system to deliver results in a set timeframe. Breaking controllers, medical devices and many parts of satellites need to be Real Time-capable to prevent catastrophes. A brake needs to react quickly when a pedal is pushed, otherwise the driver will get a serious problem. It’s not about “the answer comes as quickly as possible”, but about “the answer will come within the time span”. Much more appropriate would be the term Near Time to describe what Big Data and Fast Data applications are trying to become.

With Near Time, incoming Data (1) is consumed by two layers. Within the Batch Layer, long distant runner analysis is being processed on the deployed raw data. The results of this analysis will be provided to the Serving layer (4), where they can be demanded by clients. The Speed Layer (3) relinquishes performance hungry persistence solutions and fulfills most of its duty in the main memory. Since it can never get hold of all data at the same time, it must focus on a subset. Certain results can thus be determined a lot faster. In the end, the aggregates will also move into the serving layer and can be related to the results of the batch layer. On closer inspection, the bidirectional relation between serving and speed layer stands out. Certain values can be accessed as distributed in-memory data structures e.g. distributed counters and can be grabbed live.

Redundant logic can be problematic

In these three layers, many technically elaborated frameworks have been established and have bred success in recent years. But growing success triggers bigger requirements. New analysis should always be formulated faster and in a more flexible way. Results needed to be seen in different resolutions: second, minute, hour, or day. MapReduce quickly reached the limit of what is possible. Thanks to its flexibility and significantly lower response times, more and more analysis moved into the speed layer. However, this did not result in less logic in the batch layer. Since the in-memory processing is limited by the size of the main memory, many studies still
need to be carried out in batches. The often incompatible programming modules meant that a lot of logic needed to be implemented several times. Such redundancies could lead to severely different results when evaluating the same data sources. At this point, it is clear why a unified programming model which covers large areas of the analysis is desirable.

The birth of Fast Data
In 2010, the AMPLab at the University of California, Berkeley published a new open source analysis tool which should solve this exact problem. Spark was donated to the Apache Software Foundation in 2013 and has undergone an impressive development ever since. In essence, everything regarding Spark revolves around the so-called Distributed Resilient Data Sets (RDD): distributed, resilient, parallelized data structures. These can be used in conjunction with many different modules:

- Processing of graphs (GraphX)
- Spark SQL, to deal with data from various structured data sources (Hive, JDBC, Parquet etc.)
- Streaming (Kafka, HDFS, Flume, ZeroMQ, Twitter)
- Machine Learning based on MLib

Besides Scala, Spark also supports Python and R. This makes it easier for data scientists to use it if they are familiar with one of the two. As you can see from the above-mentioned list, Spark is a fairly connection joyous framework and is thus able to combine many of the existing data sources in a unified API. The resulting analysis framework has prevailed rapidly.

The combination of structured data sources and streams makes it possible to combine much of the speed layer with the batch layer into a single interface. Analysis can be performed in almost any resolution. Spark jobs can even be deployed and developed by non-developers in an astounding timeframe. The arguments for Spark are quite clear:

- Scalability – to deal with millions of data sets
- Fast enough to provide answers in Near Time
- Suitable to implement analyses of any duration
- A unified, comprehensible programming model to handle various data sources

But even Spark has its limits. Tools for data delivery and persistency are still necessary. This is where we can resort to the experience of recent years.

Enter the SMACK
One often hears of Spark in conjunction with the SMACK stack – the combination of known technologies from different areas of Big Data analysis into a powerful base framework. The acronym SMACK stands for Spark, Mesos, Akka, Cassandra, and Kafka.

At first glance, one might assume that somebody has opened the hipster technologies box. I suppose you have at least heard of any of the aforementioned frameworks before. But I would nevertheless like to explain the tasks they fulfill in the stack context briefly.

Apache Mesos [2] is a kernel for distributed systems. It represents an abstraction layer over a cluster of machines. Rather than deploying an application on one note, it is, alongside one requirement description (number of CPUs, required RAM etc.), passed to Mesos and distributed to appropriate machines. That way, several thousand machines can be specifically utilized pretty easy. Mesos is the central nervous system of the stack. Every component of the SMACK stack is available in Mesos and perfectly integrated into its resource management. In addition, the commercial version Mesosphere is already available on Amazon AWS and Microsoft Azure. A convenient cloud data center can thus be built in record time.

