REIMAGINING MONOLITHS WITH DDD

Using domain-driven design to reimagine monolithic applications in a world of microservices

Eric Murphy and Aleš Nosek
Red Hat Consulting Architects
May 8, 2019

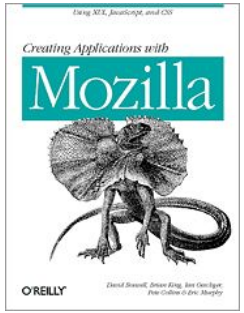
INTRODUCTIONS

Eric Murphy

RED HAT APP DEV CENTER OF EXCELLENCE

- Architect
- App Dev Center of Excellence

Fun Fact: Co-authored *Creating Applications with Mozilla*, O'Reilly, 2002



Aleš Nosek

redhat. CERTIFIED ARCHITECT

- Red Hat Certified Architect
- Consulting, West Region

Fun Fact: Worked at SUSE on open source projects, then worked on proprietary products. **Felt guilty**, so came to work on open source at Red Hat!

TODAY’S DISCUSSION

- Not a full introduction to Domain Driven Design (DDD)
- Practical focus on DDD — Keep it simple, stupid (KISS principle)
- Technical, rather than high level
  - Show me the code!
  - Special focus on using Quarkus/Vert.x, but transferable to other technologies
- **Consider using DDD with monolithic architecture as the preferred choice for brand new applications**
  - Take today’s advice back to your team

2012:
DREAD FOR

SOA AND MONOLITHS

2012:
HYPE FOR

MICROSERVICES!

2018:
HYPE FOR

SERVERLESS (FUNCTIONS)!

2012-2018:
DISDAIN FOR

MONOLITHS

Today:
RETHINKING

MONOLITHS

?

WHY RETHINK MONOLITHS?

Not all apps should be microservices or functions

WHY RETHINK MONOLITHS?

Monoliths can easily be split into microservices or functions if designed correctly

WHY RETHINK MONOLITHS?

Domain-Driven Design (DDD) can guide you in building monoliths that are decomposable

WHY RETHINK MONOLITHS?

You don't have to be "locked in" to a monolithic architecture when building a monolithic application

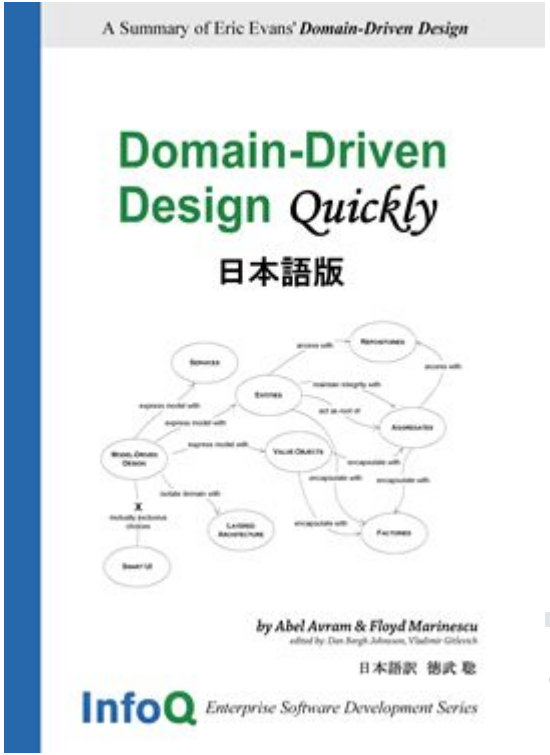
DOMAIN-DRIVEN DESIGN (DDD)

DOMAIN-DRIVEN DESIGN (DDD)

- **Merge your business domain with software modeling**
- Special focus on:
  - Capturing how users interact with the software (via event storming activities)
  - Identifying clusters of domain objects in the system (aggregates)
  - Finding boundaries within business domain (bounded context)
- Check https://en.wikipedia.org/wiki/Domain-driven_design for a great summary!
- **Warning:** DDD content is often very academic and dry, but invaluable to modeling a domain and effectively designing modern software

RECOMMENDED READING

- Domain-Driven Design Quickly
- Summary of Eric Evan's definitive book
- Online version available for free (Google it)



EVENT STORMING

- Facilitated group learning practice using Gamestorming and the principles of domain-driven design (DDD)
- https://openpracticelibrary.com/practice/event-storming/
- **Red Hat Open Innovation Labs leverages the Event Storming practice**

