EPISODE 489

[INTRODUCTION]

[0:00:00.3] JM: In the 1980s and the 1990s, most applications used only a relational database for their data management. In the early 2000s, software projects started to use an ever increasing number of data sources. MongoDB popularized the document database, which allows storage of objects that do not have a consistent schema.

The Hadoop distributed file system enabled the redundant storage and efficient querying of high volumes of data that are spread out across multiple commodity disks. The Cassandra Database is a hybrid between key-value storage and column-oriented storage.

The benefit of these different data systems is that you can choose a system that gives you the read and write performance that you need. The downside is that each of these databases has different querying semantics. If you're a developer trying to access data from your application, you often need to know how to access that data from the specific data source and whether that data needs to be queried with SQL, or with the document style query, or with a MapReduce job.

Spring Data is a project to standardize the programming model for data access within Spring. The vision for the project is to give Spring developers a consistent way to access their data from any database, or retaining the performance characteristics of those databases.

Spring is a Java framework for writing web applications, but this conversation is useful even for people who are not building these Spring applications. Whatever application you're building, you are probably pulling from multiple data sources. The question of how to abstract away the complexity of those multiple data sources is also being tackled by projects such as GraphQL and Falcor.

John Blum is a staff engineer who works on the Spring Data Project at Pivotal. He joins the s how to discuss how to design a data access layer. We discussed the API between a database and the Spring Data layer and also talked about reactive programming. Reactive programming allows the application layer to respond to changes in the underlying data layer.

I interviewed John at SpringOne Platform, which is a conference that is organized by Pivotal, who full disclosure is a sponsor of Software Engineering Daily. This week's episodes are all conversations from that conference.

If there's a conference that you think I should attend and do some coverage at, please let me know. Whether you like this format or not, I would love to get your feedback. We have some big developments coming for Software Engineering Daily in 2018, and we want to have a closer dialogue with the listeners. Please send me an e-mail jeff@softwareengineeringdaily.com, let me know what's up. Or join our Slack channel.

Thanks for listening and let's get on with this episode.

[SPONSOR MESSAGE]

[0:02:58.4] JM: Simplify continuous delivery with GoCD, the on-premise, open-source, continuous delivery tool by ThoughtWorks. With GoCD, you can easily model complex deployment workflows using pipelines and visualize them end to end with the value stream map. You get complete visibility into and control over your company's deployments.

At gocd.org/sedaily, find out how to bring continuous delivery to your teams. Say goodbye to deployment panic and hello to consistent predictable deliveries. Visit gocd.org/sedaily to learn more about GoCD. Commercial support and enterprise add-ons, including disaster recovery are available. Thanks to GoCD for being a continued sponsor of Software Engineering Daily.

[INTERVIEW]

[0:03:58.9] JM: Jon Blum is a staff engineer at Pivotal. John, welcome to Software Engineering Daily.

[0:04:03.0] JB: Thank you, Jeff. Glad to be here.

[0:04:05.2] JM: You work on Spring Data, which is a programming model for data access within Spring. I'd like to ease us into a conversation about Spring Data by starting with just the ways

that people access data when using Spring. What are the different ways that people are accessing data in a spring project?

[0:04:27.1] JB: There's several different types of data stores that a user might use. A relational store obviously is the most common, but now we have things like Mongo, Redis, GemFire, Geode, Cassandra, Couchbase. I mean, that storage are sort of very large and growing and probably will continue to be that way.

Spring Data gives you a way to access those repositories in a very common way. It all started from these core Spring framework, you know of the JDC template and so on, with JDBC access, with databases and then having integration with ORM tools, like Hibernate and TopLink and so on. Now, we're extending that to obviously the NoSQL stores as well. Spring Data represents that clutching of data access technologies across a diverse set of stores.

[0:05:12.5] JM: Let's talk in more broadly. Why do people use different databases for different situations? Redis is an in-memory, key-value store, as opposed to MySQL, which is a on-disk database. Why am I choosing different database in different circumstances?

[0:05:32.1] JB: Well, you might have different types of data that you're storing, right? If I have a document, I might want to store that in Mongo. It makes then a story and a structure that's conducive to the type of data that you have. Key-value store, very conducive to caching for instance; if I caching an application either in a service tier, or even my data tier between the application and the database, then a key-value store makes a lot of sense there.

You look at things like graphing databases, like being able to search and make relationships between different types of entities. A graph database is really applicable there. I think you want to match type of data source with the type of use case that you're having.

It largely depends on your queries. Cassandra is a good example of that, where you design tables around the queries that you're writing. Not necessarily around the way you would represent that entity naturally. It's all catered to how you're accessing that information.

[0:06:22.7] JM: An example might be if you are designing a database to have a query across three billion users, if you want to be doing queries regularly for three billion users, that's a much different type of querying than typically querying for a single user and getting all the fields for that single user. There is just different databases that accommodate different types of queries.

[0:06:47.8] JB: Absolutely. I mean, also for different types of application purposes, right? If I'm searching for a large number of people cause a social graph, that's a lot different than if I'm just pulling up some customer records because somebody called in. Maybe someone named John Doe calls in and I want to do a quick search, because the person doesn't know their account number of policy number or something like that. Very different ways to manage those types of data.

Also the data structures are very different between data stores too. Obviously, in a relational world you have the relational model, you have tables, you have rows and columns and so on. A lot of modern data storage we're seeing unstructured data, or partially semi-structured data and be able to access that data in the most conducive way possible. Like for searching for instance. Solar, Lucene, they provide a very rich API to basically search for textual-based information, which isn't easy to do in something like a database, because it's not geared towards that specific type of use case.

[0:07:45.8] JM: Yes, of course. We're talking about the different data use cases at the data layer. When we're talking about manipulating data at the application layer, we pretty much want the data to be in the same format once we bring in to memory and start doing stuff with it. Like when I make a query for a user to get it out of my database, I don't care if it's in MySQL, or Cassandra, or Hadoop, or whatever. Once I pull it into memory, I want to be able to perform operations on that regardless.