The reactive framework Akka [3] is based on the known author model from Erlang. In recent years, Akka has evolved into one of the leading frameworks for distributed, resilient applications. In this context, it is mainly used in ingestion range and as access layer in the serving layer.

Another member of the Apache ecosystem is Cassandra [4]. Cassandra is a distributed, resilient, scalable database capable of storing gigantic amounts of data. It supports the distribution across multiple data centers and survives the concurrent failure of multiple notes. In this case, it is used as primary data storage. Apache Kafka [5] is often considered a distributed messaging system – and that is true for the most part. In fact, it is nothing more than a distributed commit log. Its simple structure allows users to transfer huge amounts of data between a number of systems, and thereby to scale linearly.

When put together, they form a solid base for Fast Data infrastructures (Figure 2): Akka (1) consumes incoming data like MQTT events, click streams or binaries and writes it directly into corresponding topics in Kafka (2). Now that the data persists, we can decide how fast we want to get different answers. Various Spark jobs (3) consume the data and interpret it into different resolutions:

- Raw data persistency: A job that writes incoming raw data to S3, HDFS or Ceph, and prepares it for later processing.
- Speed Layer: Implementation of “quick win“-analyses, whose results are measured in seconds.
- Batch Layer: Long-term analysis or machine learning
Results are written to HDFS (5) and Cassandra (4), and can be used as input for other jobs. In the end, there is Akka again as HTTP layer to display the data e.g. as a web interface.

Automation clinches
In addition to technical core components, automation is a key point in determining the success or failure of a real Fast Data platform. And Mesos already provides many important basic components for that. Nevertheless, we will continue to need tools like Terraform, Ansible, Kubernetes and comprehensive monitoring infrastructures. At this point it should be clear where I am heading: Without DevOps, it is difficult to achieve the goals set. Cooperation between developer and operator is essential for a system, which is intended to elastically scale and work on hundreds of machines.

To Scala or not
Scala is notoriously the parting of the ways. However, I want this article to deliberately initiate another debate on language features. In this particular case, the normative power of reality slams because every framework used is either written in Scala or is very Scala-like:

- Akka: written in Scala; primary API in Scala, but also available in Java
- Spark: written in Scala; primary API available in Scala, Python and R
- Kafka: written in Scala; integrated into Spark and little direct coding is necessary (however, since 0.9 primary Java API)
- Cassandra: written in Scala; Interaction primarily takes place via CQL and Spark integration

A SMACK developer will not get past Scala code. Scala is the pragmatic choice if you want to succeed with the stack.

Conclusion
Hadoop is not dead. In the future, HDFS, YARN, ZooKeeper and Co. will still remain important components of our Big Data world. However, what changes is the way in which they are being used. Existing components will, thanks to Spark, be usable within one unified programming model. Spark is just a tool. Much more important are the goals behind the catchphrase “Fast Data”:

- Low entry barrier for Data Scientists
- Differences between Speed and Batch Layer will disappear
- Exploratory analyses will be significantly easier
- The deployment of new jobs will be easier and faster
- Existing infrastructure is easier to use

SMACK offers a combination of all those goals and relies on proven technologies. The key lies in their use and in their highly automated combination. The result is a platform which is hard to beat in its flexibility.
Interview with Heiko Seeberger, Daniel Westheide, Daniela Sfregola, Julien Tournay, Markus Hauck and Ivan Kusalic

“Expert checklist – Why Scala and not Java?”

Which is the most popular JVM language and where are we heading to? We asked six Scala developers to weigh in on the state of Scala and answer some questions regarding the past, present and future of this programming language.

JAX Magazine: Why are you a Scala developer? What fascinates you personally about Scala?

