
synyx
code
with
attitude

# Trends in modern software architecture

# ...and how to avoid them

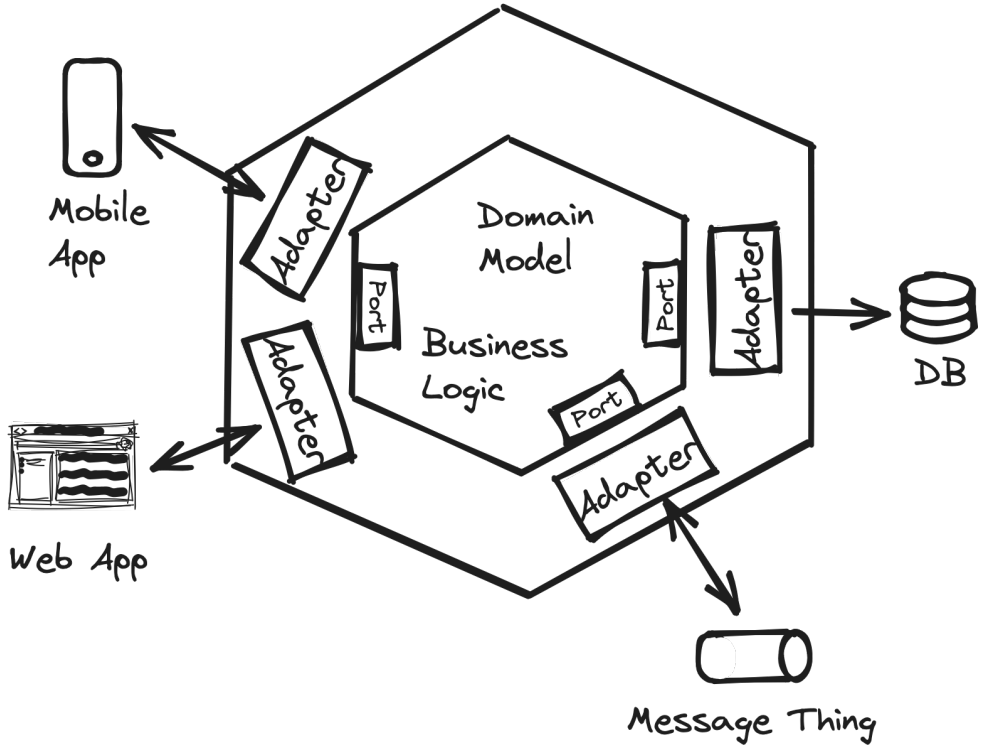
# OVERVIEW

- Hexagonal Architecture
- Reactive Programming Models
- CQRS
- Event Sourcing
- Summary

# HEXAGONAL ARCHITECTURE

# WHAT IS IT?


Layered




Hexagonal

# PROMISE

- Protect your core domain model against leaking
- Outer layers can never force changes to the core

# PITFALLS

You end up with lots of interfaces and duplications that constantly change during development, creating a hard-to-maintain codebase

# ALTERNATIVES

- What do you want to achieve?
  - Protect core domain → Why ports & adapters?
  - Enforce architectural rules → ArchUnit
- Apply abstractions where they make sense
- Apply common sense everywhere else

# CONCLUSION

- Preventing your core domain model from leaking to the web is common sense
- Dozens of interfaces with exactly one implementation are ridiculous

# REACTIVE PROGRAMMING MODELS

# WHAT IS IT?

- Classic, blocking programming models
  - → Thread per Request-Model
  - → Methods return results
- Reactive programming models
  - → Event Loop Model
  - → Callback functions or Futures

# PROMISE

- Threads not blocked by downstream work
- Very responsive applications
- Tens of thousands of requests in seconds

# PITFALLS

- Very different programming model
- Very steep learning curve
- Blocking downstream APIs

# ALTERNATIVES

- Consider expected load scenarios before making design decisions
- In most cases, horizontal scaling might prove to be more cost-efficient
- In Java, Virtual Threads might make reactive APIs obsolete

# CONCLUSION

From the Spring WebFlux documentation:

*“ We expect that, for a wide range of applications, the shift is unnecessary. ”*

# CQRS

# WHAT IS IT?

- Command-query responsibilty segregation
- Separate write- & read-models

# PROMISE

- APIs with asymmetric read-write load
- APIs where queries require computed outputs
- Message-driven APIs