EVENT STORMING OUTPUTS

- **Events** which help define the internal functionality of the software
- **Commands** (Actions) initiated by a user or policy/procedure
- **Aggregates** (Entities) which maintain application state
  - **Aggregate Root** which is the top level of an Aggregate hierarchy
    - May identify transactional boundaries within the software
      - May identify separate services or microservices
- **Bounded Context** which identifies boundaries between business domains
  - May identify separate applications or services
- **Read Models** which allow data to be efficiently retrieved
- **Screen Layouts** which visually represent data to a user (optional)

EVENT STORMING OUTPUTS (SIMPLIFIED)

- **Events**
- Commands (**Actions**)
- Aggregates (**Entities**)
- Bounded Context (**Services**)
- Read Models (**Data Views**)
- **Screen Layouts**

EVENT STORMING (TO CODE)

- **Events →** `PhotoCreated`
- Commands (**Actions**) → `createPhoto()`
- Aggregates (**Entities**) → `Photo`
- Bounded Context (**Services**) → `PhotoService`
- Read Models (**Data Views**) → `PhotoWithLikes`
  - Combines data from two bounded contexts, photos, and likes
  - We will see the details in our example application
- **Screen Layouts →** `Add Photo` page

BUILDING A (MODULAR) MONOLITH WITH AN EYE TOWARDS MICROSERVICES

4 LEVELS OF MONOLITHIC MODULARITY

1. Source Control Modularity
   - Utilize multiple Git repositories for 1 monolithic application
   - Scale your application development across teams (**programming in the large**)
2. Build Modularity
   - Utilize Maven Modules or Gradle Projects for building discrete application components
3. Code Modularity
   - Utilize Java (9+) modules to provide granular (package level) contractual guarantees during compile time
   - Separate API code from Implementation code
   - Emit and listen for events for cross-module communication
     - Don’t call another module’s code directly
4. Data Modularity
   - Do not create tight coupling of data between modules

5 LEVELS OF MICROSERVICE MODULARITY

- ✓ Source Control Modularity
- ✓ Build Modularity
- ✓ Code Modularity
- ✓ Data Modularity

New:

5. Deployment Modularity *Warning: Distributed System!*
   - Deploying multiple microservices/functions instead of one application
   - May involve partitioning data into separate deployed databases
   - Inter-service rather than intra-service communication
     - Propagate events between services using a network transport
   - User interface or clients must use multiple services (potentially through an API aggregator)

DDD SUPPORTS DEPLOYMENT MODULARITY

- Partitioned microservices/functions
  - Commands (**Actions**) - separate API per microservice Bounded Context (**Services**)
- Partitioned microservice/function data stores
  - Aggregates (**Entities**)
  - Read Models (**Data Views**)
    - **Screen Layouts**
- Inter-service communication (event-based)
  - **Events** - propagated through a distributed bus between services in an asynchronous manner

WHAT'S THE POINT?

By leveraging DDD and modularity, you can build an application that may be either a monolith or microservices

PHOTO GALLERY SAMPLE APPLICATION

PHOTO GALLERY SAMPLE APPLICATION

- Photos Service
- Likes Service
- Query Service

PHOTO GALLERY SAMPLE APPLICATION

- Photo Gallery Monolith
  - Photos Service
  - Likes Service
  - Query Service

Option #1

PHOTO GALLERY SAMPLE APPLICATION

- Photos Microservice
- Likes Microservice
- Query Microservice

Option #2

WHAT'S THE POINT?

Same code, same modular design, different deployment options

WHAT'S THE POINT?

Monolithic deployments avoid problems of distributed systems

WHAT'S THE POINT?

Monolithic is simpler, but you still have opportunity to scale out in future with microservices

DEMO

TAKEAWAYS

1. Learn the basics of DDD, but you don’t have to be an expert
2. Learn about Event Storming, and consider adopting the practice in your organization
3. Implement the 4 + 1 levels of modularity to easily switch from monolith to microservice, and maybe even serverless functions!
4. Leverage an event/message bus to communicate between services by passing events
    a. Local bus for monolith (ie. Vert.x Event Bus)
    b. Distributed bus for microservices/functions (i.e. Kafka)

TAKEAWAYS

5. Consider starting with a monolith first and break it apart later, only when necessary

RED HAT SUMMIT

THANK YOU


linkedin.com/company/Red-Hat


facebook.com/RedHatinc

youtube.com/user/RedHatVideos


twitter.com/RedHat