Heiko Seeberger: What fascinates me about Scala is the possibility to write code that is not only concise, but also comprehensive. The best examples for that are the case classes that depict immutable value objects with “built-in” implementations of equals and hashCode and the pattern matching which is basically “switch on steroids”. In Akka, these features are used to define how actors react to messages (Listing 1).

If you want to implement this example into Java code, you would have to switch the one line case classes with multiple line classes including fields, getters and implementations for

```scala
Listing 1
...  
case class User(id: Long, name: String, email: Email)

case object GetUsers  
case class AddUser(name: String, email: Email)  
case class RemoveUser(id: Long)

class UserRepository extends Actor {  
  ...  
  override def receive = {  
    case GetUsers => // do stuff
    case AddUser(n, e) => // do stuff
    case RemoveUser(i) => // do stuff
  }
  
  override def postReceive = {  
    ...  
  }
}  
...  
```

Which is the most popular JVM language and where are we heading to? We asked six Scala developers to weigh in on the state of Scala and answer some questions regarding the past, present and future of this programming language.

**Interview with Heiko Seeberger, Daniel Westheide, Daniela Sfregola, Julien Tournay, Markus Hauck and Ivan Kusalic**

**“Expert checklist – Why Scala and not Java?”**

Which is the most popular JVM language and where are we heading to? We asked six Scala developers to weigh in on the state of Scala and answer some questions regarding the past, present and future of this programming language.

JAX Magazine: Why are you a Scala developer? What fascinates you personally about Scala?

Heiko Seeberger: What fascinates me about Scala is the possibility to write code that is not only concise, but also comprehensive. The best examples for that are the case classes that depict immutable value objects with “built-in” implementations of equals and hashCode and the pattern matching which is basically “switch on steroids”. In Akka, these features are used to define how actors react to messages (Listing 1).

If you want to implement this example into Java code, you would have to switch the one line case classes with multiple line classes including fields, getters and implementations for

```scala
Listing 1
...  
case class User(id: Long, name: String, email: Email)

case object GetUsers  
case class AddUser(name: String, email: Email)  
case class RemoveUser(id: Long)

class UserRepository extends Actor {  
  ...  
  override def receive = {  
    case GetUsers => // do stuff
    case AddUser(n, e) => // do stuff
    case RemoveUser(i) => // do stuff
  }
  
  override def postReceive = {  
    ...  
  }
}  
...  
```
JAXmag: Why Scala and not Java? In your opinion, what are the reasons to chose Scala over Java?

Daniel Westheide: In addition to the often mentioned powerful type system, there is an entire list of reasons why I would choose Scala over Java: I would like to emphasize two reasons which are somehow connected to each other. First of all, with Scala you are able to define algebraic data types. The other benefit is pattern matching which allows you to work with readable code and the aforementioned data types. The following example shows both pattern matching and algebraic data types in action. We define an algebraic data type session and discriminate between the session of a logged in user and an anonymous session. We then use pattern matching to return either a personalized suggestion or a general one (Listing 2).

JAXmag: Some people say that after Java 8 introduced lambda expressions, Scala lost a bit of its appeal because functional programming is now also possible directly in Java. What’s your take on that?

Daniela Sfregola: I don’t think Scala lost its charm after lambda functions were introduced in Java 8. Quite the opposite actually! Java is still missing a lot of features that Scala can offer such as implicits, for-comprehensions, traits, type inference, case classes, easy currency support, immutable collections and much more! Introducing lambda expressions in Java 8 is an opportunity for OO developers to start tasting the power of functional programming – before going all the way to the dark force with languages like Scala or Haskell.

JAXmag: An entire stack was formed around Scala; it consists of Akka, Play, Lagom, Apache Spark and others. How can we define this stack? Is it an alternative model for Java EE or Spring? Is it a loose set of interesting technologies or is there a closer relationship between these technologies?

Julien Tournay: It’s true that the Scala open-source community is very active. Spark is of course a huge driver of Scala’s adoption. The technologies you’re mentioning are mostly developed by Lightbend, the company behind the Scala language. Java EE and Spring are both large ecosystems so yes, projects in the Scala ecosystem are competing with projects in the Java EE or the Spring ecosystem. The projects developed by Lightbend share a vision of what they think Scala should be. They try to make everything they build usable to Java developers. For example you can use Play without writing a single line of Scala code. Of course there’s a trade-off here. Developing for both languages requires more time. It can also be hard to design an API that is usable by a Java developer, while not impacting the design of its Scala counterpart.

