EPISODE 323

[INTRODUCTION]

[0:00:00.4] JM: Command Query Responsibility Segregation, otherwise known as CQRS, is a powerful concept that has the potential to make or reliable and maintainable systems. It is also broadly misunderstood and means different things to different people.

Derek Comartin learned about the idea of CQRS after viewing some talks by Greg Young and he has since successfully applied the CQRS approach with great success. It's transformed the way that he views features, business requirements and dependencies. The result is a system that is easier to maintain and aster to enhance. Among Derek's key lessons are the idea that slices are better than layers, mediators improve dependency management, and cohesion is better applied to business concerns than to technical ones.

In this episode, Derek is joined by Dave Rael, who is the host of Developer On Fire. He's guest hosting this episode, and they sit down for a conversation about CQRS and how to apply it to make code your own and to separate it from technical concerns in order to make your software development operation work better and faster.

I want to thank Dave Rael for guest hosting this conversation, and his show Developer On Fire is great. If you like this episode, let me know. If you don't like it, let me know. Give us any sort to feedback. We are piloting some guest hosts and we want to open up the floor to other voices. Please let us know your feedback.

[SPONSOR MESSAGE]

[0:01:42.4] 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. Candidates get immediate exposure to the best tech companies with just one simple application to Indeed Prime.

Companies on Indeed Prime's exclusive platform will message candidates with salary and equity upfront. If you're an engineer, you just get messaged by these companies and the average software developer gets five employer contacts and an average salary offer of $125,000. If you're an average software developer on this platform, you will get five contacts and that average salary offer of $125,000.

Indeed Prime is a 100% free for candidates. There are no strings attached, and you get a signing bonus when you're hired. You get $2,000 to say thanks for using Indeed Prime, but if you are a Software Engineering Daily listener, you can sign up with indeed.com/sedaily, you can go to that URL, and you will get $5,000 instead. If you go to indeed.com/sedaily, it would support Software Engineering Daily and you would be able to be eligible for that $5,000 bonus instead of the normal $2,000 bonus on Indeed Prime.

Thanks to Indeed Prime for being a new sponsor of Software Engineering Daily and for representing a new way to get hired as an engineer and have a little more leverage, a little more optionality, and a little more ease of use.

[INTERVIEW]

[0:03:32.0] DR: Derek Comartin is director of engineering at Full Circle TMS and the founder of windsorsx.net developers. Derrick, welcome to Software Engineering Daily.

[0:03:40.9] DC: Thanks for having me Dave.

[0:03:41.8] DR: Awesome. You titled your talk for the CodeMash Conference in January 2017 *Fat Controller CQRS Diet* and you have a series of blog posts and videos on your YouTube channel about this subject. Where did you get the inspiration to share this content?

[0:03:56.3] DC: Like I Kinda actually described in talk when I did at CodeMash was it stems from almost the beginning of my career creating various business apps and having difficulties along the way and evolving those apps. Kinda as I described in my talk and, really, what the subject of it was working at a distribution company in the app that we're creating kinda morphed overtime from being a native WinForm app, then moving along to an ASP.NET MVC app and trying to share code between the two and the coupling that was happening to the native WinForm app and the MVC app and the difficulties that that post.

[0:04:41.8] DR: Because there is a diverse audience that may not necessarily be Windows and .NET familiar, can you just tell me a little bit more about what are WinForm apps?

[0:04:50.5] DC: Yeah. A WinForm app is I would say the classic version of creating a native Windows kinda GUI application. That's what WinForm refers to. There's the newer version of creating those apps in WPF, Windows Presentation Foundation. That's kind of the native Windows UI side of the mix, and then there is —

[0:05:11.9] DR: Yeah. Okay. You're in a situation here where you've got this native Windows client and this ASP.NET application and you are dealing with kind of the several concerns of common functionality between them and some disparate functionality. That's kind of the context for going down this path and experiencing some of the pain that you felt in application development and some of that stuff. Can you set a little bit more context just describing some of the pain that you are experiencing?

[0:05:38.8] DC: Yeah, it's exactly what you're describing. In specifics, it was related too that the application had the commonality between order entry. Orders would come in to the system, in the native application. These were generally entered in-house, actually internally in the company. This goes back – Shelling the age of the application here. This goes back where a lot of our orders were mail orders, so that they would be sent in through physical mail and we had an actual data entry department that would be basically rekeying orders into the system.

As the web kind of evolved and it became important to have a presence online, having that same order entry functionality in our MVC app and — That was really the crux to the issue, because there was a lot of business logic that related to the items and the products that you could add to an order and what the prices may be at any given time. It was a constantly changing month-to-month set of business rules. Being able to share that code, not really just the — I should clarify. It's not really about sharing the code, it was more about sharing the concepts and the features in order entry between the two applications.