We want some way of unifying all of these different database representations into a common mapping. This was the historical reason for having what is called an ORM, the object relational mapper. Can you explain what the role of the object relational mapper is?

[0:08:41.1] JB: Well, its primary responsibility is to take your entity or your object that you have represented in your application in whichever language and map it to a relational structure, so

your tables, your columns in those tables and across tables. So depending to how denormalized or normalize your data is, you're going to break that data up.

One of the reasons for that might be to reduce duplication and overhead in the database as to amount of information that you're storing. In other cases, you might want to actually denormalize that data so that you can run more efficient query.

Again, it always has to be driven from some used case. What's the purpose of my data, right? If it's transaction processing, it probably needs a very normalized structure. If it's analytical processing, you're going to denormalize that to get more performance out of your queries.

Unfortunately, there really isn't a holy grail for doing the mapping between different types of data stores, and that's how these NoSQL stores have become quite popular, because I'm actually storing the object and memory. I'm not actually – I'm not having to perform any mapping. I just have to accommodate a few things for the underlying data store. If it's a key-value store, I obviously have to have a key that maps to my object. But I don't have to transform that object anyway if I don't want to. I store it in memory.

It makes it easier to do other things, such as realization. I need a persistent information. I can serialize that object to disk. I don't have to come up with some intermediate format that persist this information. I can just serialize the bytes and deserialize those bytes when I read the information back in. It reduces the amount of overhead, I guess in maintaining a data model, as well as the mapping infrastructure that always persist your data and retrieve it.

[0:10:17.9] JM: I think of the ORM, and you're much more experienced in this than I am so correct me if I'm wrong here. But I think of the ORM as sitting between the developer and the querying layer, so that the developer can make calls to the data in the native language that they're coding in.

If I'm a Java developer, I should be able to code against that native language. I should be able to code against that data in Java. I shouldn't have to write a Mongo query, or a MySQL query, or a Hadoop query every time I'm asking the database from my data. I should just be able to have

a Java query, essentially a query of a command that gets turned into a query through the translation layer of the ORM. Would you say that's accurate?

[0:11:05.5] JB: Pretty accurate. I mean, the whole idea behind ORM is mapping. Its primary responsibility isn't to deal with querying. In fact, it's probably not really ideal to do queries, because you can't do complex queries. There are query API that Hibernate provides to be able to do more complex queries.

At the end of the day, it's probably just easier just to straight JDBC if you have really complex queries in or outer joins, that sort of thing. Not many people that I know of, at least in applications that I build up in the past have I used ORM tool to really manage my queries. It is a great tool for mapping to my relational database. I'll probably use some other means to actually query that information.

Now if we compare that with something like that Spring Data, the nice thing about Spring Data is in addition to providing the code operations, or create update to lead, we also provide querying and we do it in a natural way, which is an object-oriented way. We have a lot of conventions where you can specify a method on your interface.

First of all, I should back up and say that we provide mapping or we provide data access through an interface that basically each store provider implements for the user. Out of the box, all you have to do is extend an interface and you get basic credit querying operations, like find line ID or something like that.

We also give the user the option to actually create additional methods. If you follow certain conventions when you create these methods it will generate the queries and text for you. Now obviously, it's really difficult to create complex queries just following a naming convention, especially one that's conducive across a wide number of data stores.

Each store provides specific hooks where you can customize a querying engine a little bit, to do things like joins or maybe perhaps invoke a function even to manipulate that data in some way. Each store is a little bit different in that regard, but they all start from that common repository abstraction.

The idea behind that is I can invoke a query just like I would any object method. It just looks like a method. I have an instance of a customer repository. I have customer repository about find by name or something like that. I don't have to write the query for that – the underlying framework and infrastructure smart enough to figure out how to create their appropriate query for that data store.

[0:13:11.3] JM: What are the goals of the Spring Data Project?

[0:13:16.3] JB: Well, the primary goal is to provide a common programming model for all the different types of data stores that we actually integrate with, so things like Mongo, Redis, GemFire, Geode, Cassandra, Couchbase, anyone that has a query engine that can support repository infrastructure. We want to give our users a very common look and feel to data access regardless of the data store.

In many ways, these repositories are actually, I guess you could say transferrable across the different data stores. We don't really generally recommend that, because each data store is a little bit different in terms of how index this data, some of the query semantics, like inner joins or outer joins and so on aren't necessarily transferable across all data stores. Generally, the repository abstraction is very generic and looks very consistent across all the data stores.

[0:14:08.3] JM: Okay. What is the repository abstraction?

[0:14:10.8] JB: The repository is an interface. It's essentially what we used to call the data access object. It provides, again it just provides basic CRUD operations, you Create, Read, Update, Delete, as well as some basic queries, or some simple queries. Like I can find an object by an ID or something like that. I can find one, I can find all. Real simple queries. Anything that's pretty common across majority of the data stores that we support, relational or non-relational or NoSQL.

In addition to that, it's an extensible mechanism so it's an interface, it's just a specification of what I want my data access to be, and each provider provides a default implementation of the back that's extensible, so users can write their own queries.

They have really advanced use cases that can specify annotations and say, “You know what? I just want to write this query myself.” Of course, Spring Data will delegate to that. The repository abstraction is a way to basically provide an interface to your data store.

[0:15:02.6] JM: Let me understand this correctly. I could build an application, where I have a query to my Spring Data repository and the repository implementation is sitting over a MySQL database. Then I could port that same query to repository that hits a Cassandra database. The exact same query would run despite the fact that these are totally different databases?

[0:15:29.2] JB: A very similar query would run. But again, you have to be careful there because the way index in Cassandra is very different than the way you would index in a database. You have to have the notion of a secondary index if you include a particular column of your table in a predicate for instance. Otherwise, Cassandra will most of the time fail without that index.