But just like the fact that Java is not limited to projects developed by Oracle (Spring is an excellent example of that), Scala is not limited to Lightbend’s initiatives. The work done by Typelevel is especially interesting to me. They are pulling Scala into more functional ways and are also building a coherent ecosystem. I think people coming to Scala from a Java background will probably start using Lightbend’s technologies and move some projects to Typelevel after a few months – once they’ve become comfortable with more functional ideas.

JAXmag: For what kind of applications can we use the stack?

Markus Hauck: The stack is really useful for creating applications that have to react very fast and / or scale to big amounts of data. At the same time, it is modular enough to give the user the opportunity to choose and use only those parts they really need.

JAXmag: Work has begun on Scala 2.12 . What do you find most interesting in this release?

Ivan Kusalic: I’m really interested in the style checker – Scala syntax is very flexible which is actually great, but as a consequence it requires extra effort to have consistency in a bigger codebase. In my team we currently take care of that in code reviews, but it takes a while for new team members to increase the speed.

JAXmag: Could you name something that you still miss in Scala and would like to be implemented in the next release(es)?

Heiko Seeberger: Scala has been around for awhile and has collected some burdens. Martin Odersky, the creator of Scala, is currently working on the next major release: 3.0. Some of the old things will be dropped once the 3.0 version is released.

JAXmag: Should Scala move more in the direction of a mainstream language like Java in the future (and possibly value more things like backwards compatibility)? Or would you rather welcome more innovative features (which could possibly break backwards compatibility)?

Julien Tournay: Backward compatibility is extremely important and one should be wary of breaking it at a language level. The Python community has learned this the hard way since the release of Python 3.0 – which, after all these years, has failed to exceed Python 2. Overall I hope the Scala language will continue to evolve and improve. I follow the work Dotty does (the next major version of Scala) with a lot of interest.

Read the full interview on www.JAXenter.com.

Listing 2

sealed trait Session
case class LoggedInAs(userId: String) extends Session
case object Anonymous extends Session
def showRecommendations(session: Session): List[Recommendation] = session match {
  case LoggedInAs(userId) => personalizedRecommendationsFor(userId)
  case Anonymous => trendingArticles
}
Modern DevOps – Connecting business and IT

DevOpsCon 2016: Our mission statement. This is how we interpret modern DevOps

DevOpsCon 2016 will take place in Munich between 5–8 December. What is the idea behind DevOpsCon? How does this conference bring together a broad range of topics such as Continuous Delivery to Microservices, Cloud, Container and Corporate Culture? Program-Chair Sebastian Meyen will give you some insights in this article.

by Sebastian Meyen

Modern DevOps: Connecting business and IT: Bringing teams from different fields together in a good way is rarely easy, when those teams are involved in the same business processes but do not work together directly. That’s why a group of people led by Patrick Debois suggested a new concept back in 2009: DevOps. They offered a solution to tackle the problem which exists in both development (Devs) and administrative (Ops) level. The DevOps movement developed substantially and made fundamental changes to basic concepts in IT and their roles in organizations.

Business-driven DevOps

Originating from the idea of making processes in conventional IT settings – classic on-premise-server, separated dev-and ops-departments – smoother, the DevOps movement is now mostly concerned with consistent digitalisation and areas with a high pressure to innovate.

Powered by the internet, many industries are subjected to an increasing pressure to change. While some are still looking back half-heartedly at their losses in traditional market shares, others are already making steps toward an open, hard-to-plan future. Consistent digitalisation and high-performance IT-structures are imperative – as demonstrated by renowned companies such as Netflix, Spotify, and Uber.

What exactly are the driving forces in business towards a DevOps culture (Figure 1)? Allow me to start by naming some (although certainly not all) buzzwords:

![Figure 1: DevOps culture](image)
Globalization results in increased competition in almost all industries.

