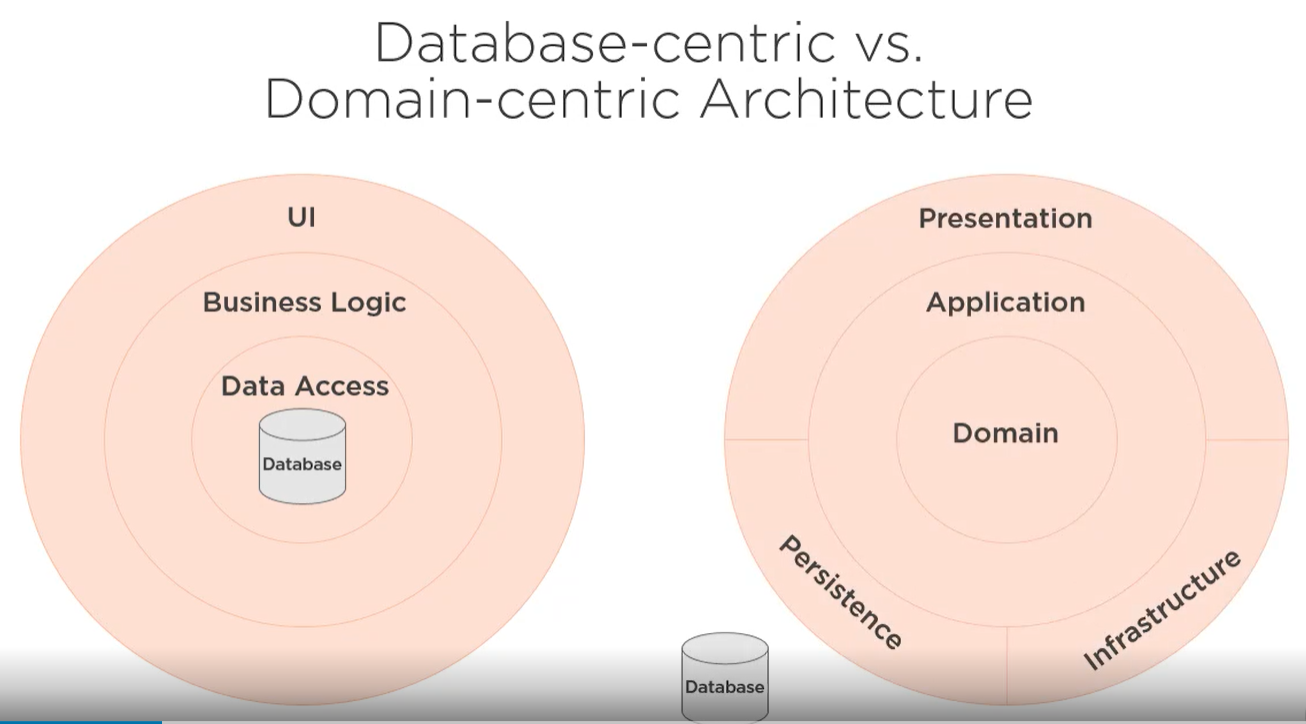
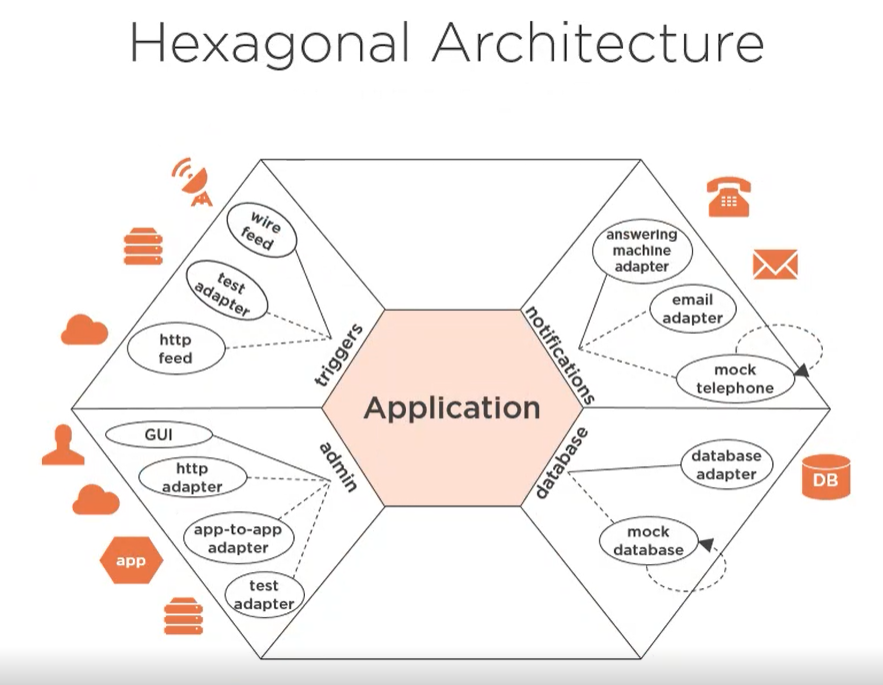
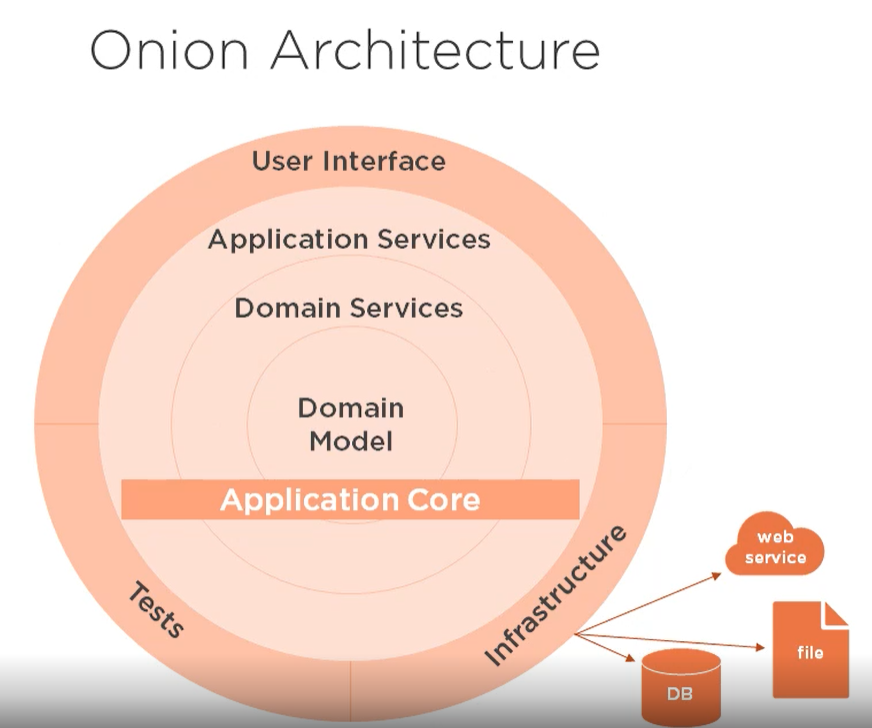
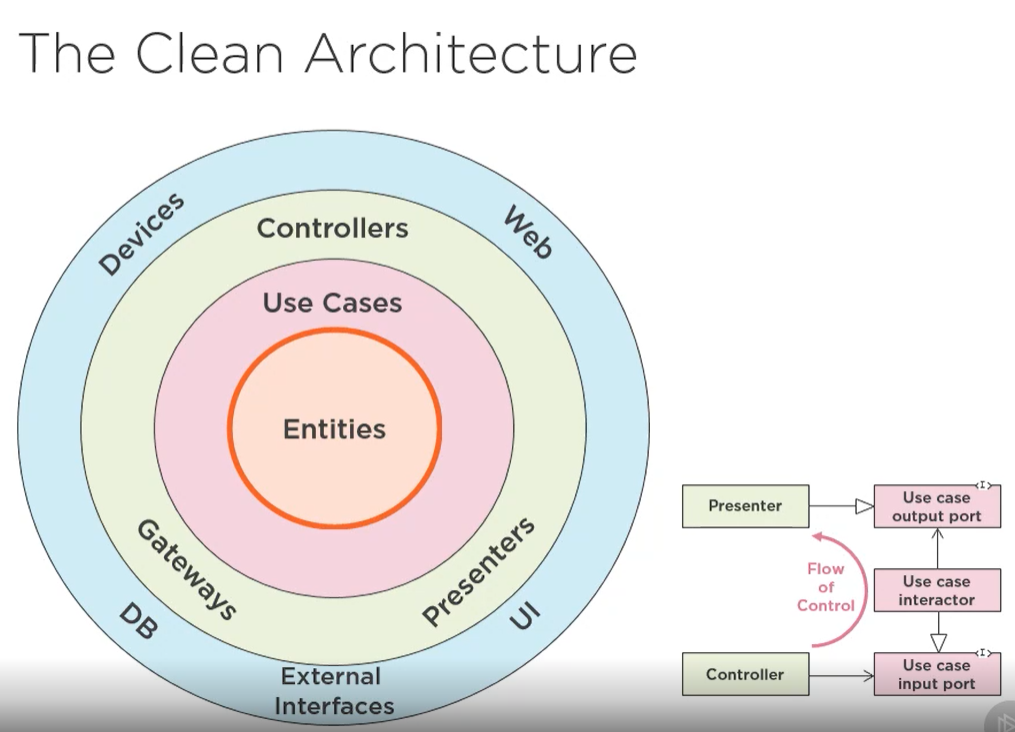
* <https://github.com/matthewrenze/clean-architecture-demo>
* **What is Clean Architecture?**
* Architecture that is designed for the inhabitants of the architecture (users, developers that work on it, developers that maintains it… etc.) not for the architect or the machine.
* Prevent premature optimization (the root of all evil)
* **Why invest in clean Architecture?**
* Minimize cost
* Maximize value
* Focus on the essential
* Build only what is necessary when it is necessary
* Optimize for maintainability



