CQRS stands for Command-Query Responsibility Segregation and was introduced by Greg Young in 2010. It is based on the command-query separation principle coined by Bertrand Meyer, which states that every method should either be a command that performs an action, or a query that returns data to the caller, but not both. It is important to note that not all methods can follow the command-query separation principle, and some may need to have a side effect and return something.

CQRS is a method-based system that takes the CQS principle and extends it to a higher level. It encourages users to untangle a single, unified domain model and create two models: one for handling commands or writes, and the other for handling queries or reads. The FileExists method is a query that returns a Boolean value and doesn't mutate the file, while the WriteToFile method is a command that changes the file and its return type is void. The downside of this method is that it no longer follows the CQS principle. CQRS also encourages users to make the CQS principle their default choice, and depart from it only in exceptional cases. CQRS is a pattern that splits a single model into two; one for reads and one for writes.

It provides benefits such as scalability, performance, and simplicity. By introducing two models instead of just one, complexity can be offloaded from the code base, allowing users to handle two different use cases with the same code.

CQRS is a single responsibility principle that optimizes decisions for different situations, allowing for different levels of consistency, database normal forms, and even different databases for command and query sides.

It is used in real-world projects such as Entity Framework, NHibernate, and raw SQL with plain ADO. NET.

CQRS is a pattern that extends CQS to the architectural level, allowing for two models to be created: one for handles commands and one for handling queries. It brings benefits such as scalability, performance, and simplicity.

Commands in CQS and CQRS are two different concepts. A command is a method that modifies the application's state and doesn't return any data to the client apart from the operation confirmation.

A command handler is a regular ASP. NET controller with just a single method that mutates state and doesn't return anything aside from the operation confirmation. Commands, queries, and events are three categories of messages in an application. Commands are telling the application to do something, queries are asking it about something, and events are informational messages.

Naming guidelines are associated with all three types of messages, with commands always being in the imperative tense, queries usually starting with the word Get, and events always being in the past tense.

The distinction between command and event is important when using domain events. Commands, queries and events should be named in either imperative or past tense to make it clear which message is which.

It is important to use the ubiquitous language and ensure that commands are task-oriented. Postfixes should be used when naming commands, queries and events, but it is optional to leave them out.

The onion architecture shows elements of a typical application, with the core representing the domain model and parts touching the external world.

Commands, queries, and events are all part of the core domain model, which is located in the center of the onion layer. The core domain should be isolated from the external world and self-sufficient. Commands trigger a reaction in the domain model, while events are the result of that reaction.

The core domain has two models: push and pull. Commands and queries should not be implemented in those layers due to domain model isolation. However, handles should refer to the external world or outer layers of the onion. The handlers should go to the Application Services layer and move their code from the controller to the handler.

Event Sourcing and CQRS are two important concepts to consider when refactoring applications to follow the CQRS pattern.

CQRS can be applied on its own, but Event Sourcing brings a lot of complexity to the table. Event Sourcing is best used for systems that need to track domain events, such as finance tech applications. Evolutionary Design is an important concept to consider when implementing Event Sourcing.

CQRS is a design pattern that should be bound to a specific bounded context or microservice. It should be implemented with the task-based API alone, and the separation down to the domain model level. Commands and queries from command and query handlers can be used, but the long answer depends on the situation.

Event Sourcing is a system that uses a combination of the client, the Write model, and the Read database to retrieve the current state of an application. The client sees the current state of the application and produces a command, while the Write model receives the command and produces events. If there is no reliable way to request the current state, a query from a command handler is needed. However, the Read and Write databases are not immediately consistent, so unique constraint validations might not be done due to this.

Event Sourcing and CQRS are important concepts to understand when using query handlers. Commands should not be unidirectional and should instead poll an external source for results. One-way commands should return acknowledgments or locators if the operation is not inherently asynchronous and doesn't take much time to complete. It's fine to return an id of the newly created entity if it's not inherently asynchronous. CQRS and the Specification Pattern are two different principles that come into play when implementing Event Sourcing. CQRS and the Specification pattern are two Domain-Driven Design patterns that are incompatible. The Specification pattern allows us to encapsulate a piece of domain knowledge into a single place and then reuse it in three scenarios; data retrieval, user input validation, and creation of a new object.

On the other hand, CQRS proposes the separation of the two, dealing with those concerns separately. This is an example of the contradiction between the DRY principle, which stands for Don't Repeat Yourself, and the principle of loose coupling, which stands for Don't Repeat Yourself. Loose coupling wins in the vast majority of cases, except for the simplest ones. CQRS is a pattern that prioritizes simplicity, performance, and scalability. CQRS is a pattern that involves separating API endpoints, extracting explicit commands and queries, and creating a separate database for reads.

Command and Query Responsibility Segregation (CQRS) and Event Sourcing, when leveraged within the context of Domain-Driven Design (DDD) in cloud-native services, offer a powerful approach to building scalable and resilient systems.

CQRS separates the read and write operations into distinct responsibilities, enabling independent optimization and scaling of each. By adopting CQRS, developers can design cloud-native services that efficiently handle high query loads while ensuring data consistency through strict command processing. Event Sourcing complements CQRS by capturing all changes to the system's state as a sequence of events. These events serve as a single source of truth, providing an audit trail and enabling easy system reconstruction.

When combined with DDD, CQRS and Event Sourcing allow developers to model complex business domains effectively, aligning technical implementations with the business's language and logic.

Cloud-native services designed using this approach exhibit high scalability, fault tolerance, and are well-suited for event-driven architectures. Furthermore, with the inherent support for asynchronous communication and eventual consistency, these services can seamlessly integrate into distributed and microservices-based cloud environments, facilitating agility and adaptability in modern application development.