Modernizing Systems

with Microservices, Hystrix and RxJava

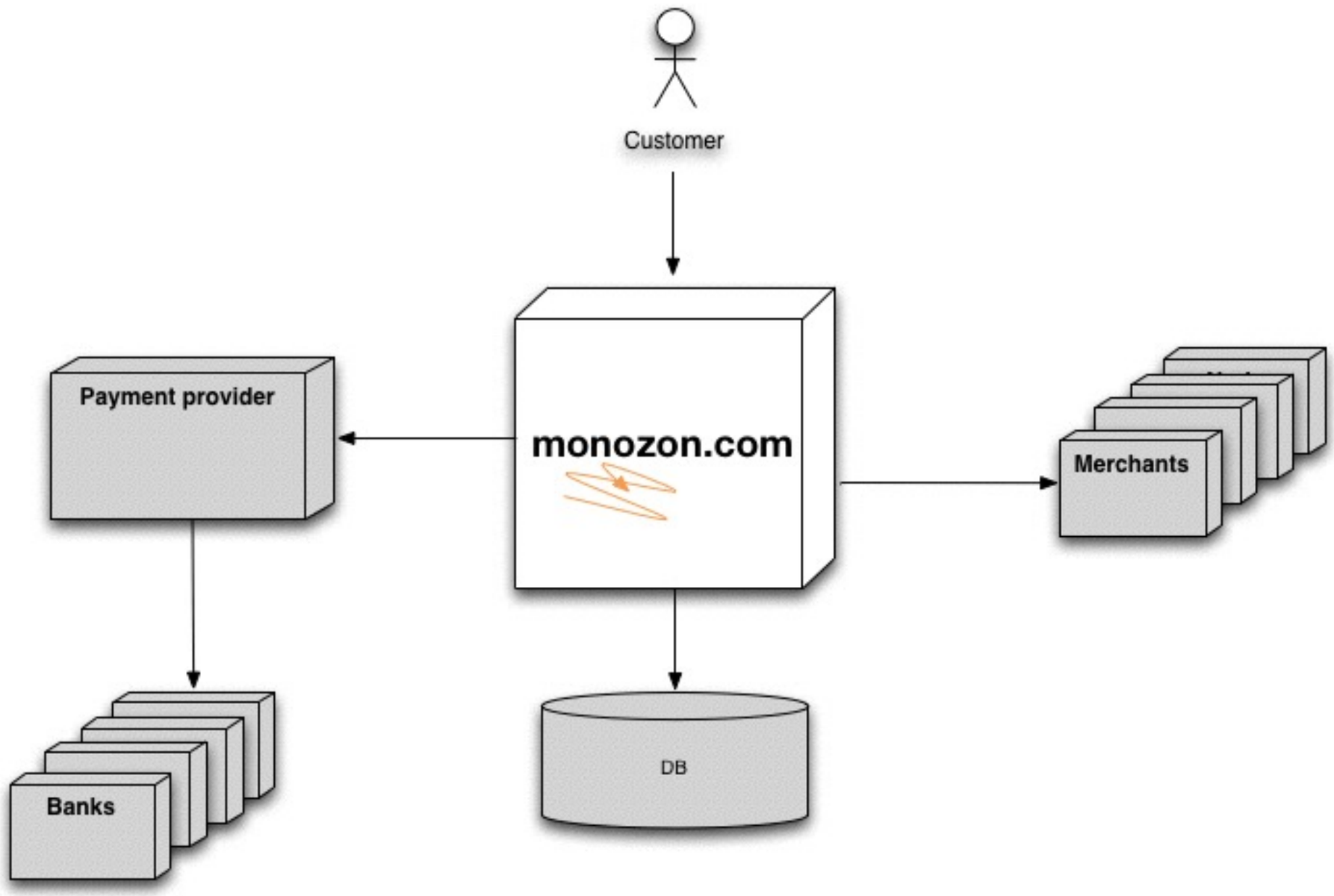
Holger Kraus, Arne Landwehr

OOP, München, Feb 4, 2016

A typical System

monozon.com

The context



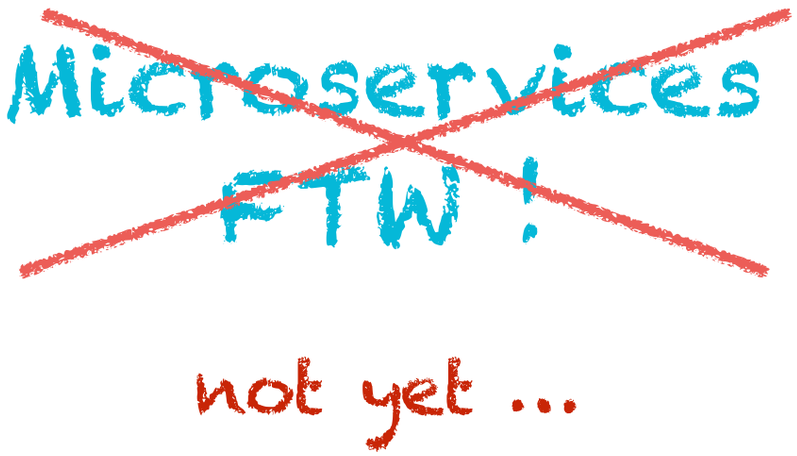
Current problems

- Maintenance is difficult
- New features need a lot of time
- Very unstable
- Outdated technology
- Doesn’t scale