* **The first concern of the architect is to make sure that the house is usable, it is not to ensure that the house is made out of brick (uncle bob)**
* **Essential vs Detail:**
* Domain is essential
* Use cases are essential
* Presentation is a detail
* Persistence is a detail
* **Types of Domain-Centric Architect:**



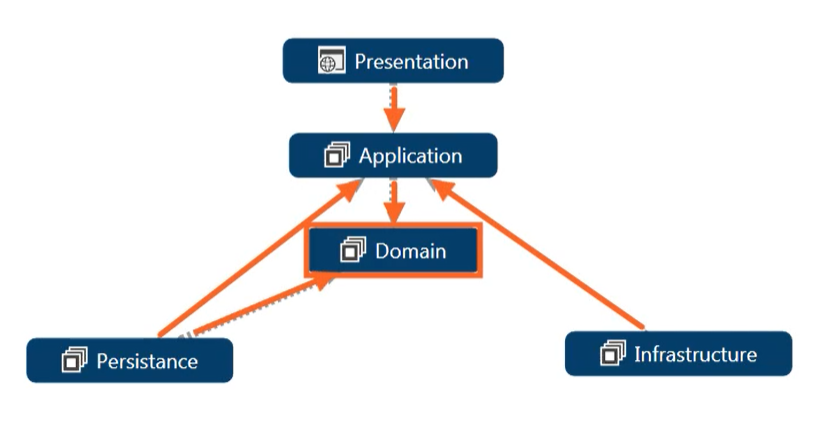




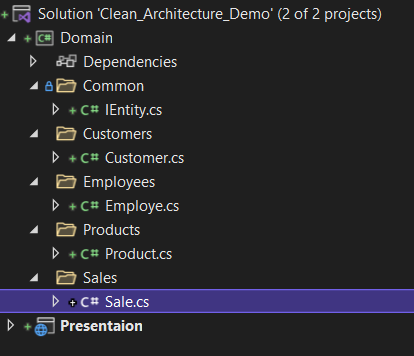
* **Pros:**
* Focus on domain
* Less coupling
* Allows for DDD
* **Cons:**
* Mindset change is difficult
* Requires more thought
* Initial higher cost

1. **Domain Layer:**

* Domain is at the center of the architecture and all dependencies point either directly or indirectly towards the domain.
* Domain project contains our domain model which is an abstract representation of the business problem being solved.