The internet is more than just a modern marketing and sales platform for traditional fields of business. It has the power to transform classic business models, modify them or make them obsolete altogether.

Disruption is not an exception, but will be the norm in most markets. The ability to innovate will, therefore become the key to success for companies.

Therefore, markets cannot be perceived as stable, making long-term planning obsolete. Iterative strategies and many changes will become essential for companies’ success.

Five factors of DevOps

Modern DevOps does more than just bring together Devs and Ops; it aims to integrate business and IT across teams and systems. We would like to discuss the relationship between business and IT with speakers from around the world at our DevOpsCon conference which takes place between 5–8 December.

I will now try to outline the most important movements which can bring a sustainable change towards DevOps if brought together. I would also like to talk about what inspired us – myself and Peter Rossbach – to design the program of our DevOps conference. If we want to make extensive changes, the gradual improvement of conventional systems is not enough. We need to focus on the following aspects:

1. Continuous Delivery
2. Microservices
3. Cloud Platforms
4. Container Technology
5. Business Culture

Let’s take a closer look at each of these five factors and how they come together.

Continuous Delivery

Continuous Delivery – automating each and every aspect of delivery – has been an important aspect for online companies for quite a while. Bringing bugfixes, modifications and new features into production as fast as possible without taking too big a risk represents a very important goal.

Such companies usually don’t bring new software releases into production every six months; they don’t just do that every month or even every day but in most cases several times a day! Why is it that many small releases are better suited for such teams than just a few big ones? Because this prevents large backlogs from building up in the first place. Pending work? Doesn’t fit into the mindset of continuous delivery proponents. Important changes to usability or improvements to performance don’t have to wait until the next big release, they are put into production immediately. Even if that code does not stay the same too long, these modifications can also be rolled out without delay.

This culture of welcoming well thought-through experiments, encouraging all contributors (not just application developers) to try something new because they know a path once taken can always be corrected if new insights suggest so is part of the world we live in right now.

Continuous Delivery is putting gentle pressure on developers to optimize their software for smooth deployment. Developers will put more thought into architectural concerns and technical details that are important for deployment when they are responsible for transferring applications to real life, rather than just take responsibility for applications in test environments.

Microservices

Microservices are modeled with one goal in mind: to reduce complexity in software systems. The theory reads as follows: By “cutting” software into small “bites”, inherent complexity can be reduced. This revelation is added to the long history of ideas on modularity of software in IT (from object-oriented programming and component orientation to SOA and even modularizations like OSGi and Jigsaw.)

Dependencies between parts of the system, being responsible for problems around complexity, are eliminated this way; when working with microservices, they can be resolved by using APIs: When you change a service, you are obligated to consider “neighbouring services” to ensure the API stays consistent. You need to keep this important goal in mind throughout all development and deployment activities. If you have to change an interface, it’s easier to explicitly tell all neighbouring services and initiate a cooperative plan to kick off the change.

There is no need to use the same technologies for all microservices (one can be written in Java, another in Ruby on Rails, the next one in Go in the cloud …). Many experts see this as an advantage. We are merely mentioning this aspect as a side note; its relevance to the DevOps perspective is not a major one.

It is important to mention that microservices should not be seen simply as a new style of architecture which can replace other architectures and lead to better technical results. Microservices represent a new solution not only for technology but also for the organisation. It makes sense to use microservices when you wish to change certain things beyond technology. These encompass:

1. Autonomous, self-organising teams, each taking full responsibility for one (micro-)service.
2. Technical considerations are not the driving force behind the design of such services; functional considerations are (explaining the vast popularity of domain-driven design in the field of microservices).
3. “You build it, you run it”, this quote by Werner Vogels (CEO at Amazon Web Services) is a description of the responsibilities of microservice teams. They are not just responsible for developing an application, but also for its full lifecycle, meaning deployment, monitoring, bug fixing, optimizations, further development …
4. Furthermore, microservice teams are often cross-functional – that is to say, there might be operations/platform experts in the team, in addition to application developers; quite often domain experts, marketing specialists and designers join the team too.
5. Microservices are beneficial in the long run only to such organizations which are seeing them not just as some technical innovation, but as a way to follow their business goals.