[0:06:50.1] DR: Okay. You had this great overlap of things that you were doing and that was just leading to a lot of duplication and a lot of, really, maintenance nightmare, I guess is kind of what I'm hearing there. Is that about right?

[0:07:01.6] DC: Yeah, just duplication, and this doesn't seem like a difficult problem of being able to share code between separate applications, and it's fundamentally not. How you go about doing that is ultimately what my talk was about with related to CQRS and how CQRS helped alleviate those – It was one solution of the problem, I guess.

[0:07:21.8] DR: Yeah, that makes sense. Two things obviously jumped out in that title, right? They're grabbing the attention. One of them is the emphasis on CQRS that you really just said there and the other is on bloated controllers, the idea that there's too much code in this web thing. I think, let's give a high-level overview of both CQRS and controller bloat and then we'll just dive into each of those and kind of the relationship between them. Does that sound like a good way to go forward here?

[0:07:48.0] DC: Sure. Yeah, that makes sense.

[0:07:49.4] DR: All right. Okay. Yeah, go ahead with – Not all of the listeners are necessarily going to be familiar with the MVC pattern and they may not know what we mean when we talk about controllers. I think that's a good place to start here, is can you give a quick introduction to what we're talking about with controllers?

[0:08:04.8] DC: Yeah. In most web frameworks, they're using MVC. A controller is essentially the entry point that the framework basically is invoking your controller and specifically a method that ultimately relates to a route to basically a URI in your application. For example, if we were talking about the application I was working in, we had controllers that would have many methods, which basically represented many actions within a control, many routes part of the application. Each one of these controllers' actions would be doing, like you said, just a variety of different concerns. They had a lot of concerns related to validation, authorization, just general data validation, business logic, data access, you name it, you name the concern – Blogging. You name the concern. It was in these controllers.

Ultimately, the controller was everything. All the code lived in controllers for the most part. Yeah, you could separate things out and have a service, whatever that means to you, and separate it to separate classes. At the end of the day, it was really about the dependency on your MVC framework was there no matter where you move the code to.

For example, if you wanted to access data in the QueryStream, there's obviously some way to do that. In the world of ASP.NET there is your HTTP context, for example, which is a dependency. If you're using that, which is a part of the controller, you're basically coupled to the controller. The way I describe it is your code is no longer your code. You have a direct dependency on MVC and you can't really get away from that.

[0:09:41.7] DR: Okay. I think you answered the next thing that I wanted to ask about there, is really just what's so wrong with having these different controllers? You gave a description that it's really just your code that is handling a request or some kind of action from the user and then you kind of described that having a lot of dependencies on the framework. That this particular code, it knows about the web, or it knows about that you're in a native Windows application and some of that stuff and you wanted to separate out your business logic from that thing that's specific to the technology that you're riding on. Why is that? What's so important about that separation?

[0:10:17.4] DC: Exactly, is that I think it came to the point where we realized, "Are we writing an application, or are we writing a web application?" There's nothing necessarily wrong with creating a web application that you're going to be coupled and bound to your web framework as long as you realize you're making that choice. In our particular case, we had multiple frontends, essentially, to the application. It's not that we're creating a web application. We are creating an application that had a web frontend. It also had a native Windows frontend. That's really where the distinction is. There's nothing necessarily wrong with coupling yourself to it if you're aware that you're making that decisions.

[0:10:57.2] DR: Okay. Yeah. This really facilitated that vision of being able to just lift up this code and put it into different places, and you are struggling with that because of this coupling of the code itself, the business logic to the specific platform that it was running on.

[0:11:12.1] DC: Yeah, exactly.

[0:11:13.4] DR: Okay. That makes sense. I think we kind of talked about the controllers there and why this was important. Let's move to the insight that you gained from applying CQRS. I'm sure there's a wide variety of listener familiarity with the idea of CQRS. For some, it's a new and unfamiliar term. Others may have heard of it and wants to know more. Still, others are probably applying it, probably with varying degrees of success and varying interpretations of what it might mean. Simply, put CQRS is confusing. A lot of us have different ideas about what it is, and we'll need to get into CQRS and what it means. I think, first and more fundamentally, why? What is the problem that CQRS addresses and why this is worthy of attention?

[0:11:57.4] DC: Yeah, there's a lot going on there with CQRS and what definition we can get into that. For me, it was really about a commander of query in that specific unit of what it is, of what its responsibility is. Whether it'd be you're mutating state of the system, or you're returning a value from the system. Thinking of it that way, once you go deeper than that, but that's just general premise of separating reads and writes, can be really beneficial and it has a lot of implications.