There is also other things that distinguish Cassandra from a relational database, right? Where the degree of normalization in Cassandra is pretty denormalized, like I’m creating a table specifically for this query. In simple use cases, I would say yes, you could probably reuse a repository.

When you start getting a little bit deeper and your queries become a little bit more specific geared towards your business use case, or performance is a concern, then you’re looking at maybe slightly different queries, but the abstraction is fundamentally the same. I still have an object that I’m invoking and it invokes a query under the hood. It generates different syntax of course, because the database has SQL and Cassandra has a CQL, right? Different query language, but effectively the same effect.

[SPONSOR MESSAGE]

[0:16:40.5] JM: If you are building a product for software engineers, or you are hiring software engineers, Software Engineering Daily is accepting sponsorships for 2018. Send me an e-mail, jeff@softwareengineeringdaily.com if you’re interested.

With 23,000 people listening Monday through Friday and the content being fairly selective for a technical listener, Software Engineering Daily is a great way to reach top engineers. I know that the listeners of Software Engineering Daily are great engineers, because I talk to them all the time. I hear from CTOs, CEOs, directors of engineering who listen to the show regularly. I also hear about many newer hunger software engineers who are looking to level up quickly and prove themselves.

To find out more about sponsoring the show, you can send me an e-mail or tell your marketing director to send me an e-mail; jeff@softwareengineeringdaily.com. If you're a listener to the show, thank you so much for supporting it through your audienceship. That is quite enough. But if you're interested in taking your support of the show to the next level, then look at sponsoring the show through your company.

Send me an e-mail at jeff@softwareengineeringdaily.com. Thank you.

[INTERVIEW CONTINUED]

[0:18:07.4] JM: Let's explore what this repository object is doing for us. If I'm accessing a file in HBase, for example; HBase is a columnar database that's built on top of the Hadoop file system. If I'm pulling a file out of HBase, this is probably going to be an enormous file that I'm going to need to manage and Spring Data might have to do some management to access that file in a way that is comfortable to the application developer. What would Spring Data be doing in that example?

[0:18:41.5] JB: Well, that's I guess, maybe a tough example for me to answer since I'm not really familiar with HBase, but I could maybe compare and contrast let's see, like a SQL database with say, a NoSQL store.

[0:18:53.4] JM: Sure. That's much safer.

[0:18:56.0] JB: Yeah. It's probably not all that much different. I don't actually know if we actually even have a Spring Data community module for HBase. Not that I know of. I know we have

several community modules for Couchbase and elastic search and other ones. HBase doesn't – I don't think it fits in there.

In a SQL world, let's go back to the example, like I have a customer I want to look him up by name. I would just specify him that it called something like find by name, or find by first name and last name. That's a convention that we would use on the repository query method that users could refer to the documentation and figure out how to write for the particular data store.

Under the hood, the JPA provider or spring at a JPA will actually look at that query method, parse it by name, generate the SQL statement and then any method parameters that are actually passed to that method, or any method parameters that that method declared actually use these arguments in that query.

We can inspect that method. We can parse that method by name, generate the appropriate SQL and use whatever runtime arguments that the user pass to that method when they invoked in that repository and pass it into the query.

How is that different from something like GemFire? Not much actually. GemFire uses a query language called OQL, which stands for Object Query Language. What is generates is something really similar to SQL. In fact, if you look at the two they're almost indistinguishable, other than the fact that I'm not accessing the table in the database. GemFire, there's a cache and there is regions. A cache is really similar to a schema in a database and the regions are really similar to tables.

The syntax is a little bit different and obviously, I can do things like property notations so I can do like customer dot address, dot city and something, so I can use this dot notation to drill into my object model. That's the object part of the query language.

The repository abstraction is smart enough to know that, okay if I have a customer and I said, "Address city," it knows that it's a address object on a city and I'm accessing that city property or field on that address object. It generates appropriate query for that data store to access it. Again, it just uses the method arguments that I passed in, and I can invoke this query in an object-oriented way.

That's what the repository abstraction is doing for you. It's giving you a Spring programming model, and then have it only the way to basically do a CRUD, as well as querying operations and sometimes quite complex querying operations depending on the underlying store support. That's mechanically what we're looking at.

[0:21:27.4] JM: Yeah. If somebody has a database that they want to write a compliant – I think the term is module. If I have a new database, if I have JeffDB and I want to write a module for it to have it work within Spring Data, what would I have to do?
[0:21:51.7] JB: If we look at like what Spring Data JPA does under the hood, that's a great question because it doesn't necessarily generate directly the SQL for every particular store. Like in JPA's, Spring Data JPA's case is going to use entity manager API and the query objects to generate the query.

It's indirectly generating this query through JPA provider, like Hibernate. Then in GemFire world, it's using GemFire's objects or query service to generate. It's OQL CRUD access that database. If somebody would implement a data store for their own – excuse me, a Spring Data repository abstraction for their own data store, then there is certain abstract types that we provide from a framework perspective that they have to implement to basically parse the query methods, generate the appropriate queries by accessing the underlying data store's API.

On Cassandra's case, there is a driver and that driver provides a query API that we use to actually generate the query. A lot of the times it isn't just the roll like select star from some table, like there's some API that we can use. In JPA's case, it's Hibernate or some JPA provider. In Cassandra's case, it would be the data stack's client driver and there's a query API behind that. In GemFire's case, there is a query service that you can use to get a query object, construct your OQL that way.

Some of them provide a very object-oriented, or builder style API to construct queries in a tight safe way. Other ones like GemFire, Geode don't do that, so you're specifying more raw SQL. The implementer of the repository abstraction would do whatever is appropriate for their particular data store and implement through abstraction. But we provide the common types that

you implement to fit in within their infrastructure, so that essentially to the user it doesn't look any different. To a Spring user, it won't look any different.

[0:23:41.1] JM: Okay. I think if I understand it correctly, there are these collection of – there is this abstractions that whether you are building a Cassandra module, or a Mongo module, or a MySQL module, you have to implement those different abstractions in order to be compliant with the Spring Data interface, is that correct?