Cloud
Modern cloud platforms represent more than an opportunity to transfer applications to public data centres. In fact, they are offering plenty of technical services which are challenging the conventional ways of building and using software.

Where can the consequences of this cloud paradigm be seen in reality? You need to put some serious effort into deploying an application in classic environments: You must choose an operating system and set it up, add application servers, database, manage users and permissions, configure a firewall, manage compatibilities and dependencies. Having done all this, the application itself finally needs to be configured and adjusted to any given production environment.

Modern cloud environments such as Amazon Web Services, Microsoft Azure or Google Cloud Platform make this process substantially easier. Complicated infrastructures from the traditional on-premise-world are almost trivial in comparison! Data management, user and permissions management (identity), networks, management and monitoring, scaling are at hand as services in such environments. Calling one of those services takes just seconds to complete.

Container
Container technologies, made popular by Docker, solve a long-standing problem of the software world in an elegant way: How can you make software running in one system context run easily in another system context? Containers separate software from factors like operating system versions, library-conflicts or network topologies, making the “transfer” of an app from the test environment into production system less “painful” than before.

What is the difference between Docker or CoreOS (to also mention the alternative container platform besides Docker) and traditional virtualization? Classic virtualization systems bundle not just the application, but also the corresponding operation system and further components into a package. Therefore, a machine needs to run three operating systems in addition to its primary OS when running three virtualization instances.

Docker and CoreOS virtualize only the application plus selected libraries while using shared access to services from the system kernel. This leads to decreased start-up times in comparison to classic VMware instances. With those, start-up takes up to minutes; with Docker, it takes seconds. Because of these properties, Docker virtually invites developers to split complex applications into small bites in microservices style.

Business culture
None of the above will start a revolution in IT on its own. Taken together they are able to change the tune unequivocally.

Continuous Delivery opens up new mental spaces in companies to shape their digital business. In the light of high releases frequencies, IT can be seen as an agile and ductile medium instead of a rigid shell that needs huge investments to initiate change.

Microservices facilitate such automated deployments by substantially reducing volume and complexity of the artefacts to deploy. They help companies focus on business goals, meaning that they do not let software infrastructures defined in the past decide the layout and focus of departments; instead, they help companies focus on goals and services that make sense business wise.

Furthermore, crossfunctional microservice teams promise to nullify classic boundaries between specialty departments/marketing/design/development/operations and, therefore, encourage different stakeholders to collaborate in a lively, multidisciplinary way focussed on the success of the product. When teams, put together like this, cultivate a communication culture in the spirit of Agile, without constraints from hierarchical structures, iterative development of products guided by (changing) customer needs is being facilitated. An agile infrastructure as defined by DevOps can supply a likewise iterative IT.

Cloud platforms help such iterative businesses on a technical level as solely software-based infrastructures; Docker containers are helpful too. Those make deployment and changes to the infrastructure a no-brainer and could potentially dispose of the image of IT being the “party pooper” in business development once and for all.

Final thoughts
In our understanding, modern DevOps will work only when combined with the techniques presented in this article. The topics – Continuous Delivery, Microservices, Clouds, Docker, Business Culture – might be subject to controversial discussions; certainly there are many important topics to be mentioned.

Furthermore, not all of the ingredients mentioned above are needed to build a “real” DevOps. In fact, I’ve heard of true Continuous Delivery miracles that are based on one single mega-monolith (instead of microservices) and of intelligent applications of microservices completely without clouds and container.

There is no “true DevOps”, just as there is no one authoritative definition of DevOps. Defining DevOps as being the description of a movement that forces us to rethink our basic understanding of IT, should do for now (and, of course, will be discussed at our DevOpsCon, that did not gain the motto of “Rethink IT” by chance).

I’m looking forward to many sessions, keynotes, and workshops, as well as lots of enthralling impulses and enlightening conversations. All of our speakers are well-versed in DevOps environments, some as infrastructure experts, some as application developers in the cloud, others as transformation consultants for companies.