[0:12:27.1] DR: Okay. Yeah, and I think we will certainly get deeper into how that benefited you and some of those things. Let's talk about those terms. CQRS; Command Query Responsibility Segregation. You already mentioned, commands and queries being the big distinction there and segregating those things is kind of the crux of what CQRS is all about. What are commands and queries? I think you already started to touch on separating reads and writes and what that is. Why is it good to treat those things different and separately?

[0:12:58.7] DC: There's a couple of different reasons for me there, but I'm going to deviate a little bit and really more focus on at least the beginning of this conversation of what it is not, because you kinda mentioned it of what the confusion is, and I'm totally guilty of this early on when I started understanding what CQRS was and some of the blog posts, et cetera, that you would read. Really, it goes back to Greg Young who coined the term along with Udi. Greg has a post, it dates back probably now, seven —

[0:13:30.4] DR: I just want to clarify there. Udi Dahan was the reference when you say Udi there.

[0:13:35.2] DC: Yeah, Udi Dahan and Greg Young. Greg young has a blog post. I don't even know how old it is. It's fairly old. Where he mentions it's just not that complicated as what it's being made out today. It's really just simply a matter of separating the changing of state in your system as a command and returning a value as a query, and it's nothing more than that. I think where the problem lies is if people have a little bit of a misunderstanding with it is if you're searching for CQRS, you're going to find a lot of articles, or blog posts, or even samples on GitHub that are doing CQRS, but they're doing a bunch of other things with it; VR, either trying to demonstrate some concepts, like technical concepts that people may use in domain-driven design. They may be showing how you can do event sourcing. They may be using things like a service bus, or queues. They may be using multiple data stores. Yeah, CQRS facilitates. That's why I said it has some implications that it kind of enables you to do different things.

Ultimately, it's not that — Isn't CQRS. CQRS is simply the fact that you're going to have mutating state of your system, be a separate concern than reading the state of your system. That's it.

[SPONSOR BREAK]

[0:15:04.7] JM: You are building a data-intensive application. Maybe it involves data visualization, a recommendation engine, or multiple data sources. These applications often require data warehousing, glue code, lots of iteration, and lots of frustration.

The Exaptive Studio is a rapid application development studio optimized for data projects. It minimizes the code required to build data-rich web applications and maximizes your time spent on your expertise. Go to exaptive.com/sedaily to get a free account today. That's exaptive.com/sedaily.

The Exaptive Studio provides a visual environment for using back end algorithmic and frontend component. Use the open source technologies you already use, but without having to modify the code, unless you want to, of course. Access a k-means clustering algorithm without knowing R, or use complex visualizations even if you don't know D3.

Spend your energy on the part that you know well and less time on the other stuff. Build faster and create better. Go to exaptive.com/sedaily for a free account. Thanks to Exaptive for being a new sponsor of Software Engineering Daily. It's a pleasure to have you onboard as a new sponsor.

[INTERVIEW CONTINUED]

[0:16:36.2] DR: Yeah. I have the text that you used in your talk from that blog post by Greg Young in front of me here and I'll just go ahead and read what Greg said specifically in that blog post, "CQRS is simply the creation of two objects where there was previously only one. The separation occurs based upon whether the methods are a command or a query, the same definition that is used by Meyer in command and query separation. A command is any method that mutates state and query is any method that returns a value"

That's what Greg said about it. I think you do well there to say, "Hey, this is simpler than it's made out to be," and there's a lot of things — Event sourcing is obviously the first thing that comes to mind and something very near and dear to Greg Young's heart as well. Yes, it often complicates these presentations of it when you do that. At its core, CQRS is two things where there were previously one. You're separating that state out. I think it makes sense. When you interact with a web application — And it goes well with the idea of rest-based services and all of those kinds of things and with the HTTP verbs, that there are different things that you're trying to do in this application.

Sometimes you are creating a request to take an action. Sometimes you merely want information that is going to inform you perhaps in making some kind of action or something like that. I think that's probably a pretty good description that listeners can take something away from. Is this particular action that the user is taking, are they looking for information, or are they looking to change something? Are they looking to do something? Have we covered that? Do you think that that describes CQRS pretty well?

[0:18:11.3] DC: Yeah, it's simple at heart. It's really not overcomplicated, and that pretty much describes is.

[0:18:16.7] DR: Okay. Sounds great. You've described the context and some of the problems that you were dealing with and where it was that you were experiencing pain and what you are trying to accomplish, and you came across Greg Young. How did you find CQRS? Was there something specific that led you to thinking that this might be something that would help with your problems?

