micro-service architecture: the essentials

David West, Ph.D.

CEE-SECR 2016

Moscow

Motivation

Critical Business Challenges:

Pace of Change - Adaptability /
Evolvability - Innovation - Scale

Single biggest obstacle to meeting these challenges?

Monolithic computer systems built according to the dictates of software engineering!

(including SE inspired, non-conscious, habits and presuppositions)

Inspiration

Rapid
Application
Development

Object-Oriented
Development

Extreme
Programming
/ Agile Development

Domain-Driven
Design

Service-Oriented
Architecture

Definitions

Microservices are small, autonomous services that work together. —Sam Newman, Thoughtworks

Loosely coupled service-oriented architecture with bounded contexts. —Adrian Cockcroft, Battery Ventures

A microservice is an independently deployable component of bounded scope that supports interoperability through message-based communication. Microservice architecture is a style of engineering highly automated, evolvable software systems made up of capability-aligned microservices.

There is no universal definition. Just Do It!

identifying micro-services

Technically ...

- a function
- a block
- a programming expression
- even an assembler instruction

... are all "micro-services."

(A block is the only one, however, that can be context free)

Think of these as the "atomic" level of micro-service architecture. But building with atoms is really inconvenient. We need a coarser granularity to work with - the "micro-service provider" (element, component, module).

Micro-service Providers

(aka Components, Modules, Elements)

A locus of Behavior.

Found in the "bounded context" of a Domain.

Identified, and differentiated from other elements, strictly on the basis of what they do — what services they provide.

autonomous and "composable"

coupling and cohesion

Determining which services should be 'packaged' in an individual micro-service provider.

Cohesion

- coincidental
- logical
- temporal
- procedural
- informational
  e.g. mainstream OOP
- sequential
- functional
- anthropomorphic

Coupling

- Content (pathological)
  e.g. friends, dot-notation, inheritance
- Common
- External
- Control (Temporal)
- Stamp
- Data (OK if minimal)
- Message

anthropomorphic cohesion

If I am a service provider, like one of those identified in exercise one, what am I willing to do, what am I capable of doing, and what is the least amount of work I need to take on?

Basic Design of a micro-service provider (MSP)

- What I am willing to do — list of services I can provide
- What I need to know in order to do them - list of information items
- How you can ask for a service and what I will provide you — protocol
- Events, changes in myself, that I am willing to share with others.

Services

MSP Name

- service one

- service two

- service three

- service four

Knowledge Required

For each service, what information must I have access to in order to provide that service. E.g. If I can tell you my age, I need to know my age, but that changes constantly so it needs to be calculated anew each time you ask me, therefore I need to know my date-of-birth and today's date.

I have four ways that I can access that information:

- I can remember it (store in a variable)
- the service requester can provide it (message argument)
- I can ask some other MSP for it (in which case I need to know the name of the MSP and what message to send)
- there is a global source of that information (I need to know what message to send)

Knowledge Required (cont)

Divide the back of the index card into two sections: knowledge required and events. Under knowledge required identify each bit of knowledge and how you will access it: (v) variable, (a) argument, (c) collaboration, or (g) global

Knowledge Required	Events
- information item 1 (v) - informationitem 2 (v) - information item 3 (a) - information item 4 (c) - information item 5 (a) - information item 6 (g)	

How to ask — what to expect

message protocol

All Communication between and among MSPs is by message! No dot notation, no pointers, no shortcuts!!

It does not matter if this is inefficient. Resources are cheap and design will compensate.

Sender — Receiver — Selector — (Arguments) — Returned

Sender	Receiver	Selector	(Arguments)	Returned
Usually implicit	Name of MSP - or Micro-service identified by message selector	name of the message	optional, if used, descriptive name of argument	descriptive name of what is returned to sender

(Airplane) — instrumentCluster — myLocation — — aLocation

(me) — Calendar — numberOfDaysSince — aDate — anInteger

Exercise Five

provide a message protocol for each service

On the index cards used in exercise Two, provide a message protocol for each service using the blank lines. There is no need to explicitly identify the sender (could be anyone) or receiver (the MSP named at top of the card) of the message.