[0:24:02.6] JB: If you're a framework builder, if you want to extend the Spring Data infrastructure to support a different data story, yes. If you're a developer just using the Spring Data repository infrastructure, you don't have to worry about any of that. All you have to do is pick a provider, specify your interface for your application you're up and running.

I may have misunderstood your earlier question in terms of if I wanted to build a new data store abstraction for Spring Data, or if I just wanted to use an existing one. If I'm using any supported one; JPA, Mongo, Redis, GemFire, Cassandra, all supported, we've already implemented that within the framework itself.

If somebody wants to build one for a new data store that we don't support, like you've mentioned HBase earlier, there are certain types that we built into the framework that you implement that would plug into the repository infrastructure. That would give them some momentum in building it out. They have to get pretty specific when it comes to their data store, when they generate the SQL in however is appropriate to do that.

[0:24:56.0] JM: That's actually what I'm curious about is if I have this brand new database and somebody made a breakthrough in database technology and it's got some weird stuff going in there, but presumably it's still a database. That whatever abstraction you've built that plug into Cassandra and MongoDB and all the different modules that people have written in the past for Spring Data, presumably you can fit those abstraction to working with this new database that I've written, that I've created.

What are those abstractions? What are the things that every new database that onboards on to the Spring Data platform has to comply with? This is just a – I'm just curious architectural-wise.

[0:25:38.2] JB: Sure. Under the hood, everything begins with a repository interface. That gives you the type that you're going to be persisting, like if it's a customer, you'd be specifying a repository for, like say a customer type. The repository abstraction gives the developer the users of that person's data store abstraction a starting point.

Under there, the implementer is going to look at things like, we have a query method abstraction that abstracts out a query method on a repository interface. If the user of that particular data repository abstraction specified a query method, then in Spring Data commons, so we have – I'll back up just one second real quick and say that we have a common – Spring Data commons project from which all the other Spring Data projects extend, like Spring Data GemFire, Spring Data Cassandra, Redis and Mongo.

They all build on this Spring Data commons framework that specifies the repository abstraction, that specifies the notion of what their query method is. The query type gives you things like, is this a paging query for instance? Can I page my results? Can I sort my results? There is abstractions for all those different capabilities that a query engine might provide, like paging, sorting, providing limits, whether I can enable debugging or tracing in a query and that sort of thing, even perhaps indexing.

We provide a hierarchy and a type library of all the different things that a query engine, data access, repository or object might contain that a particular provider like HBase or Cassandra would implement based on the features they support.

By way of example, GemFire does distributed joins or outer joins as such, because it's a very complex problem to implement. We won't provide any kind of cross-region capability to query across multiple different regions for instance. GemFire it has to be co-located. Actually, that's quite a bad example. Maybe a better example might be, let's take paging. Not all data store support paging. GemFire doesn't support paging for instance.

There is an extension of the repository interface called CRUD repository. In top of CRUD repository, there is paging and sorting. For data storage that support paging and sorting of results, GemFire supports sorting, but not paging. Those aspects of the Spring Data repository implementation and Spring Data GemFire and Geode are left out. They're not implemented.

Then basically, the user will get a nice friendly message to say that page is not supported. Or they won't necessarily be able to provide a repository abstraction that implements the paging and sorting interface. If they tried to do that, then basically the module will tell them the page is not supported.

Sorting is supported, so that's something that GemFire supports and there is types for in the Spring Data commons framework to support sorting, like sort by fields in ascending and descending order. There is a sort class, there is a direction class to specify which direction you're sorting, either ascending, descending. There's all these abstractions that represent what different capabilities that a query engine and a particular provider provides.

[0:28:40.5] JM: My step one with implementing JeffDB that is Spring Data compliant is I extend the repository.

[0:28:48.0] JB: You want to create a implementation of the CRUD repository. If your database provides simple create, read, update, delete, so the repository abstraction, the repository interface itself is just a market. But on top of that, there is the CRUD repository which extends repository with your CRUD operations.

Your first job as a data JoeDB implementer is to provider in limitation for that simple CRUD repository interface. Then beyond that, you're going to start looking at how do I support the user's ability to create queries? Maybe by conventional over-configuration, right? I don't want to have to specify whatever CRUD language that your database comes up with. I want to do it through the method, the query method abstraction, so then you'll start looking a things like, "How do I parse the method name and how do I represent that internally?"

That's where some of the types in Spring Data commons' framework comes into play like a query method. We have a thing called the part tree parser that looks at the query method name

and can parse it out. We provide implementations for all those things based on what kind of operators your query language supports, what kind of joins, like ands and ors between your conditions and so on, or what kind of product it is, I should say. Not joins.

Just depending on the syntactic structure of your query language, what it supports. There is different types you can implement to help you build into the Spring Data commons, or repository infrastructure for your database.

[0:30:09.8] JM: Now based on most of our conversation so far, people who are totally new need to stop and might just be thinking, "Okay, so this is basically like a translation layer of my Java code into a query that is specific to this underlying database." But you alluded to something there, unless I misheard you that that was pretty interesting, when you're talking about Geode. Geode, well Apache Geode. We actually did s show on this a while ago, which it's distributed, while GemFire is I think the productized version of it, is that right?
[0:30:45.3] JB: Yeah, so Apache Geode is the open source version of Pivotal GemFire. Pivotal GemFire is the commercial version of Pivotal Maintain Sys.

[0:30:52.2] JM: This is a distributed in-memory database. Basically, you got the speed of an in-memory store, but you get some, I think some durability because it's replicated. Am I articulating it correctly?

[0:31:05.9] JB: Yeah. You get what's called replication. You get replication distributed across the cluster, and durability you get the persistence aspect of it. The traditional asset sends, I could persist my transaction. It's not lost. It's durable. It doesn't lose –

[0:31:20.8] JM: Persistent to disk, or persistent through – by virtual being replicated?