Will I see you at DevOpsCon?
No software architect can resist the temptation to talk about their experience with microservices. We asked an expert to talk about the benefits and challenges of microservices, when people should not use them and what impact they have on an organization. In this interview Daniel Bryant, Chief Scientist at OpenCredo, agreed to talk about his likes and dislikes about microservices. Here are his answers.

**JAX Magazine: Why did you start using microservices?**

**Daniel Bryant:** I first started using an architectural pattern that was similar to what we now call microservices on a project in 2011. The reason we chose to use a service-oriented approach was due to the early identification of separate areas of functionality within the overall requirements. We also had several teams involved in creating the software (spread across the UK and also Europe), and we believed that dividing the system into well-defined services with clear interfaces would allow us to more efficiently work together on the project.

The development of separate services, each with its own interface, functionality, and responsibilities, meant that once the teams understood and designed the overall system-level requirements and interactions, we could minimize the need for constant inter-team communication. Well-defined service contexts and less communication meant that we delivered valuable software more quickly than if we had all been working (and coordinating) within a single codebase.

**JAXmag: What is the most important benefit of microservices?**

**Bryant:** When a microservice architecture is implemented correctly, the most important benefit is agility – in particular, the ability to rapidly change a system without unintended consequences. This means that as customer requirements (or the market) changes, the software delivery team can quickly react and adapt the software to meet these new requirements, and do so without worrying that a small change will create unforeseen issues (or require large amounts of testing to prevent regression). The properties of a microservice-based system that enable this benefit include:

- Understandable and well-defined cohesive services based around a business function (i.e. bounded contexts)
- Well-defined service interfaces (APIs)
- The ability to make assertions about functionality throughout the system stack, at a local and global level (e.g. component tests, contract tests, and end-to-end tests)

**Portrait**

Daniel Bryant is the Chief Scientist at OpenCredo. His current work includes enabling agility within organizations by introducing better requirement gathering and planning techniques, focusing on the relevance of architecture within agile development, and facilitating continuous integration/delivery. Daniel’s current technical expertise focuses on “DevOps” tooling, cloud/container platforms and microservice implementations. He is also a leader within the London Java Community (LJC), contributes to several open source projects, writes for well-known technical websites such as InfoQ, DZone and Voxxed, and regularly presents at international conferences such as QCon, JavaOne and Devoxx.
JAXmag: Have microservices helped you achieve your goals?

Bryant: The use of the microservice architectural style has definitely helped in several projects I have been involved due to the reasons mentioned in the previous answer. I work mostly as a consultant, and so am in the privileged position to see lots of different projects. Although microservices aren’t a panacea (and I haven’t used them in every project), they are a very useful pattern in my “architectural toolbox”, and I have used them to help teams I work with understand fundamental software development paradigms/qualities like coupling and cohesion.

JAXmag: What do you think should be the optimal size of a microservice?

Bryant: As a consultant, I like to say “context is vital”, and so I believe there is no optimal size for a microservice. My recommendations are to keep services focused around a cohesive business function (e.g. user service, payment service etc), ensure that the team can use a ubiquitous language within each service (i.e. a concept within a service means only one thing – for example a “user” within a payment service is simply an identifier for a payer), and make sure that a developer can readily understand the service context and code after a couple of hours of investigation.

JAXmag: What is the biggest challenge of microservices?

Bryant: One of the primary challenges is not getting caught in the hype. Resist the temptation to build everything as a microservice – it is simply an architectural style, and like all such styles, has strengths and weaknesses.

Another primary challenge includes testing – microservice systems inherently have more dependencies that must be managed, and testing the interactions between services can be challenging. This is why I have been working with the SpectoLabs team on the development of various tools to support the testing of microservice-based systems. We have an open source “service virtualisation” and API simulation tool named Hoverfly, which includes a JUnit Rule for use within a typical Java testing process and CI pipeline.

JAXmag: What technology do you think is the most important as far as microservices are concerned?