MSP Name

- service one
  message — arguments — returned
- service two
  message — arguments — returned
- service three
  message — arguments — returned
- service four
  message — arguments — returned

events

These are analogous to changes of state, with the restriction that they are intrinsic changes to the whole of me, and ones I am willing to share and that others might need to know about me.

Example One: an Instrument

- operational
- disabled
- broken (dead)
- out of calibration

Example Two: an Airplane

- on approach
- in position (for takeoff)
- no radio
- emergency
- hijacked

The Strange Case of "Data"

Each and every small bit of information —
what is commonly called "data" — is in and of
itself a full fledged Micro-service Provider.

Examples of this class of MSP would include: strings, characters, numbers, dates, times, timeStamps, etc., plus data structures like indexedCollections or linkedLists.

Despite the similarity of names, these are NOT types.

Only one of the behaviors of this class of
MSP — i.e. render (or display) self —
is directly attributed to the MSP's role as
a representation of a "fact" or "datum."

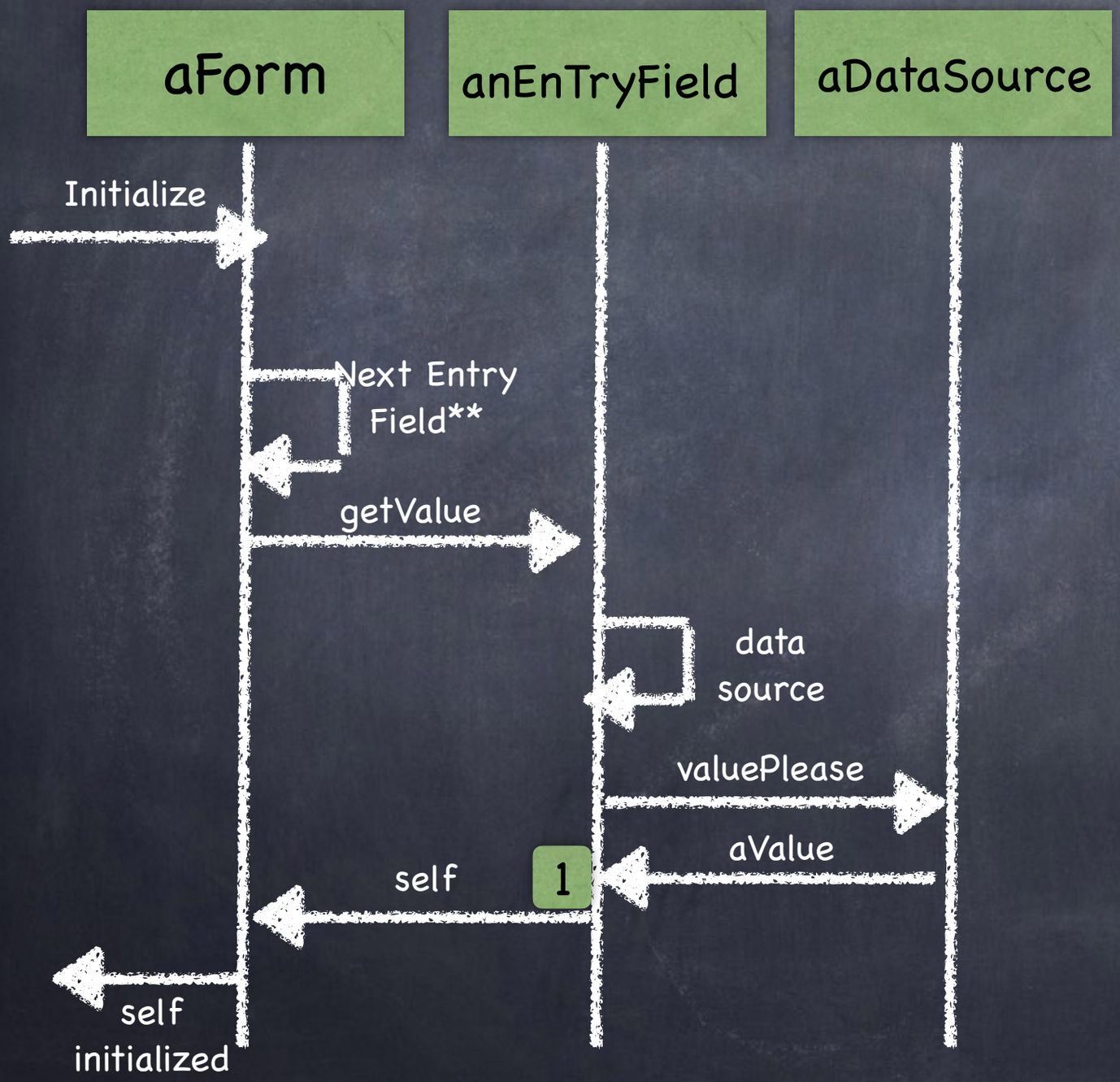
behavior examples for "data MSPs"