[0:31:23.9] JB: Persistent to disk. Both Pivotal GemFire, Apache Geode support persistence and different data management policy, so you can enable persistence and it writes to its own disk store, disk format. It's called an uplog and it's an append-only log, similar to other types of data stores. Of course, compaction and there is require, but it's sure nothing architecture so there is no single point of failure there, there is no master-slave type relationship.

[0:31:49.8] JM: What I was to say is I thought I heard you allude to something where if you issue a query, just a naïve query from your application layer to a repository that is a Geode or a GemFire repository, I should say, so it's a distributed in-memory store, you might have things going on at the repository layer where it is managing data across multiple regions. You might have some distributed query going on, some more complex, multi-region stuff that can potentially happen?

[0:32:26.3] JB: Yeah. By default, repositories don't support the notion of joins, at least not in the NoSQL space very much as like compared to say the relational world. Partially, because your data could not be on the node in which you're querying. You can join regions, so long as they're co-located. So within a single node that you're querying, the data must all reside on that node.

There's no concept of distributed joins. In order to implement something like that, you need to use other mechanisms that Geode and GemFire support to do a more distributed query, or like I'm going to run this query across my nodes that contain the data that I'm interested in. You take advantage of other features, like function execution to be able to do that.

That's similar to Hadoop's MapReduce. I can send out my functionality to the data as opposed to bring the data to me and then act on it. You would use GemFire's function execution in that case to do a more distributed like join, because you have to take the query to where the data lives and you can only join it when that data is co-located on the same node.

You might have two regions, region A, region B. If you're joining those two regions, they have to exist on the node in which that query is being executed both regions to join that stuff. There is no real notion of a distributed join. I can't join region A with region B if region A is on node 1 and region B is on node 2. The repository abstraction doesn't support that either, so you have to implement that yourself.

[SPONSOR MESSAGE]

[0:33:59.5] JM: You are building a cloud-native application and you need to pick a cloud service provider. Maybe you're just starting out with a new app, but you have dreams of scaling into the next giant unicorn. Maybe your business have been using on-premise servers and you want to

start moving some of your infrastructure to a secure cloud provider that you can trust. Maybe you're already in the cloud, but you want to go multi-cloud for added resilience.

IBM Cloud gives you all the tools you need to build cloud-native applications. Use IBM Cloud Container Service to easily manage the deployment of your Docker containers. For serverless applications, use IBM Cloud functions for low cost, event-driven scalability.

If you like to work with a fully managed platform as a service, IBM Cloud Foundry gives you a Cloud operating system to control your distributed application. IBM Cloud is built on top of open source tools and it integrates with all of the third party services that you need to build, deploy and manage your application.

To start building with AI, IOT, data and mobile services today, go to softwareengineeringdaily.com/ibm and get started with countless tutorials and SDKs. You can start building apps for free and try numerous cloud services with no time restrictions. Try it out at softwareengineeringdaily.com/ibm.

Thanks again to IBM for being a new sponsor. We really appreciate it.

[INTERVIEW CONTINUED]

[0:35:36.4] JM: I guess, I like to dive into to GemFire, Apache Geode a little bit more and then maybe talk about how the implementation of Spring Data for that specific database was done, because I know you worked on that recently. I think that will shed more light on how Spring Data works.

Just to reiterate, GemFire is a productized version of this Apache Geode project, which is an in-memory distributed, replicated persistent data store. Why do people use GemFire? What is it useful for? What kinds of transactions?

[0:36:14.6] JB: Historically, it's been used in stock market transactions to basically manage trades at a very high volume. Given its scale of nature, linear scale versus up in on a – adding more disk or CPU, just adding more nodes to my cluster to commoditize cheap, obviously scale

out infrastructure that Facebook and other companies like Google and Amazon popularized, and having that and memory is very appealing because reduced the latency and there is no disk seek.

Depending on how it managed my data across that cluster, I can get high read and write throughput. There is all kinds of appealing ways, or appealing reasons to use it. Historically, it's been used in high-volume trade transactions, because of its performance. That's where it started its roots and now it's making its way towards the cloud now, because it's becoming applicable to our cloud-native patterns there. Yeah, historically it's started as a financial system of record, kind of asset-compliant database system or record store.

[0:37:16.6] JM: Geode is – you said it's in-memory, but you can also have durability. Is all of the data in Geode also written to disk? It's all durable?

[0:37:29.6] JB: You can turn on persistence by default, so you can say I want my region to be persistent and then write all operations to disk, either synchronously or asynchronously. You can also just overflow the disk. Maybe you have readings that you don't want to be persistent. It's not necessary to keep that data around, but given the disparity between memory and disk, obviously disk exceeds the capacity to memory many times over, but the order and magnitude of speed and memory is obviously a lot faster. Any data that it's being operated on inside GemFire and Geode of course is in-memory. Then any data that's not frequently used or at least recently used gets overflowed to disk. There is always that option to configure either overflow, or persistence to write the disk.

[0:38:11.3] JM: Okay. When you were implementing the Spring Data repository specifically for Geode, give me some examples of things that you had to do specifically for that repository that you probably would not have had to do on some other repository implementation? Does that question make sense?

[0:38:32.6] JB: Yeah, a little bit I suppose. There is, I guess OQL and the capabilities in GemFire are a subset to like say SQL, for instance was a lot more, I guess power and SQL in general then there is an OQL. Although, the languages are different. Obviously, there is different advantages and disadvantages to each approach.

Obviously, SQL doesn't have the notion of object notation, nesting of particular fields or properties, accessing stuff in a object-oriented way. In terms of the implementation, it was pretty straightforward. Again, we start with repository abstraction and we build on what kind of capabilities GemFire support. Well, because GemFire supports persistence, or supports adding data to GemFire and pulling data out, it made sense to extend the CRUD repository abstraction.

I started there and we implemented the CRUD repository operations to create the read, update, delete. The read operations are like your find by ID, find all exist, count OQL language supports the notion of account. It's like the account star by some region, or from some region. It made sense to implement all the basic data access operations that GemFire supported, as well as the basic queries and then to extend it from there.