+ frustrated developers :(

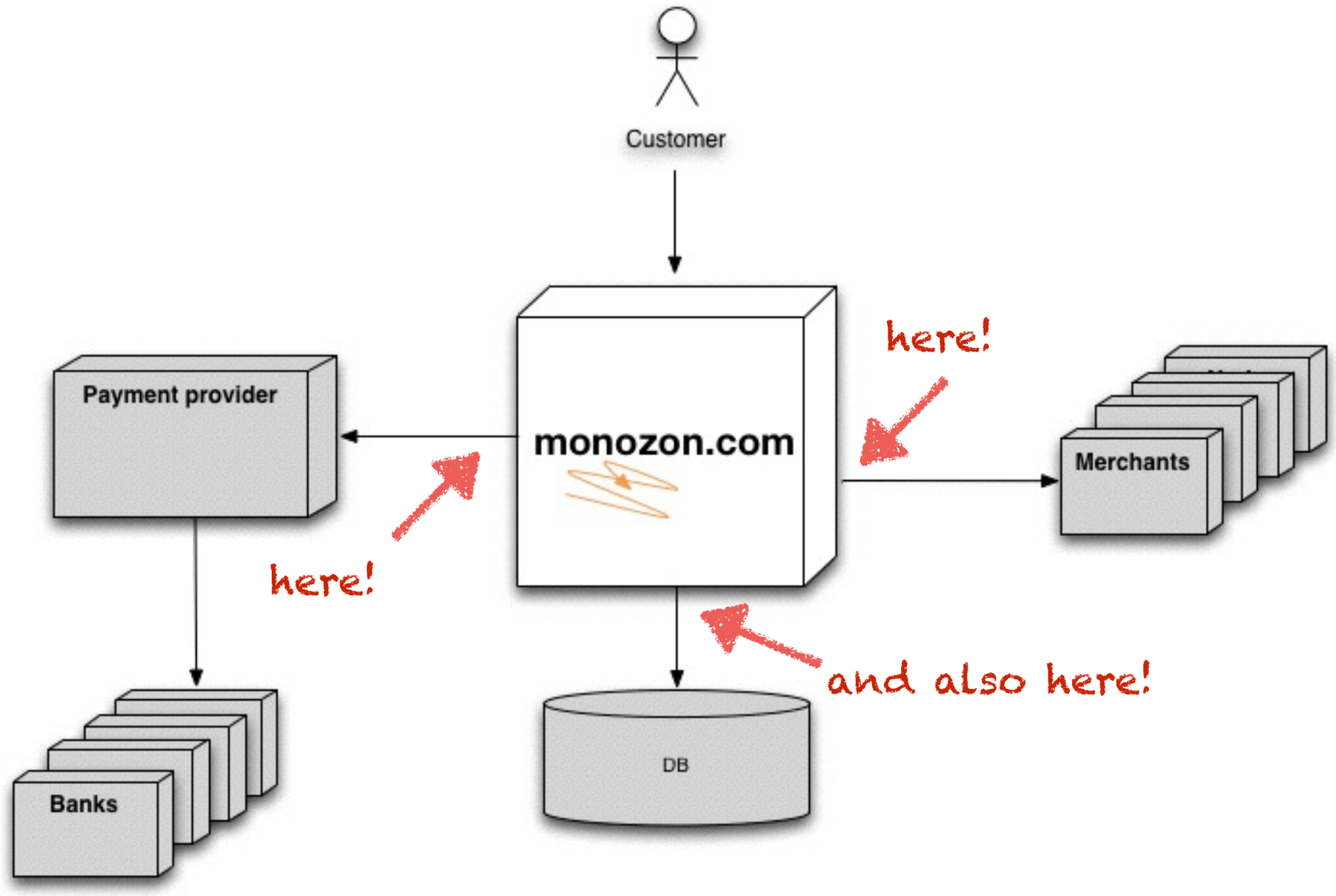
Current problems

- Maintenance is difficult
- New features need a lot of time
- Very unstable
- Outdated technology
- Doesn’t scale



Stabilize first!