Bryant: In my opinion, microservices are technology agnostic, and the pattern can be applied with any programming language. Implementing microservices does, however, increase the need for a good platform i.e. deployment/runtime fabric, such as Cloud Foundry, Kubernetes or AWS EC2. Although I like to encourage developers to cultivate “mechanical sympathy” and understand key properties of the deployment fabric, I also believe they shouldn’t have to be overly concerned with things like process scheduling, configuration and service discovery.

Container technology, such as Docker, rkt and LXD, is also quite complementary to microservices, and these homogenise the deployment and runtime packages, but I don’t think this is essential for microservice development.

JAXmag: How much do microservices influence an organisation? Can they leave the organisation unchanged?

Bryant: If the spirit of the microservice architectural style is fully embraced — creating services based around business functionality – then the potential impact on an organisation is massive (particularly for a traditional enterprise), as business teams will have to re-organise away from horizontally functional silos (e.g. finance, marketing, PMs BAs, developers, QAs) to vertically integrated cross-functional teams (e.g. the conversion uptake team, or the user signup team). Many people have already written about Conway’s Law, and so I won’t cover this here, but I’ve witnessed the results enough time to know that this is a real thing. In fact, it’s worth noting that many of the pioneers in the microservice space started developing architectures that we now recognise as microservices because of business requirements for agility, team autonomy and decreased time-to-market.

Many people have already written about Conway’s Law, and so I won’t cover this here, but I’ve witnessed the results enough time to know that this is a real thing. In fact, it’s worth noting that many of the pioneers in the microservice space started developing architectures that we now recognise as microservices because of business requirements for agility, team autonomy and decreased time-to-market. This wasn’t necessarily a top-down edict to componentize systems like there was with the traditional SOA approach.

JAXmag: Why should you not use microservices?

Bryant: I get asked this question quite a lot, and it is difficult to give a definitive answer. However, situations where microservices may not be beneficial include:

- Working with a simple application domain
- Developing software with a small number of people in the team
- No problem/solution or product/market fit e.g. you are a startup that is still exploring your business domain and how to make money
- When the organisation is not embracing DevOps principles (i.e. see Martin Fowler’s MicroservicePrerequisites article).

JAXmag: What are your guiding principles in the division of an application into microservices?

Bryant: Firstly, make sure you understand your domain. I am a big fan of Domain-Driven Design, and so readily recommend the use of Context Mapping in order to fully understand (and model) your application domain. I’m also finding the technique of Event Storming increasingly useful to help build a dynamic (event-driven) picture of the system over and above the static picture provided by Context Mapping. Once you have modelled your domain, I typically find that natural service boundaries emerge.

“In my opinion, microservices are technology agnostic.”
Other techniques include the use of code analysis, both in terms of complexity (e.g. cyclomatic complexity or more crudely, lines of code) and churn (e.g. code change over time, as shown by VCS logs), and the identification of natural friction points (e.g. interfaces, potential seams, locations where data is transformed). Adam Tornhill’s book “Your Code as a Crime Scene” is an excellent resource here, as is Michael Feather’s “Working Effectively with Legacy Code”.

JAXmag: Should every microservice be written in the same language or is it possible to use more languages?

Bryant: I like the idea of polyglotism, both at the language and data store level, as this embraces my personal philosophy of using the “right tool for the job”. However, context is again key here, and if an organisation is struggling with understanding core architectural patterns or using (and operating) a single language platform, then adding more into the stack will only cause more trouble.

JAXmag: What aspects should you take into consideration before using microservices?

Bryant: As a consultant, I like to conduct organisational “health checks” before embracing any new architectural style or software development technology. These health checks would typically cover things like:

- Is the business/organisational strategy well understood i.e. do the development team know the overall mission and goals, and also understand why they are doing what they are doing
- Is the organisation’s leadership well-aligned and are teams capable of working together effectively; is the organisation “operationally” healthy – are they embracing “DevOps” principles, and do they have a well-functioning continuous delivery build pipeline
- Are developers/QA being well trained on architectural patterns, domain modelling, and technologies;
- Is there a process for feedback and learning e.g. retrospectives, career development programs etc.