What kind of operators does the query language support for instance? Things like, I can select something that's in a particular set, or I can join two predicates like with an and, or an or. I can do order by clauses. You start adding on to that repository implementation based on the capabilities of the store.

There is a point where you reached the peek at which the query engine for that story gives you. In GemFire and Geode's case, I mentioned that it doesn't support paging, there's no notion of a cursor. It's not as easy to page data engine fire as it is a relational database when you have the notion of cursor and fetch sizes and all that stuff, the OQL language doesn't support that.

That part of the repository implementation, I didn't implement. We just put in messages to say that this feature is not available. Or perhaps it is available and they – driver matures, and then we go back and we build it in. Depending on where this data, the data store is out of the box, it may or may not be something that we have to implement initially.

[0:40:44.1] JM: Because I'll take it on totally different. Elastic search, which is basically a search index, which is technically a database. I make search – when I make a search query on an elastic search cluster, that is the database. I believe Spring Data has an elastic search clause, or does it?

[0:41:02.5] JB: It does. Spring Data elastic search and it's a community-led module. We don't lead that internally. There's actually a handful of modules that are led by Pivotal and then a handful that are led by the community. We roll them all up into a single release based on the relationship we have with the providers of those data stores.

Hazelcast for instance, there is a Spring Data Hazelcast, which isn't actually a part of our release cycle, but it exists nonetheless. It's community module. Other ones, like Couchbase, or elastic search which are community-led modules, we do roll into our release train releases. It just depends.

[0:41:35.5] JM: Maybe you're not the right guy to ask about this, but just to further the point of this design of this repository abstraction, what would be some of the things that search index might want to implement if it's extending the repository, if you're the person who is creating the repository that is – the module that is compliant with the repository and you're making it for elastic search?

[0:42:00.5] JB: Yeah, that's a difficult one for me to probably answer, since I'm not qualified to really talk about elastic search. I guess, I can relate that to something in GemFire and Geode, because we recently added support for textual searches using Lucene. That's something that I haven't actually, I guess completely thought about how we would actually integrate that with the repository abstraction right now.

It's definitely something that I think would be worthwhile. Maybe I want to run like a Lucene textual search, but I want to invoke that in a repository programming model, an object programming model. I don't want to say my customer repository dot search for something, or search for like say a book by title, or search for all books in a particular category.

That might be not such a great search example, but you might look for certain text within titles or descriptions and you want to pull those out of your data store. That's something that I'm venturing in to a new space with on GemFire and Geode that I haven't thought about before.

Another good example, maybe not search related would be something like continuous query in GemFire. The nature of continuous query is to handle eventing. I'm interested in some events

that happen. This is really maybe common in internet of things, right? I've got these sensors and they're constantly feeding some particular backend data store with all the information that's coming in from that sensor.

Maybe I'm interested in like a temperature gauge or something like. When it goes below a certain threshold or goes above a certain threshold, I want to be notified of that, so I can act on it. GemFire has a notion which is called a continuous query, which is basically just using the OQL abstraction, or the OQL syntax to write a query that says, "Whenever data is updated or put into my system that matches the predicate of my query, tell me about it. Fire us an event."

I can register the CQ in it continuously runs in the background and I can register a listener with that CQ as well, and I can get notified anytime data changes that matches my predicate and my query. It's actually a really quite slick mechanism compared to the traditional ways in which interest registration is done. I'm usually registering an interest in the key, or I'm using some crazy rejects expression, regular expression to try to express what I'm interested in.

It's easy to not be able to capture the right information in that particular scenario, or either catch too much or too few. With the querying capability I can say, "Well, I want to just match on specific criteria." Then I get notified of that event. That's a good example of something I'm thinking about how to integrate with a repository abstraction.

Say right now, Spring is all about the reactive use cases. We're talking about reactor, we're talking about RxJava, the pub/sub, subscriber models and the model in the flux. The CQs would fit perfectly with what reactor calls a flux, which is a publisher of data. I can get information back and do that in a reactive way when that event becomes available.

The CQ only gets triggered when an event happens. Then the flux is essentially a continuous stream of events, so I can wrap those events in that flux and provide reactive support inside of Spring Data GemFire. That would be an extension to the repository abstraction that Spring Data GemFire provides today, but it's something I would have to implement.

Fortunately for me however, there is a bunch of new classes in Spring Data commons Kay. We renamed all of our release trains after famous computer scientists. Kay is our latest release of

Spring Data commons, or the Spring Data framework itself overall. Allan Kay, yeah the famous computer scientist Allan Kay.

Before that, we had Ingles and our next release train is actually going to be Lovelace, after Ada Lovelace. On Kay, we provided all the reactor support that match what's in Spring Framework 5. There's a reactive repository abstraction now that returns things like when I find by an ID, it returns a mono of that type. A mono meaning, I either have zero or one item in that particular stream.

The flux being I want to find all, or I want to find by people with last name Doe, or something like that returns you a flux in a stream. Data may not be available at the time that you're querying it. I don't want to block that thread, so it provides this flux. The nice thing about reactive is you really can't be reactive unless you're reactive across your entire stack, from the web front into the data tier. Spring Data provides that abstraction for reactive data access for the stores that support reactive.

Some of the stores support it like Mongo and Redis. Other stores like Cassandra also have an implementation, but it's built around their async API, and of course bringing to GemFire, Geode doesn't have anything yet. But I think a natural pairing there would be to pair that with the CQ functionality, because it's asynchronous in nature and we could wrap it with types that people are familiar with, either pub/sub, from RxJava, or you have your flux and mono from reactor. That would be something where I would extend Spring Data's repository infrastructure to support that. Like I said, fortunately I can build off some stuff that we've already put in Spring Data Kay to make that a little bit easier for me to do.

[0:47:01.8] JM: I want to unpack some of the things you just said. First, this model of reactive programming; I think of this and correct me if I'm wrong, in the sense that I've got some service that is interfacing with my database. There is another service somewhere that also interfaces with the same database.