External dependencies



Cascading Failures

https://www.flickr.com/photos/benstassen/2991003141/

Stability patterns

- Timeouts
- Circuit Breaker
- Bulkhead

... Fail Fast, Steady State, Handshaking, Test Harness, Decoupling Middleware

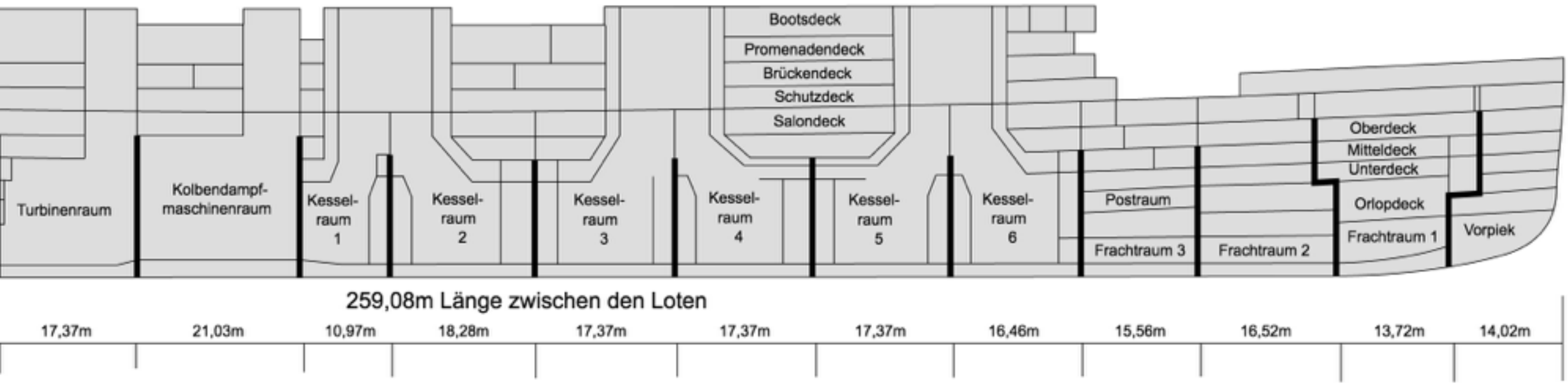