...all benefit from separate models

# PITFALLS

Maintaining separate models, along with controller classes, business logic, persistence layers etc, leads to far more complex software projects

Separate models implicitly lead to eventual consistency (stale reads)

# ALTERNATIVES

Always consider using a plain CRUD API first

Start CRUD API, evolve a separate write API over time, keeping the CRUD API for queries

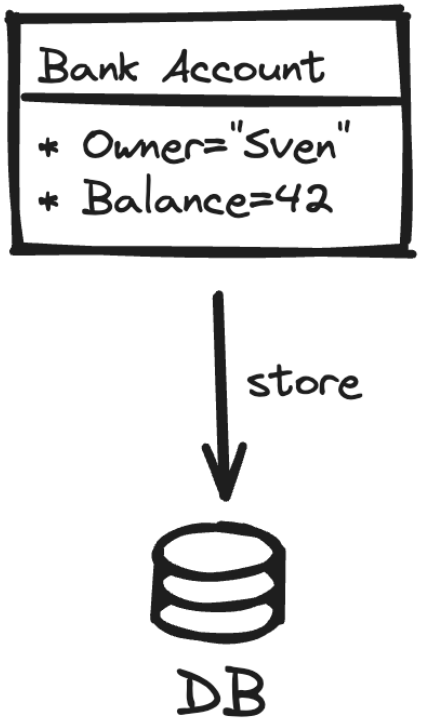
# CONCLUSION

From Greg Young's blog:

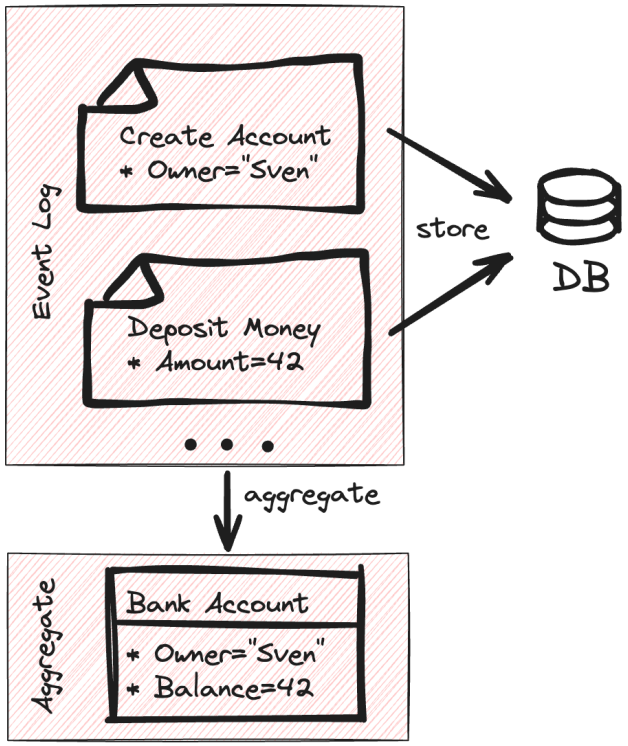
*CQRS is not a silver bullet*
*CQRS is not a top level architecture*
*CQRS is not new*
*CQRS is not shiny*
*CQRS will not make your jump shot any better*
*[...]*
*CQRS can open many doors.*

# EVENT SOURCING

# WHAT IS IT?




Object Persistence




Event Sourcing

# PROMISE

Focus on change instead of state makes building reactive systems easier

All domain objects implicitly have a history

# PITFALLS

Very complex persistence pattern

Usually implemented along with CQRS, leading to expotentially more complex software systems

When used with CQRS, the aggregates aren't allowed to answer queries

Very steep learning curve

# ALTERNATIVES

Event-driven systems can easily be build using simple persistence patterns

When a history of domain objects is required (eg, for auditing), consider using Envers

# CONCLUSION

Unless you're building a real-time stock trading system, don't do it

# THE QUEST FOR SIMPLICITY

# THE QUEST FOR SIMPLICITY

Do the simplest thing that could possibly work

Non-functional requirements matter!

Consider the available skill set

Mind Conways law

Highly aligned, loosely coupled

Make sure technical and business goals align

# THANK YOU!

Slides

LinkedIn