DZone

CQRS: Understanding From First Principles

by Justin Boyer · Mar. 08, 18 · Microservices Zone · Tutorial

There seems to be no end to the choices you have for architecture when building an application. You don't want to fall victim to cargo cult programming, so you need to truly understand the options available. Today, we'll focus on one option, called CQRS.

CQRS leads to a clean architecture that's easy to maintain. Let's take a look at the underlying principles of CQRS. This will help you understand what the benefits are and whether you want to use it in your applications.

CQRS Principle: Separate Commands From Queries

CQRS stands for command query responsibility segregation. What does that mean? In a nutshell, it's an architectural pattern that separates the application layer of your software into a command stack and a query stack. It was first described in the book *Object-Oriented Software Construction* by Bertrand Meyer.

Meyer stated that there are two types of methods in object-oriented software: **commands** and **queries**. Commands are operations that change the application state and return no data. These are the methods that have side effects within the application. Queries are operations that return data but don't change application state.

In concrete terms for .NET developers, your application will have two "stacks," or namespaces. One namespace will be *YourApplication.CommandStack* and the other *YourApplication.QueryStack*. This structure gives you a clear picture of what pieces of code change the state of the application. If you like to think of software in terms of domain models, this structure has two of them-a query model and a command model. These are different models that have their own operations and representations of data. These representations can then be optimized for their specific purpose. (More on that later.)

Changes to data tend to cause the most bugs. Having a clear understanding of which parts of the application change data and which do not will help with maintainability and debugging.

CQRS Principle: Build SOLID Software

SOLID is a set of object-oriented software principles that are expressed quite well in the CQRS architecture.

CQRS states that commands need to be separate from queries. Methods that have surprise side effects are not welcome. A method does one thing and one thing only, either returning data or changing it. Thus, CQRS is a great expression of the single responsibility principle.

Without this separation of commands and queries, you may end up with domain models that are full of state, commands, and queries that make them much harder to reason about and maintain over time.

The interface segregation principle states that many client-specific interfaces are better than one general-purpose interface. CQRS forces you to define clear interfaces between the parts of the system. The client talks to either a command interface or a query interface. There's a clear distinction and choice for the client. Therefore, the client knows exactly what to expect.

CQRS Principle: Optimize for Your Specific Situation

One of the great parts of using CQRS in your application is the choices it enables. The flexibility gained by separating commands and queries allows for specific optimization of different parts of the application based on your needs. Let's take a look at three examples of this.

Basic CQRS

First, following basic CQRS design allows for optimization of reads and writes. The design of the query stack represents the most efficient way to read data. The command stack only needs to worry about how to write data. Basic CQRS typically shares one database between both the command and query models.

CQRS with Two Databases

However, you can take optimization a step further by creating your application with two databases: a read database and a write database. Use a relational data store that's optimized for writing on the command side. Use a denormalized or NoSQL data structure on the query side to make reads from the read database as fast as possible. Since most applications read data much more often than they write data, this separation and subsequent optimization makes a ton of sense in many applications. If your applications require high performance at all times, this variation of CQRS will allow you to highly optimize the command and query models for optimum performance.

Event Sourcing CQRS

Event sourcing CQRS involves storing all changes to an object as a series of events in an event store. The write side replays the events of a particular object to derive the current state of the object. The read side database holds the current state of all objects for fast reading of data. Event sourcing CQRS is the most complex option.

Event sourcing CQRS provides several strong benefits, despite the added cost. First, the event store acts as an audit trail for the entire system. Heavily regulated industries will find this feature quite useful. Second, you can always recreate the state of any object by replaying the event store. If your production read database is destroyed somehow, use the event store to recover in a fraction of the time. Third, the event store can feed its data into multiple read databases. The query code chooses which data store to use for any given situation. For example, you could populate graph databases, OLAP cubes, search indexes, and any other store you need.

Do You Need CQRS in Your Application?