I am the service owner of service A, some other person is the service owner of service B. They are just communicating with the database. I want to be able to know when service B has made a change to the database. I want to be able to react to the change that service B has made to

the database. What you said there is in order to be reactive, you need to be reactive throughout the entire stack. The reason for that is the database that service B writes to needs to push a notification to service A, is that correct?

[0:48:03.9] JB: If you think of like messaging architecture where you publishers and subscribers and queues, if I'm a publisher I don't have to care who subscribed. I just have to publish to a queue, and then anybody that's interested can subscribe to that. GemFire's CQ functionality is the same mechanism there, right? I can register a CQ for specific events that happen within a region.

You can actually use GemFire as a messaging system as well. I can write events or data into a region. When it matches my criteria as I mentioned before based on a career predicate, I will get notified about that. The whole idea behind reactive is I don't want to be blocked waiting for something to happen.

I want to be notified when something happens. It's an inventing-type architecture that doesn't tie up resources for things that quite frankly I don't know when it's going to happen. If the temperature changes 1 or 2 degrees, I don't know when that's going to happen, but I'm definitely interested when it does happen, so I don't have to have a thread that constantly pulls and listens for that. It can get notified asynchronously when that even happens. That's the whole idea behind the reactive architectures. I don't have this blocking infrastructure that ties up my application resources to specifically look for things. I just get towed when those things happen.

It's a natural extension for us to build that infrastructure from into Spring Data, because it extends all the way back to the frontend when I'm looking at writing – interacting with users. They come and go, or they send it, request, and depending on what type of information they're looking for, that information may or may not be readily available. We need to basically react to things in a very reactive way, and that's how reactor and the Spring Data reactive extensions are built into that.

[0:49:52.0] JM: If I want to at my application level make a continuous query to a repository, what is the right interface between me, the application developer and the repository that is capable of doing continuous reactive queries?

[0:50:13.6] JB: Essentially, I just implement a listener. I just subscribe. I just want to be notified when this happens. I'm just going to subscribe with the publisher or the source of that data and get notified when an event happens. A very specific event; I could be specific about which events I'm interested in, so that's where that query comes in.

Effectively, we only have to have a common queue, if you will, where we collaborate through. A publisher writes things to this queue and I receive things from this queue based on my interest. It's a really simple mechanism. Essentially, you have a publisher that's publishing and writing and you have a listener that's reading and pulling the information when these events happens. Or technically, it's getting pushed these events when these events happen. It doesn't even have to actually pull for anything.

It's a really simple interface. It's not much different than your observer-observable pattern that we've known forever. It just manifested itself in different ways, now that we've got new frameworks and tools for expressing that. Whether it's our old message, pub/sub, or it's our new reactive type world.

[0:51:18.0] JM: Sure. If I'm the application developer, so let's say I'm developing just – I'm a random developer building an application and I'm going to instantiate a repository that is backed by let's say GemFire; the GemFire database, and I want to subscribe to changes in that underlying GemFire database. You're saying that the programming primitive I'm using is the subscribe primitive?

[0:51:47.9] JB: This is where I've been thinking about extending Spring Data's GemFire's repository infrastructure to basically have it just be another query method on a repository interface.

[0:51:57.9] JM: Continuous query.

[0:51:58.1] JB: A continuous query in this case. Not a regular OQL query, where I just want to go out and fetch that data and I block and wait until that data is available to me. In this particular case I want to say I'm going to invoke a – say a repository method that returns – excuse me, that registers itself as a CQ and it gets called in an inventing mechanism that gets notified of the events when that CQ gets triggered. I'm sorry, continuous query.

[0:52:24.4] JM: Continuous query. All right.

[0:52:25.3] JB: Whenever I refer to CQs that just mean continuous query. They write a query that's continuously run by GemFire in the background. Every repository is always tied to some data source. It's a very table, or in GemFire's case it's a region. On that region, that can create using just the normal semantics of OQL, or I can register interest in particular events and I can use GemFire's continuous query functionality to express my interest that way.

That could potentially become a repository cray method on a Spring Data GemFire repository implementation. What is returns me in that case isn't specific data, it returns me a flux. In that flux, we'll actually take care of handling the events that come in and I can register callbacks with that flux. In the reactor world, it's a flux I think in RxJava. I'm forgetting what the type is specifically theirs.

[0:53:18.3] JM: Specifically there?

[0:53:19.9] JB: Yeah. There is publisher and I can't think off the top of my head. Anyhow, in reactive world I'm very familiar with that since we – Spring is all centered around reactor right now. That would just be a flux, because it's a continuous stream of events. Or in the Java 8 world, maybe that's a better example. It's a Java util stream stream.

We actually have a streamable type in Spring Data commons, so easy to confuse those two. It's Javautilstream.stream, so I could return a stream and that stream could just be endless. It could just be an endless stream of events until I cancel my subscription. I'm no longer interested or whatever.

It's not blocking. I'm not waiting for something to go out, execute that query on the servers and get the data back. I'm just basically calling this method express that I'm interested in receiving events when they happen. When I get back as that flux and I can treat that flux harder. I can register listeners with it, I can trigger different actions when those events happen.

[0:54:17.3] JM: You said you're deliberating. Should I extend the repository interface itself to have this notion of a continuous query? Because maybe other databases in the future will want to implement the continue – Sorry, maybe other repository implementations in the future will want to implement the continuous query methodology. Is there some deliberation? Are you pretty sure that you want to implement this continuous query?

[0:54:42.5] JB: First of all, this is very specific to Spring Data GemFire, so continuous query as a terminology and a functionality that's provided by GemFire and Geode. In our Spring Data commons framework we have what's called a reactive repository and more specifically the reactive cred repository.