Timeout

https://www.flickr.com/photos/55293400@N07/15564061004

Bulkheads

https://www.flickr.com/photos/10413717@N08/6935206524

Ships



http://de.wikipedia.org/wiki/RMS_Titanic#/media/File:Titanic_Structure_de.svg

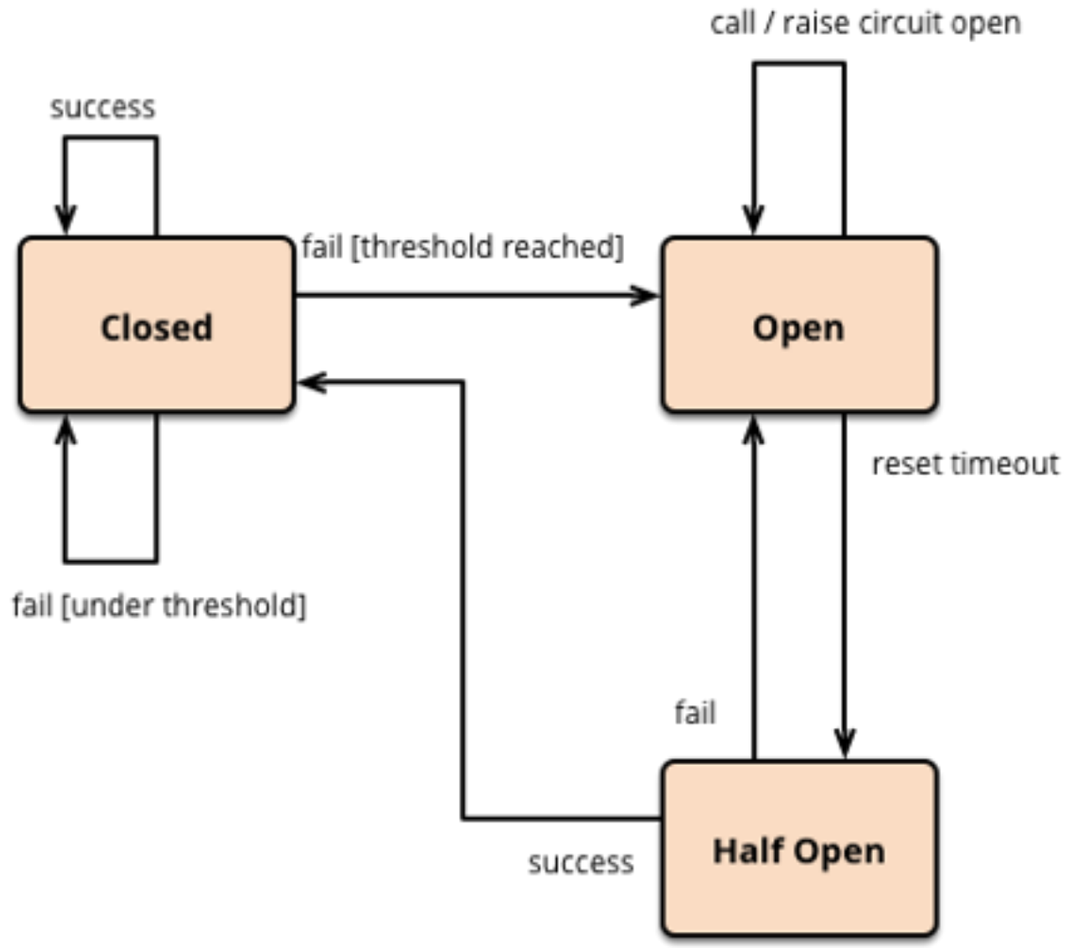