String
- iterate across self
- change one or more character to upper/lower case
- provide my length
- renderAs XML
- renderOn: aMedium

Integer
- add self to another Integer
- do other integer arithmetic
- convert to realNumber
- renderOn: aMedium
- convert to String

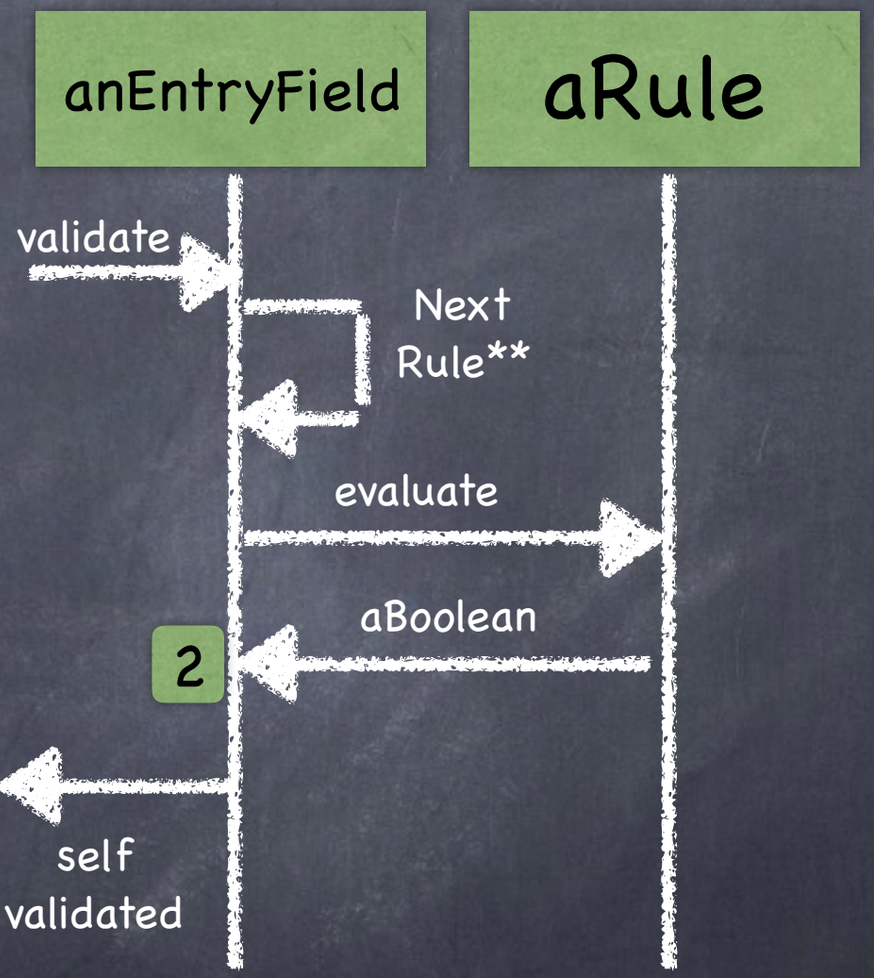
Testing MSPs with illustrated stories

aForm initialize



1 — subStory, entryField Validate

entryField validate



2 — subStory, ifFalse

Designing MSPs for Autonomy and Composability

- reflection services
- "rules" as micro-services
- decouple "shared" services
- collections and observers
- subscription-based event sharing

Reflection Services

There are four broad categories of reflection service:

1 - being able to test itself - important when we discuss devops.