Even though CQRS sounds impressive, caution is in order. CQRS is a software design pattern. It's great to have in the toolbox, but it may not be applicable to every situation. Don't make the mistake of applying this one pattern to every application you build. A skilled software engineer or architect will carefully determine where this pattern fits. Let's take a look at some guidelines of where you can use this pattern.

CQRS fits best in domain-driven design (DDD) architectures. DDD focuses on building rich domain models to capture complex business logic. Use it within certain bounded contexts to help simplify the models. Having a command model and query model can help to simplify each model so they don't grow too large. On the other hand, be careful not to introduce it into contexts where the command and query models end up sharing similar functionality. In these cases, it will add complexity to try to separate the models. Just use one.

Using CQRS with event sourcing (CQRS/ES) is a natural fit for applications that depend greatly on events. For instance, if you own a downstream service that responds to events from a UI or other system, you can use CQRS/ES to keep track of all events and thus reconstruct the domain at any point in time by reading the events. The API exposed to the users of the service will be simple and clean.

Use Your Tools Skillfully

You can find a good review of CQRS and its benefits and challenges at Martin Fowler's blog. In a nutshell, remember that it brings great benefits:

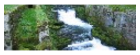
- Smaller models that handle either commands or queries.
- Simpler API for the consumers of the service
- The ability to optimize the read and write sides for your specific needs (domain models and data sources)

However, realize that CQRS also adds complexity. More advanced applications require the use of multiple data stores. Those data stores and the domain models at runtime have to be kept in sync. Even skilled teams can have trouble if it's used in the wrong applications. CQRS is a scalpel, not a machete. Use it skillfully, carefully, and in the right contexts, and you'll greatly improve the maintainability and performance of your applications.

Like This Article? Read More From DZone

RavenDB Conference Videos:

The Good of Event Sourcing:


Implementing CQRS and Event Sourcing

Projections

Software Architecture: The 5 Patterns You Need to Know

Free DZone Refcard
Microservices in Java

Topics: CQRS , MICROSERVICES , EVENT SOURCING , API , DATABASE

Published at DZone with permission of Justin Boyer . See the original article here.
Opinions expressed by DZone contributors are their own.

licroservices Partner Resources

odern APM in a Containerized World
stana
|
odernizing Application Architectures with Microservices and APIs Virtual Summit Series
\ Technologies
|
eploy Pre-Built Sample Microservices OR Create Simple Microservices From Scratch.
oudentity
|
ie Six Pillars of AI-Powered APM for Containerized Microservices
stana
|

Micronaut Mastery: Using Reactor Mono and Flux

by Hubert Klein Ikkink MVB · Aug 17, 18 · Microservices Zone · Tutorial

Containerized Microservices require new monitoring. See why a new APM approach is needed to even see containerized applications.

Micronaut is reactive by nature and uses RxJava2 as the implementation for the Reactive Streams API by default. RxJava2 is on the compile classpath by default, but we can easily use Project Reactor as the implementation of the Reactive Streams API. This allows us to use the Reactor types `Mono` and `Flux`. These types are also used by Spring's Webflux framework and make a transition from Webflux to Micronaut very easy.

How do we use Project Reactor in our Micronaut application? We only have to add the dependency the Project Reactor core library to our project. In the following example, we add it to our `build.gradle` file:

```
// File: build.gradle
// File: build.gradle
...
dependencies {
    ...
    // The version of Reactor is resolved
    // via the BOM of Micronaut, so we know
    // the version is valid for Micronaut.
    compile 'io.projectreactor:reactor-core'
    ...
}
...
```

Now we can use `Mono` and `Flux` as return types for methods. If we use them in our controller methods, Micronaut will make sure the code is handled on the Netty event loop. This means we must handle blocking calls (like accessing a database using JDBC) with care and make sure a blocking call invoked from the controller methods is handled on a different thread.