When you say find by ID, it returns you a mono of that particular type, because that data may not be available yet. I might have to go and aggregate a bunch of data and then it returns that data. But it doesn't block. The moment I call find by ID, it immediately returns me a mono. In Java 8 world or even the Java 5 world it returned me something like a future, right? A Java Util concurrent future. Then I can invoke that future at some point later and it blocks and it waits to get that data.

In the reactive world, it's a little bit more slick than that. I can just register a call back and then I can be notified when that data is available. It's very similar in concept, or at least in style to what the future is. The CQ mechanism is specifically specific to GemFire and Geode. I can wrap that in Spring Data commons' reactive support, or reactive repository and a reactive types like the mono and the flux.

I can provide a very similar programming feel for something that's very GemFire and Geode-specific continuous queries. For instance, Mongo doesn't have the notion that I know of of continuous queries for instance. Other stores provide maybe a similar functionality, but they don't call it continuous query. Trying to reduce the amount of GemFire-Geode specific

terminology and put it in terms that basically users are going to understand by relating it to our common abstractions in the core framework.

[0:56:16.2] JM: I know we're up against time. I just got a couple more questions. We touched on event sourcing idea a little bit. Basically the idea with event sourcing is changes to the overall application architecture get published to this global event stream that other subscribers can read from and change the underlying databases to respond to for example, but the event stream is this single source of truth for the whole application architecture.

The event stream is a data source basically. If I'm building a Spring application and I have an event sourcing architecture, am I interfacing with the event stream through a Spring data repository, or is there some other abstraction that I'm typically interfacing with? Like maybe some sort of pub/sub abstraction? Or does that question make sense?

[0:57:14.7] JB: It does. The answer to that is pretty simple. It's just the flux. I would basically trigger my interest in these events, these set of events that I've specified that I want to know about to the CQ. Then what I get back is that flux and that's all I have to interface within that point.

The active basically saying, "Okay, I want to register the CQ and be notified of events," is as simple as saying, "Take the repository for abstraction, call this method, get back to flux, and now I can use that flux from that point forward to basically receive the events and process them however I choose."

At that point, it's similar to the pub/sub, right? I'm observing some observable thing and maybe what I get back is a future, or maybe I get back an observable object of some type, or I get back a consumer or I mean, I won't get back to publisher, because the publisher is the one pushing that data into for me to consume. I would have some interface that I could implement that I could receive these events.

In these particular case, we've already got the interface, we've already got the abstractions in place with reactive types with the mono and the flux. Because it's an endless stream of events, I just take that flux and I can register other things like a listener and say, "Well, when I received

an event I maybe do some additional filtering within that flux to say, 'Well, I want to take en route this event for here and I want to take event B and route it over there. I can do whatever I want with events that come in.'"

Going back to our temperature setting for instance, like I have high and low thresholds, I have – temperature is too hot, temperature is too low. Maybe I want to do something different based on the low temperature than when it's a hot temperature. My engine is overheating, my engine is too cold, or whatever the case might be. I can do that filtering at the application level, because it's probably very application-specific. I want to act on differently depending on the event. I am interested anytime I have an event, where the temperature is either too high or too low.

[0:59:13.8] JM: Got it. Okay. Last question; we've been talking around this idea of reactive programming. What are the trends that – zooming out, what are the trends that are leaving more people towards adopting reactive programming? Why was that such a focus of the keynotes today of we're at SpringOne Platform? The keynotes, there was a lot of stuff about reactive programming.

[0:59:39.9] JB: That's a great question. I think it's twofold. In my perspective, I think it will also depend – depending on who you ask. From a technical level, I'll start there because at the end of the day it all boils down to how many resources you're consuming, and the less resources you can consume to accomplish your task, obviously the more efficient you are and the cheaper it's going to be.

If you think about things at cold scale, it's really easy to burn through your cost at a very large scale and infrastructure like Amazon, Google, or Microsoft. Something you have to be at least mindful of and you want to use your resources intelligently and efficiently, so you can maximize your efficiency and your throughput and all that stuff.

On a more, I guess maybe fundamental level, I think it just makes sense that we were entering this world where actions are happening in real-time. If we want to respond to people's sentiment or their feelings or – we want to be able to be responsive to our users, our customers and we want to be able to ensure them that we're meeting their needs.

It's just, I guess in the simplest terms, it's being able to respond to change, because change happens. It's continuous, it's frequent, it's always going to be there. It's being able to act on it in real-time when it matters. If somebody's not happy with my product or my service, but I don't find out about it a couple days later, then maybe I've already lost my customer.

I think it's being able to be more responsive and manage things in a responsible way, which I think reactive gives us that paradigm. It gives us both the technical capabilities, as well as these more semantical capabilities are tied to whatever our business is about, right? Well, let's just be more efficient and more responsive and all that good stuff.

[1:01:23.5] JM: Okay. John Blum, thanks for coming on Software Engineering Daily.

[1:01:25.6] JB: Absolutely. Thank you, Jeffrey. It's been a pleasure.

[1:01:28.2] JM: Yeah. Great.

[END OF INTERVIEW]

[1:01:31.2] JM: Indeed Prime flips the typical model of job search and makes it easy to apply to multiple jobs and get multiple offers. Indeed Prime simplifies your job search and helps you land that ideal software engineering position, from companies like Facebook or Uber or Dropbox.

Candidates get immediate exposure to top companies with just one simple application to Indeed Prime. The companies on Prime's exclusive platform message the candidates with salary and equity upfront. Indeed Prime is a 100% free for candidates. There are no strings attached.

Sign up now and help support software engineering daily by going to indeed.com/sedaily. That's indeed.com/sedaily if you're looking for a job and want a simpler job search experience.

You can also put money in your pocket by referring your friends and colleagues. Refer a software engineer to the platform and get $200 when they get contacted by a company and $2,000 when they accept a job through Prime.

You can learn more about this at indeed.com/prime/referral. That's indeed.com/prime/referral for the Indeed referral program. Thanks to Indeed for being a continued sponsor of Software Engineering Daily. If I ever leave the podcasting world and need to find a job, once again Indeed Prime will be my first stop.

[END]