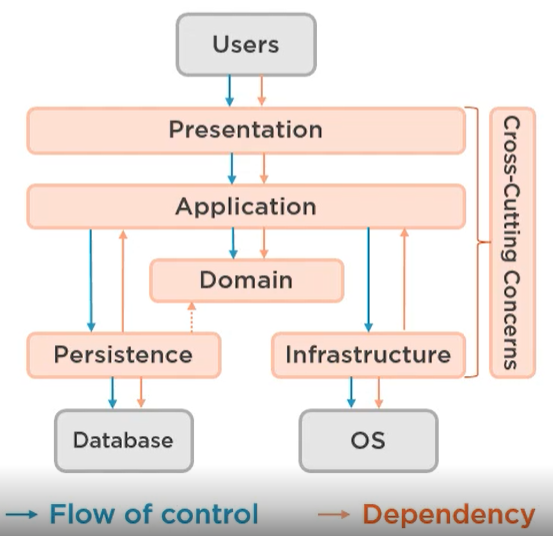


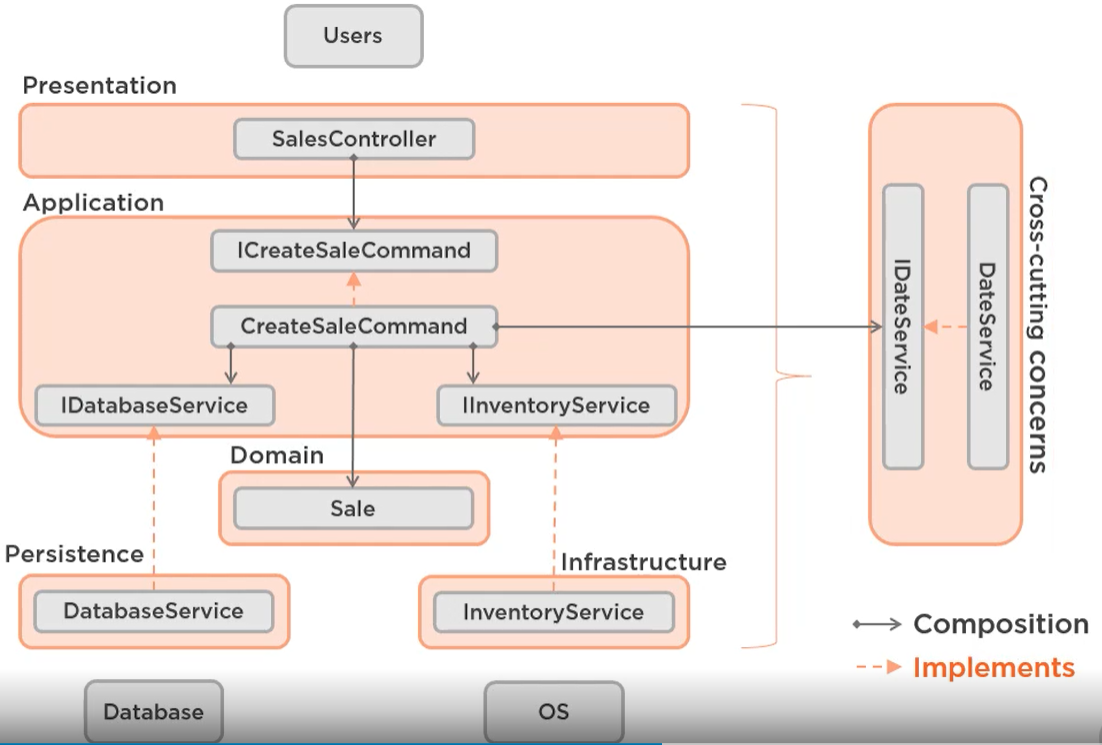
* We organize out folders, classes and namespaces via a concept we refer to as functional cohesion, we organizing things by the key entities in out domain (**Aggregate Root Entities**).
* Each folder is named after the root entity.
* Folder contains any value objects or domain services that these entities depend upon.