In the following example, we have a simple controller. Some of the methods use a repository implementation with code that accesses databases using JDBC. The methods of the repository implementation are not reactive, therefore we must use `Mono.fromCallable` with Reactor's elastic scheduler to make sure the code is called on separate threads and will not block our Netty event loop.

```
package mrhaki;

import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;
import reactor.core.scheduler.Schedulers;

import java.util.concurrent.Callable;

@Controller("/languages")
public class LanguagesController {

    // Repository reads data from database
    // using JDBC and uses simple return types.
    private final LanguagesRepository repository;

    public LanguagesController(final LanguagesRepository repository) {
        this.repository = repository;
    }

    @Get("/{name}")
    public Mono<Language> findByName(final String name) {
        return blockingGet(() -> repository.findByName(name));
        return blockingGet(() -> repository.findByName(name));
    }

    @Get("/")
    public Flux<Language> findAll() {
        return blockingGet(() -> repository.findAll()).flatMapMany(Flux::fromIterable);
    }

    // Run callable code on other thread pool than Netty event loop,
    // so blocking call will not block the event loop.
    private <T> Mono<T> blockingGet(final Callable<T> callable) {
        return Mono.fromCallable(callable)
                   .subscribeOn(Schedulers.elastic());
    }
}
```

The repository interface looks like this:

```
package mrhaki;

interface LanguagesRepository {
    List<Language> findAll();
    Language findByName(String name);
}
```

Let's write a Spock specification to test if our controller works correctly:

```
package mrhaki

import io.micronaut.context.ApplicationContext
import io.micronaut.core.type.Argument
import io.micronaut.http.HttpRequest
import io.micronaut.http.HttpStatus
import io.micronaut.http.client.HttpClient
import io.micronaut.http.client.exceptions.HttpClientResponseException
import io.micronaut.runtime.server.EmbeddedServer
import spock.lang.AutoCleanup
import spock.lang.Shared
import spock.lang.Specification

class LanguagesControllerSpec extends Specification {

    @AutoCleanup
    @Shared
    private static EmbeddedServer server = ApplicationContext.run(EmbeddedServer)

    @AutoCleanup
    @Shared
    private static HttpClient client = server.applicationContext.createBean(HttpClient, serve

    void '/languages should return all languages'() {
        given:
        final request = HttpRequest.GET('/languages')

        when:
        final response = client.toBlocking().exchange(request, Argument.of(List, Language))

        then:
        response.status() == HttpStatus.OK

        and:
        response.body()*.name == ['Java', 'Groovy', 'Kotlin']

        and:
        response.body().every { language -> language.platform == 'JVM' }
    }

    void '/languages/groovy should find language Groovy'() {
        given:
        final request = HttpRequest.GET('/languages/Groovy')

        when:
        final response = client.toBlocking().exchange(request, Language)

        then:
        response.status() == HttpStatus.OK

        and:
        response.body() == new Language('Groovy', 'JVM')
    }

    void '/languages/dotnet should return 404'() {
        given:
        final request = HttpRequest.GET('/languages/dotnet')

        when:
        client.toBlocking().exchange(request)

        then:
        HttpClientResponseException notFoundException = thrown(HttpClientResponseException)

        notFoundException.status == HttpStatus.NOT_FOUND
    }

}
```

Written with Micronaut 1.0.0.M4.

Like This Article? Read More From DZone

A Light RX API for the JVM [Video]

Reactor 3.0, a JVM Foundation for Java 8 and Reactive Streams [Video]

Accessing Data - The Reactive Way (Part 4 of Introduction to Vert.x)

Free DZone Refcard
Microservices in Java

Topics: JAVA, MICROSERVICES, MICRONAUT, PROJECT REACTOR, MONO, FLUX, REACTIVE STREAMS, API, CODE

Published at DZone with permission of Hubert Klein Ikkink , DZone MVB. See the original article here.
Opinions expressed by DZone contributors are their own.