[0:18:37.3] DC: I can't remember specifically, but at the time, I was reading Eric Evans' — The blue book, the *Domain-Driven Design* book, and I think that's when I probably stumbled upon CQRS at the time. I think I was probably seeing a talk by Greg Young. I can't remember specifically what it was, but I think he made mention of it.

There was actually one specific talk that I do remember him talking about, which I don't even think he actually referred to CQRS that kind of led me down this train, which was I think the talk is called *21 and a Half Ideas or Tips to Become a Better OO Developer.* In it, one of them is basically referring to thinking of a method call being the equivalent of sending a message. That's when that idea at the time didn't make a heck of a lot of sense to me. As it further went along and thinking about how I could implement CQRS, that kinda led to some ideas, probably, with some other code samples that I've seen. That's when I stumbled upon the mediator pattern which I really feel kind of uses that idea a method call being equivalent of sending a message.

[0:19:47.6] DR: Okay. All right. Yeah. You've introduced, yet, another vocabulary term here. We're dealing with commands and queries. Now, you've started talking about messages. I think it sounds a lot like Allen Kay with the original definition of objective-oriented programming that objects communicate via messages. What is a message and how does it fit into this kind of soup of different terms that we're dealing with?

[0:20:09.7] DC: Specifically, with messages, in the sense of how I would use them in the context of something like the mediator pattern, it's really just an object that contains — Can contain information about the request that you want to make. A request could be a query or it could be a command. It could be requesting, "I want to fetch a particular piece of information from the system, or I want to," like I said, "mutate state in the properties within that object." Maybe the values that you want to — Whatever it is that you're actually changing. The idea is that the object represents the action that you want to take to the system.

[0:20:52.5] DR: Moving on into kind of more talking about this mediator pattern. What does that mean? That's another piece of this thing. What is the mediator pattern and how did you apply that in your approach?

[0:21:05.9] DC: Early on, when I was trying to figure out, "Okay. I have commands, queries, they're isolated — Specifically, what they do. How can I go about decoupling my controllers from my code from how I want to deal with these commands and queries?" The way that I stumbled upon this was — And I think this maybe happens to a lot of people often is, I was actually implementing the mediator pattern and I didn't really even know that I was doing it. I didn't even know that it was actually called a term the mediator pattern.

Essentially, what it is, is you are creating an object that understands how to communicate with other objects. If I have object A that wants to, for example, call some method on object B, it doesn't directly call that method on object B. It doesn't even know about object B. You're not taking a dependency on object B from A. you know nothing about it.

What you do is you take a dependency on the mediator. What you have is a request, I.E., an object, which is a message, and you send that, you invoke the mediator with that message and the mediator understands how to then construct or access the object on the other side and then invoke and then potentially pass the results back to where you originally called it from.

[0:22:25.7] DR: Sure. Okay. Really, rather than a highfalutin kind of, "Here are patterns — " Let's be academic about the pattern application. That sounds like a very pragmatic, just, "I've got this way of invoking these handlers for these particular types of commands and queries that I want to execute." Then, my dependency graph is greatly simplified when I can just say, "Okay. I've got the mediator to talk to," instead of having to have all of these different dependencies in my controllers. I think that probably contributes greatly to simplifying some controllers as well. Is that kind of one of the benefits that you've seen of this?

[0:23:03.4] DC: Yeah, exactly. Traditionally, your controllers — If you're injecting your dependencies in the constructor of the controller, if you weren't using something like the mediator, then you'd be taking dependency on every command handler, query handler, that you have in that controller. There's just no cohesion there. If you can just take a dependency on the mediator and then the mediator understands where to essentially route that request, if you will, for your commands and queries, then you're just taking a dependency on the mediator.

[0:23:32.3] DR: Yeah, I think that makes sense. It greatly simplifies things. You kind of touched there on dependency injection as well, and I think some listeners may not be familiar with that. It's essentially that you are specifying all of the things that this particular object needs in order to do what it needs to do. Why is it better to have fewer dependencies for a particular object in this case?

[0:23:55.8] DC: I think when it comes to your controllers — This is actually kind of what happens when you're dealing with commands and queries is that because those become essentially the unit at which you're working at, if each individual request that's a command or a query, ultimately, it's an object. It will have a class that represents how it wants to deal with that message and what it should do. Then, you can decide for those small little units what the dependencies are that they have.