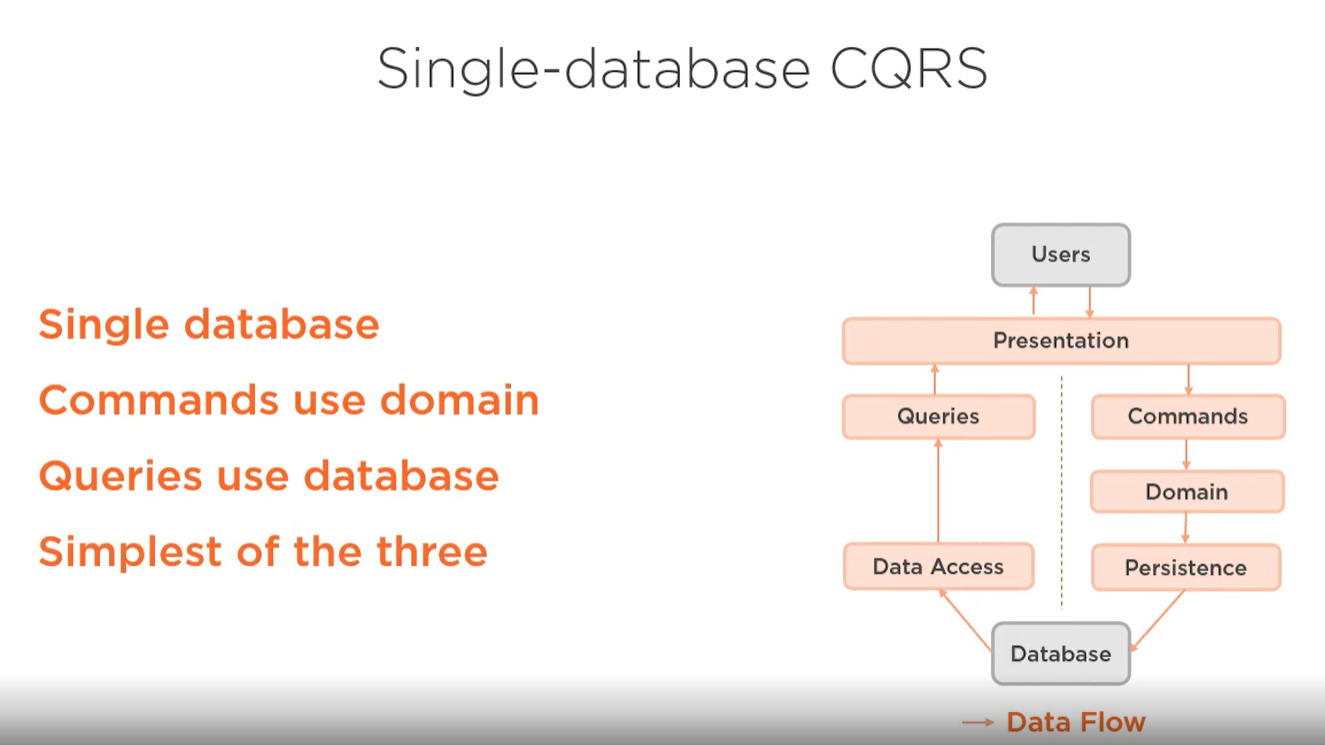
1. **Application Layer:**

* Implements the use cases of the application as executable code and abstractions (ex: customer search for a product => adds it to their cart => pays with credit cart).
* High level application logic
* Knows about the domain
* No knowledge of other layers
* Contains interfaces for its dependencies that these respective outer layers (persistent, infrastructure) then implement. Then we use an IOC framework that is an inversion of control framework and dependency injection to wire up all the interfaces and their implementation at run time.
* **Pros:**
* Focus on use cases
* Easy to understand
* Follows dependency inversion principle
* **Cons:**
* Additional layer cost
* Requires extra thought