2 - be aware of itself and be able to answer questions like, "can you do X?" 'X' being a specific micro-service identified by message selector.

3 - be aware of changes in itself and be able to notify others of those changes. We will discuss this in detail in "Event Sharing."

4 - be able to respond to messages that alter itself. For example; each MSP can be built such that is individual micro-services - the messages it responds to — are held in a collection and therefore can be added to, or deleted, or modified as needed — at runtime.

Note: "rules" and, by extension, individual micro-services (i.e. programming expressions) can be runtime modifiable.

Rule as Micro-service

A "rule" is an ordered collection of variables, operators, and constants; and,recursively, rules.
A variable is a dyad of receiver and message.

mayRent = True IF [renter age] > 26 AND [renter hasCreditCard]

Rules use collection behavior to compose themselves — including replacing any element of a rule with a variable (at runtime). Rules INITIALIZE (iteratively INITIALIZE each variable); and EVALUATE which applies operators to values returning the result.

Individual Rule = Individual Micro-service.
Individual Variable = Individual Micro-service.

Shared Services

One common refactoring is removing duplicate code and making it an independent function.

Aspect-oriented programming was obsessed with "cross-cutting concerns" which were encapsulated in an object and, using delegation, shared across the code base.

The same philosophy and approach applies to MSA - with micro-service replacing 'function' and MSP replacing object.

collections and observers

Success using MSA is dependent on developers' ability to identify micro-services AND assign them to the appropriate MSP. Two Cases:

1 - separate collection behavior (micro-services) from any MSP needing such services. Example: a portfolio of investments should utilize an internal collection of the actual investments and use that collection to implement behaviors like add, delete, replace, return subset, etc.

2 - Observers are typically concerned with aggregate behavior. Example: watching a set of point-of-sale MSPs and using the information they observe to provide other services, like inventory trends, or sales volume trends.

Event Sharing

Every MSP should be aware of meaningful changes in itself. Part of its public interface is a list of events that it is willing to share with others. Event sharing should always be implemented using an internal (i.e. via delegation) event dispatcher.

Event	Registration Queue	
E -1	[registrant message]	
E -2	[registrant message]	[registrant message]

Any MSP can send a message to another "please register me for E-2 using this message." This message is passed to the eventDispatcher who parses the message to create a registration and places it in the appropriate queue.

Each registration queue is a MSP — a collection that can be ordered and prioritized.

Each registration is a MSP that, like a rule can be evaluated, causing the message to be sent to the registrant.

Applications (Programs)

It is time for the venerable, obsolete, and harmful top-down command and control programming paradigm to die, die, die!

If MSPs are to be autonomous, composable, modifiable, and "hot swappable" a new metaphor for program structure is required — the 'script or 'screenplay'.

Each script or screenplay is itself a MSP — with ordered collection behavior — that can be modified, at run time, by asking the embedded collection to add / delete / modify / or reorder elements.

Architecture

Appropriate architectures for MSA are limited — three obvious candidates and a third, not so obvious:

Pipes and Filters

freely connected MSPs - any given architecture being a fixed script.

N-tier Client-Server

essentially peer-to-peer with MSPs as the peers.

Blackboards (variants: 'whiteboards' 'smartboards')

communication spaces for self-directed MSPs

"The System"

Stop trying to replicate a complex adaptive system as a deterministic one (that's the only kind that a computer can implement).

Instead, modify (by adding, deleting, and re-organizing MSPs in context!

This is the real lesson of Domain-driven Design.