Rather than having — For example, in a controller, where you have — Maybe you're taking a dependency on — I don't know. Some database connection, if you will. That database connection is only used in one method, for example. You're taking a dependency for one particular method in a class, which the only reason is, is because you just have these controllers that they're no cohesion between any of the methods. They're really just framework code. You know what I mean? That really has nothing to do with your apps. Just keeping those small, keeping the dependency small. The way I describe it is having your code be your code and not littering it through your framework code. Keeping that integration boundary between your framework and your code as separated as possible.

[0:25:12.7] DR: Yeah, I like that. Some people would say that if there's not a lot of cohesion in your class, then that's a smell already, but there may be a lot of reasons why; pathing and some things like that, where you may want some of those operations on the same controller, and your concerns in a web application may be different than they are through the rest of your application. I think that does provide kind of a nice way of keeping some separation of concerns among those things.

Another thing that, in my experience, testing, really, is something that can get very painful when you got a lot of dependencies on your classes. Can you talk about how it might help with your testing effort and to make things more maintainable if you have this smaller number of dependencies as well?

[0:25:51.9] DC: Yeah. When you start thinking about commands and queries, specifically in testing them, it really simplifies it, because then that particular unit, like I said, it really does only have dependencies on the things that that operation needs to do, that query or command actually needs.

For me, what I've been moving towards is testing what I consider a feature. More on like a vertical slice approach rather than layers. A feature to me could be a command, it could be a query, it could be a combination of them, if need be. The idea being is that you're testing a specific command or query, which is a feature. I don't tend to mock a lot for that particular reason, and I'll let the dependencies — Unless the dependency is something that's specifically out of my control that is external. It's really thinking about testing kind of in a vertical slice rather than — The unit for me, being a command or a query.

[0:26:47.7] DR: I like that. It's really appealing to be able to look at things in business cases rather than trying to test along technical lines and some of those things. I think that's really appealing and it simplifies things too. When you talk about mocks and some of that stuff and starting to have complex dependency graphs, it becomes a very difficult maintenance story just keeping your tests up to date. I think that's one of the barriers that keeps a lot of people from doing a lot of testing, is just the pain of having to manage all of that.

A layered approach in your talk and in some of your materials on your blog and YouTube, you compare and contrast a layered architecture approach to this idea of CQRS being kind of the trimming factor in a lot of what you've done now. Can you just describe too what I a layered architecture and what are some of the promises that it makes and some of the ways that it might fall short on that?

[0:27:40.3] DC: Yeah. Originally, when we started moving code from having everything in our controllers and in our WinForms, the first thing we moved towards was a layered approach. We decided to extract separate technical concerns, if you will; data access, business logic, different validation into their own layers, their own projects.

The idea there being is that you can ultimately refactor or rewrite any of those layers without interfering with the other layers. That sounds great on the surface, although it has some issues in that. If the layer becomes so large, the likelihood of rewriting an entire layer becomes seemingly less feasible.

Ultimately, what we ended up doing and noticing kind of this as a pain point was when we want to make small changes. A small change could just be the simple fact of maybe it's a new feature that we need to add some new data to the database. That required us to go into the data access layer and make a change to some object, maybe, for using an ORM. We have some class that represents the table and the database. We've got to make a change there. Then we'll go into some other — Maybe we're using the repository pattern at that time, so we're going into the repository and making appropriate changes to any number of methods within that class.

Then, we're jumping over to the business logic layer and making appropriate changes there to how data flows. Ultimately, one layer invokes the other. We had ended up going through from the UI all down to these layers, and you could have it anytime. You know what I mean? Four, five plus files open, scattered through different projects to make a simple change, which doesn't feel right.

Once we kinda felt this and we kinda realized it, "This doesn't feel right." What can we actually do about this? The idea was why are we organizing our code and writing our code with technical concerns in mind? Why aren't we writing code in use cases, in a feature in a way? Organizing our code and writing it that way? That's what we started doing? It's thinking of things vertically instead of horizontally. You may think, "Okay. Then, what happens to all my layers? Where am I going with this? I still want to separate data access and business logic," and you're still doing it, but you're doing it within the confines of a particular command or query. The benefits there is that, now, that technical concern is confined within that vertical slice, that feature.

For example, if you had a dependency on, say, some ORM, for example, and you are using that within a query to fetch data out of database and —

[0:30:37.7] DR: Let me just interrupt real quickly there. ORM, object relational map, where you're talking about a tool for getting data into your application.

[0:30:43.8] DC: Correct.

[0:30:44.2] DR: Please proceed.