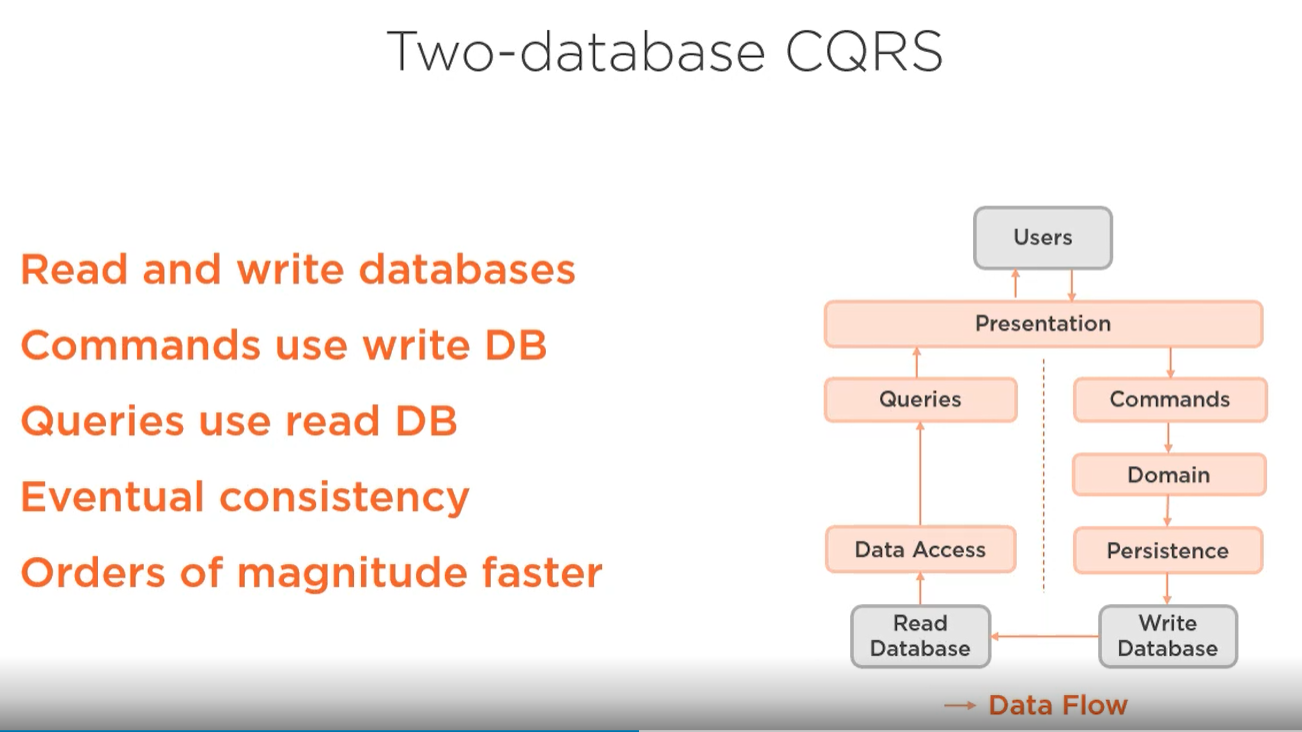




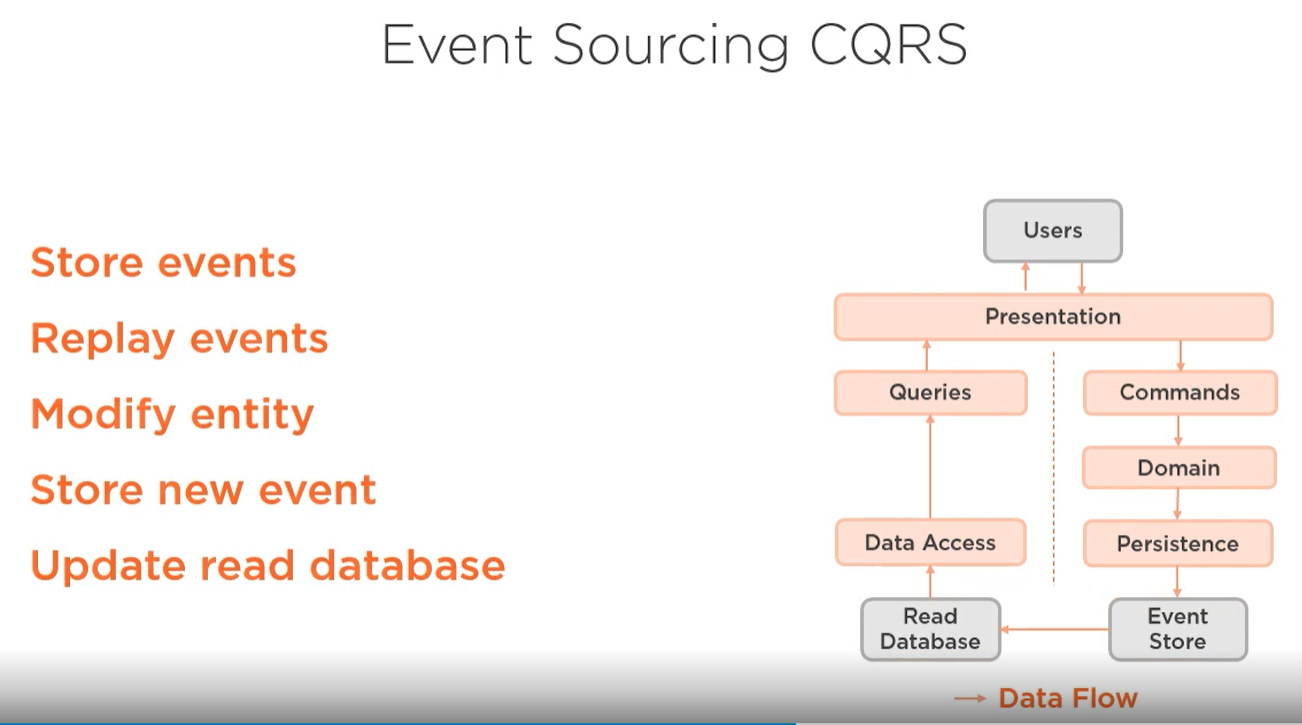
* **CQRS: Command Query Separation**
* **Command:** Does something, should modify state, should not return a value
* **Query:** Answers a question, should not modify state, should return a value
* **Exceptions:**
* Pop a stack: Remove item (command), Return top item (query)
* Create new record: Create record (command), Return new id (query)
* **Types:**
* **1- Single database:**