[0:30:45.6] DC: Correct. Say, you're using a particular ORM — With the pace of development nowadays, they come out with a new version that isn't backwards compatible. It's completely separate. You can run the old version and new version side by side, for example. Maybe you want to migrate to that. With a vertical slice and kinda thinking about features, you can make that change within the unit of a command or a query, because that command or query has that dependency and that's where it lies, versus if you were in a layered approach, you would ultimately potentially have to rewrite your entire data access layer. You're still dealing with your technical concerns, but they're just now in a smaller unit. You can migrate, you can make decisions to a much smaller degree and little pockets.

[0:31:36.1] DR: Yeah. Yeah. Autonomy, I think, is basically the word that I hear you saying there, that you've got these things that can take a character of their own and you can make changes to small pieces rather than having to have the Big Bang, test everything kind of approach to getting a new application, and that's very appealing. I think that's a lot of why the world has started saying microservices all the time, right? Is that term applicable to what we're talking about here, or is that something completely different?

[0:32:03.1] DC: It could be in a sense that you're — Kind of what you're referring to is you're thinking of smaller units. You're getting in that vein, but this doesn't necessarily mean that you are deploying them independently. How you deploy them is a separate concern entirely. It's really just about at a code level of what is an individual feature. How that feature gets accessed via HTTP, or whether it's inline through some — Whatever. It's not so much about deployment as it is. Just what the unit is and how a particular feature is encapsulated.

[0:32:43.0] DR: I guess it's realizing some of the same benefits. I kind of like your approach there to say that microservices is more of a deployment concern and it may or may not be applicable to the way that you want to design and deploy your system, but the fundamental philosophy here of autonomy in your businesses cases, in your particular features, the vertical slices, and that's a great term. Maybe it would be useful to kind of try to mentally illustrate a little bit more. I know the audio medium is probably a little challenging for this. What do you mean by this distinction between horizontal and vertical?

[0:33:17.5] DC: The best way I could describe it, the way I've seen, is with a layered approach, thinking of a cake, like a birthday cake and having different layers that represent the technical concerns. Maybe at the very top, you have the top layer, which is — You know what I mean? Some UI, an then below that there may be some business logic layer that deals with, obviously, business logic validation, that type of stuff. Subsequently, it calls your data access layer and data access, then calls a layer below that, which double your data storage. Instead of dealing with layers that are application wide, take a slice out of that cake. Take a piece of the cake out and you'll still see the layers, but now you're dealing with, instead of the entire care, you're dealing with just a slice of it.

To circle back to what I said before, is that if you now just have these little slices, you can decide what one of those layers inside that cake is, and it may not all be the same.

[0:34:16.1] DR: Great. Testing is improved when you can look at a business case and all of those kinds of things rather than having to test these technical layers. Very appealing.

We've touched quite a bit on CQRS being broadly misunderstood. In that context, what are some of the common misconceptions about CQRS and some of the confusion? More broadly, I guess, is CQRS a golden hammer? Is it something that's applicable on all situations?

[0:34:45.6] DC: When we talk about the confusion, I guess, it's kind of what I stated before, is I think because it enables so much and once people get really excited about CQRS and see where it can take them and the things that allows you to do with event sourcing, which then leads to being able to have different models for read and write purposes and for querying. I think that's where the confusion lies is — Like I said. It's just a lot of the resources that you can find be mentioned more than just CQRS. The lines become a little bit blurry if you're not really sure what's going on there. I think that's the confusion.

You said it at the very beginning whether people are actually doing CQRS to begin with and they may not know it. I'd argue that's probably the case. The difference being is whether you're strict to it or not and whether you're just aware of it. Whether you're aware of that you have a particular actions in your system that are specifically from mutating state or specifically refer returning values.

I get the sense that there're probably a lot of people that are doing it, they don't realize they're doing it. The benefit, if they did, would be more so along the lines of where it can take them and how it can help various aspects of their application.

[SPONSOR MESSAGE]

[0:36:08.7] JM: When you are continuously deploying software, you need to know how your code changes affect user traffic around the world. Apica System helps companies with their end-user experience, focusing on availability and performance.

Test, monitor, and optimize your applications with Apica System. With Apica Zebra Tester, Apica Load Test, and Apica Synthetic, you can ensure that your apps an APIs work for all your users at any time around the world.

Apica Zebra Tester provides local load testing for individuals, small teams, and enterprise DevOps teams to get started quickly and scale load testing as your needs evolve. Apica Laod Test ensures that your app can serve traffic even under high load. Apica Synthetic sends traffic to your website and your API endpoints from more than 80 different countries, ensuring wide coverage.

Right now, you can go to softwareengineeringdaily.com/apica for a webinar about the real ROI of API testing. You can also find past webinars. Just how to optimize websites for fast load time.

Go to softwareengineeringdaily.com/apica to find the latest webinars on load testing and lots of other topics, and check out Apica System for testing, monitoring, and optimization. Thanks again to Apica for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[0:37:38.3] DR: Sure, yeah. That sounds great. A very pragmatic approach to how can this help and not necessarily getting caught in what does this term mean, but just how can you make your system better. I think that's really a good emphasis.

I wanted to ask you too about the organization of your code. You talked in some of your materials about not only this logical regrouping of your concept and being able to approach your business needs as a slice, these vertical slices you talked about. You also made some observations on the organization and layout of the code inside of your repository, contrasting the default directory structure of something like MVC framework that usually puts things into these technical buckets and changing the way that you organized your code. Can you tell the audience about that and what were some of the — First of all, what that entails, and then what were some motivations and some of the benefits of it?

[0:38:34.1] DC: Yeah. It took a while. Even when I was kind of mentally thinking about commands and queries as being features in kinda driving my code that way, it was interesting that I was so ingrained with organizing code that way, that I continued doing it still. I would still have controllers in a controllers folder. I would still have my commands in queries in separated folders. It wasn't until actually quite a bit afterwards where I mentioned the pain of having multiple files open in the layered approach. That kinda came back to me, "Why am I doing this? Why am I organizing code in a technical way? Why am I doing it that way when everything I'm thinking about is about use cases?"

I just decided I'm going to put code all in the same place, even to the extreme of I put, essentially, all in one file now where I'll have a controller and it's action within one file, that same file, put the class that represents my query, or command, and, subsequently, it's handler and maybe any events that it may publish, et cetera, and subsequent things that are all relating to that individual feature.

What that kind of enabled is the ability to just — If you are making a change or you're adding new functionality, you're essentially dealing within one file essentially, or there's obviously going to be some shared concerns through different features, but you're keeping this to a minimum, essentially. You're really only having to jump to one place. You're not jumping around projects and separate files.

That's ultimately what I do for the most part, create a folder called just the features folder and then, subsequently, I name them everything according to what the actual feature is and I just lump everything together.

[0:40:25.9] DR: Heresy, right? You're a heretic for doing this. It's a common procedure, that you're going to want to separate those things. I think a lot of people had a very visceral reaction to React JS by taking that same approach. Having your markup, your JavaScript to maybe even style in the same file that this is wrong, this is not a separation of concerns, but the Facebook developers said, "No. You know what? We want to separate business concerns rather than separating these technical concerns," and you've gone that same way.

I think you've illustrated very well the benefits of why you're keeping those concepts in your mind as you're working with it. You're grouping things together that you're working with together, and that's very appealing. Can you make the case for why it's not necessarily a bad thing to have some different types of technology in some of the same areas?

[0:41:16.2] DC: I guess because — When we're talking about business concerns, versus technical concerns, we're the ones that care about the technical concerns as developers. The end users don't. They care about features, and that's ultimately what I'm after is delivering value. I guess my mindset of switching more towards that and being — Not that you're not concerned or care about the technical ends of things. That's really not the case. It's more about delivering value and how quickly I can do that. For me, it was in our team being able to thinking about individual features.

If we're working on a given project, we can kind of work — If we're working independently on features, we can do so without stepping on each other's toes and we're just that mindset of that vertical slice and what that means rather than technical concerns that all need to be grouped together. It's actually interesting, especially when you start going to commands and queries, you start realizing that there isn't a lot of sharing necessarily going on. So that if I put everything in separate folders and separate projects, there's really no reuse there potentially. I might have — "Why not put them together? Why have to jump around between some folder that contains all the data access where one of those methods is only used in one place?"

[0:42:37.6] DR: Yeah. It might depend on your context, right? I think, early on, you talked about having a web application and a native Windows application and needing to share that code. In which case, putting controllers together with some of your business concerns; your command handlers and query handlers, that might not make as much sense because of the need to share those across the application. In a case where you're just dealing with a web application, then your structure may be more optimized depending on those contexts.

[0:43:05.2] DC: Yeah. I'll give you a little bit of an example of where I was going with that. I hope people can relate to this, and I'm not in an island here, is when using a repository, or a repository pattern, this is the way this would go. Somebody would create a new need to implement a new feature and it required data access. When we were using our layered approach, ultimately, what they would go do is create some new method in the repository that would —

[0:43:32.5] DR: Let's just pause for a moment. I think a lot of listeners may not be familiar with a repository pattern. It's another one of those things. You're really just talking about methods for accessing data when you say repository pattern here.