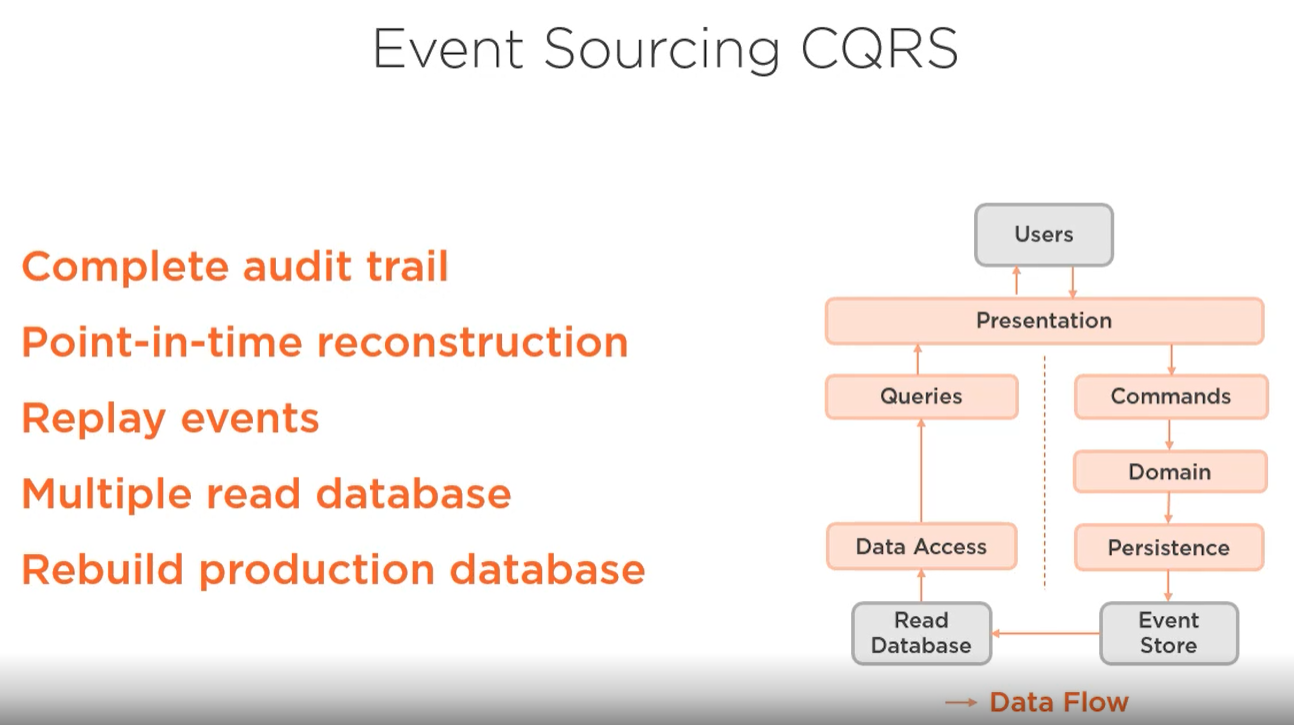


* **2- Two database:**



* **3- Event Sourcing CQRS:**



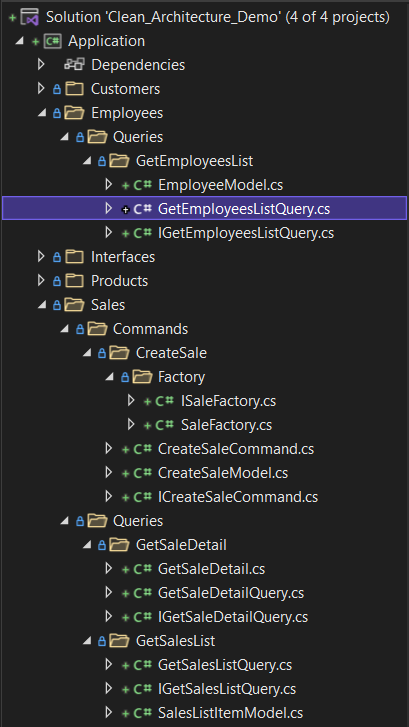


* **CQRS Pros:**

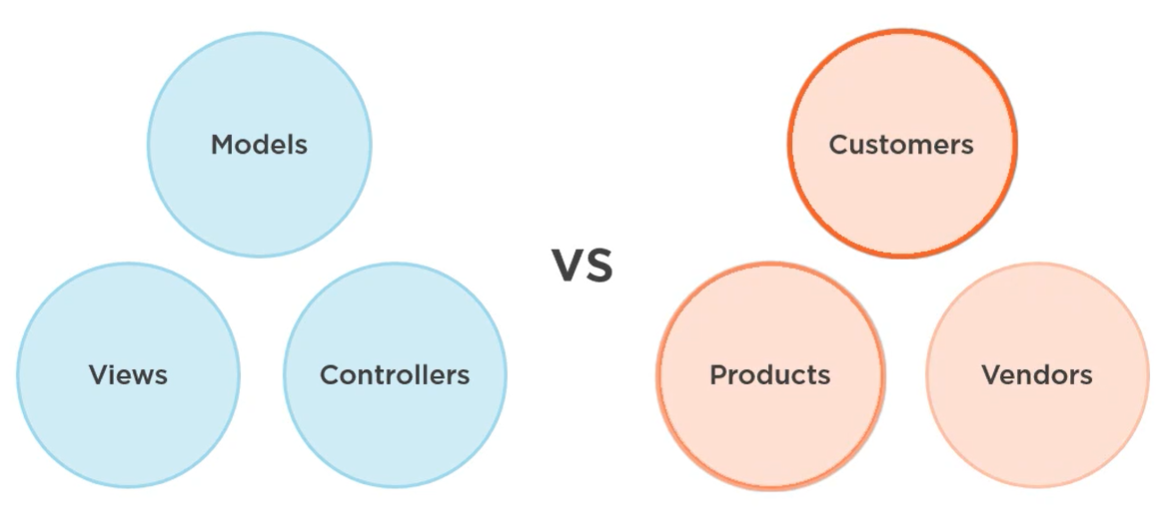
1. More efficient design
2. Optimized performance
3. Event sourcing benefits

* **CQRS Cons:**

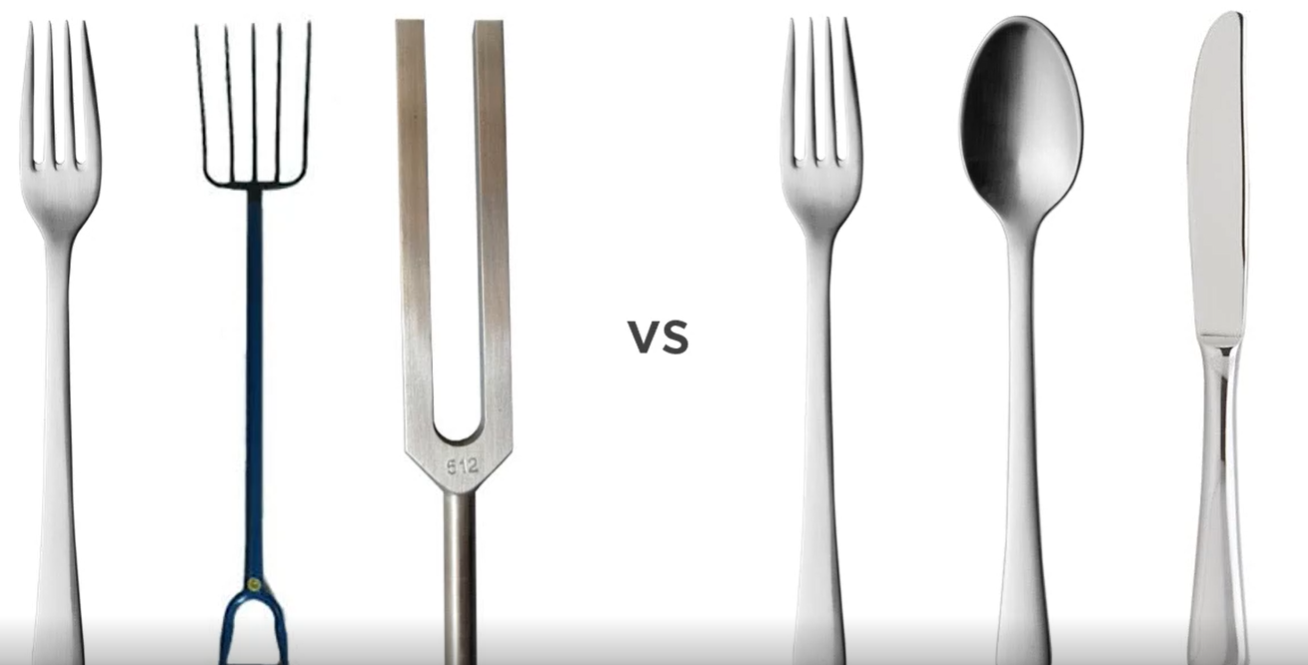
1. Inconsistent across stack
2. More complex
3. Event sourcing costs



* **Functional Organization:**
* The architecture should scream the intent of the system. (uncle bob)
* We can organize our applications folder structure and namespaces according to the components that are used to build the software (model – view - controller) => **bad way**.
* We can organize our applications folder structure and namespaces according to the use cases of the system (customer – product - vendor) => **good way**.
* **Functional cohesion:** that is organizing by use cases (items that are used together live together). (better models the way we maintain, navigate and reason about software)
* **Categorical cohesion:**  that is organizing by types.
* It’s much easier to determine the intent of the software on the right than the left.







* **Acceptance Tests:**