[0:43:43.8] DC: Yeah. Some class that represents methods for accessing data, whether that'd be — As well as updating, deleting data, whatever the case may be. Data access. Let's say that we had to implement a new feature where we needed to fetch out some product information from the database. What would likely happen in this situation is somebody would add a new method for that data access of a particular product in a specific way, if they need to retrieve it from the database, or data store, in a very specific way with maybe a certain set of criteria that never existed yet.

They would ultimately create a new method within that repository to fetch out that product. That method would only ever be used in a circumstance for that feature, because the next person that would ultimately be implementing another feature will probably end up creating another method that had different criteria, if it's some query for a report or whatever the case may be.

Where this ultimately would go down to is you'd end up either with an enormous amount of methods that did data access slightly differently with different criteria, or somebody would finally decide to, "Okay. I'm going to create this really generic method," that you could specify all the filters, and the paging, and the — Et cetera, into one, which would then was a really hard API to use. If you're going to create a method in a repository in a technical concern and you're only going to use it once, what benefit did you get?

[0:45:13.1] DR: You have cohesion in this class around a data access, but that's not necessarily the most useful form of cohesion, I guess is the crux of what you said there.

[0:45:22.7] DC: Exactly.

[0:45:23.4] DR: Yeah. All right. I think that makes the case pretty strongly. Perhaps a counterpoint is the discoverability of those data access methods. You know that you can go and look in that one class for the thing that you want to use. I think that's — There's obviously benefits and drawbacks to everything.

I wanted to go ahead and close here by talking about what you've learned and what the listeners can take away from this. You've clearly benefited from the approach that you've outlined here and have pitched as an improvement over the layer mentality. Why? What are the key takeaways and key benefits that you can leave us with that you have realized from the feature slices approach and having left behind the layered pile?

[0:46:01.9] DC: I guess it's just because it allows us to move faster, quicker in a sense of being able to deliver individual features that deliver value to our product and t our customers. When we get a request for something — When we're talking to end users, that's the language that they're speaking, is particular use cases. Having to convert that into layers, like in those concerns, just doesn't really ultimately feel natural to me anymore. It feels natural to me to implement a given feature without thinking about all the different layers that I have to go through and what benefit does that have necessarily.

If my layers are contained within a given unit of a command and query, to me, that gives me more agility later on when I need to make changes. Like I said, the pace of tech changes so often. One of the biggest benefits, I guess, to give a real world example was we were able to migrate from ASP NET Web API to the Nancy Framework with ease, because we weren't dependent on the framework and we were dealing with vertical slices. That top layer of what Web API provided for us was really thin.

I think it's just — Agility, I would say, is the biggest thing I take away from it.

[0:47:17.6] DR: Sure. That sounds great, and I think what we talked about too about a simplified dependency graph contributes to that agility and all of these things. A business focus is what those feature slices give you. It's all well said. Any parting words of wisdom before we go?

[0:47:32.7] DC: I would say just — If you're interested in CQRS, just take everything that you read with a grain of salt, I guess, a little bit, and take it for what Greg, in the quote that you read earlier, that it's really just a separation of reads and writes. It's just thinking once you start doing that, you can start setting other benefits to it, like I said, with feature slices. You can take it a lot of different ways, which a lot of the examples shows, such as different data models, using other patterns. It's just an enabler for a lot of things, but don't get all those other things mixed up for just the simplicity of separating reads and writes.

[0:48:08.7] DR: Sure. Sounds good. Greg Young, finding the material that he's written, and Udi Dahan, and, I think, obviously, what you have put out, the *Fat Controller CQRS Diet* stuff on codeopinion.com and on your Code Opinion YouTube channel are probably really good resources. Where else would you send somebody who wants to learn more?

[0:48:26.8] DC: I think — That's a good question. There's really no definitive source, I guess, for CQRS. I would say if you're searching for Greg, or Udi, Jimmy Bogard has a lot of great posts, blog posts on most techies. He is the creator of AutoMapper and .NET, as well as Mediator, which is a library that I use that's perfect for the use cases that I use. Kind of what I've been describing.

He has a lot of good content as well. I don't know if there's any really a definitive source, you kinda have to scour a little bit for content.

[0:48:58.9] DR: Sure. Okay, well thank you Derek for joining me on Software Engineering Daily and sharing your experiences and insights.

[0:49:04.5] DC: Thanks a lot Dave.

[END OF INTERVIEW]

[0:49:09.4] JM: Thanks to Symphono for sponsoring Software Engineering Daily. Symphono is a custom engineering shop where senior engineers tackle big tech challenges while learning from each other. Check it out at symphono.com/sedaily. That's symphono.com/sedaily. Thanks again Symphono.

[END]