Bulkheads and IT

- Thread pools
- Database connection pools
- Instances
- Server
- Data center

Circuit Breaker

https://www.flickr.com/photos/leafbug/409950515/

Circuit Breaker

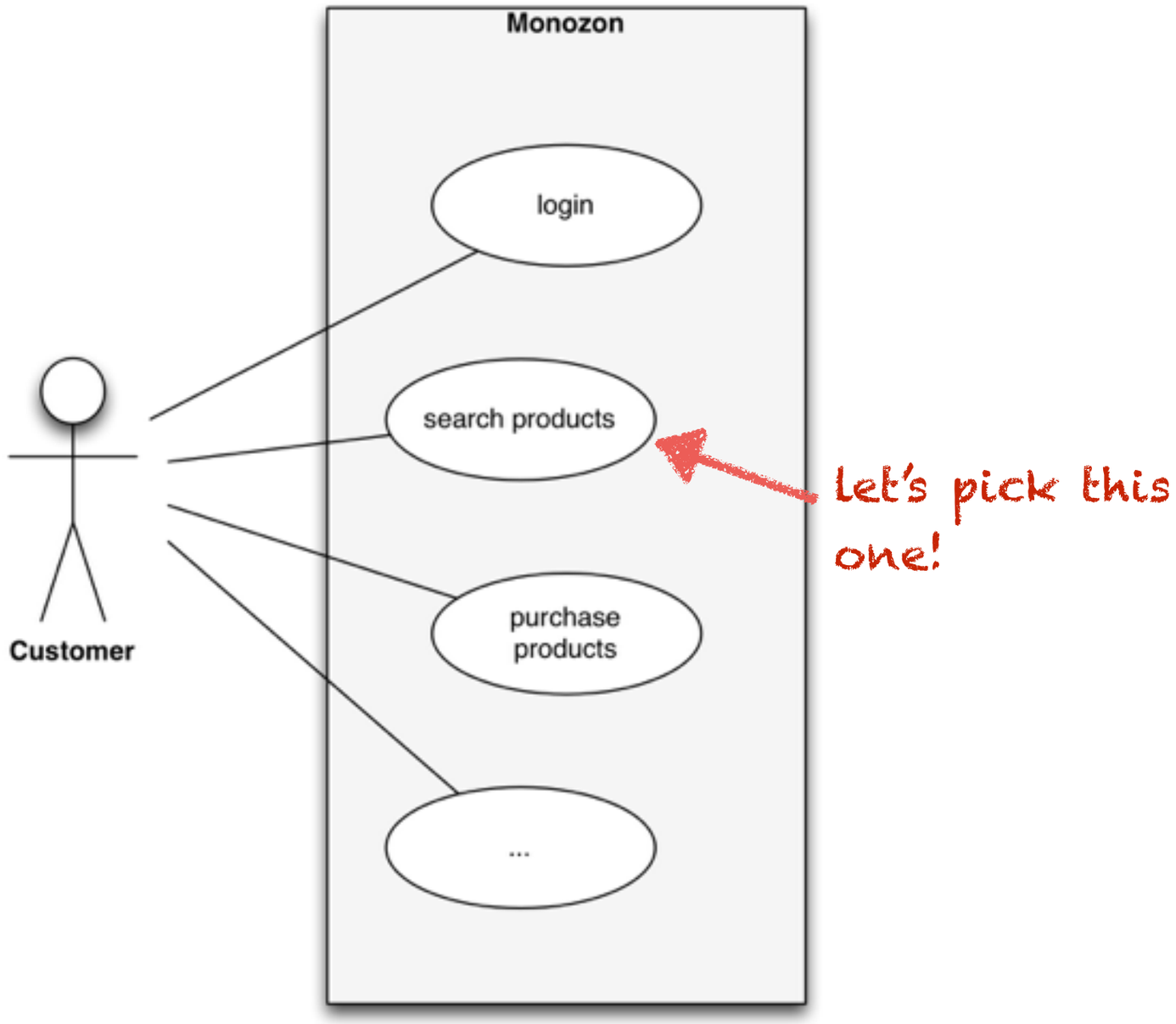


http://martinfowler.com/bliki/CircuitBreaker.html

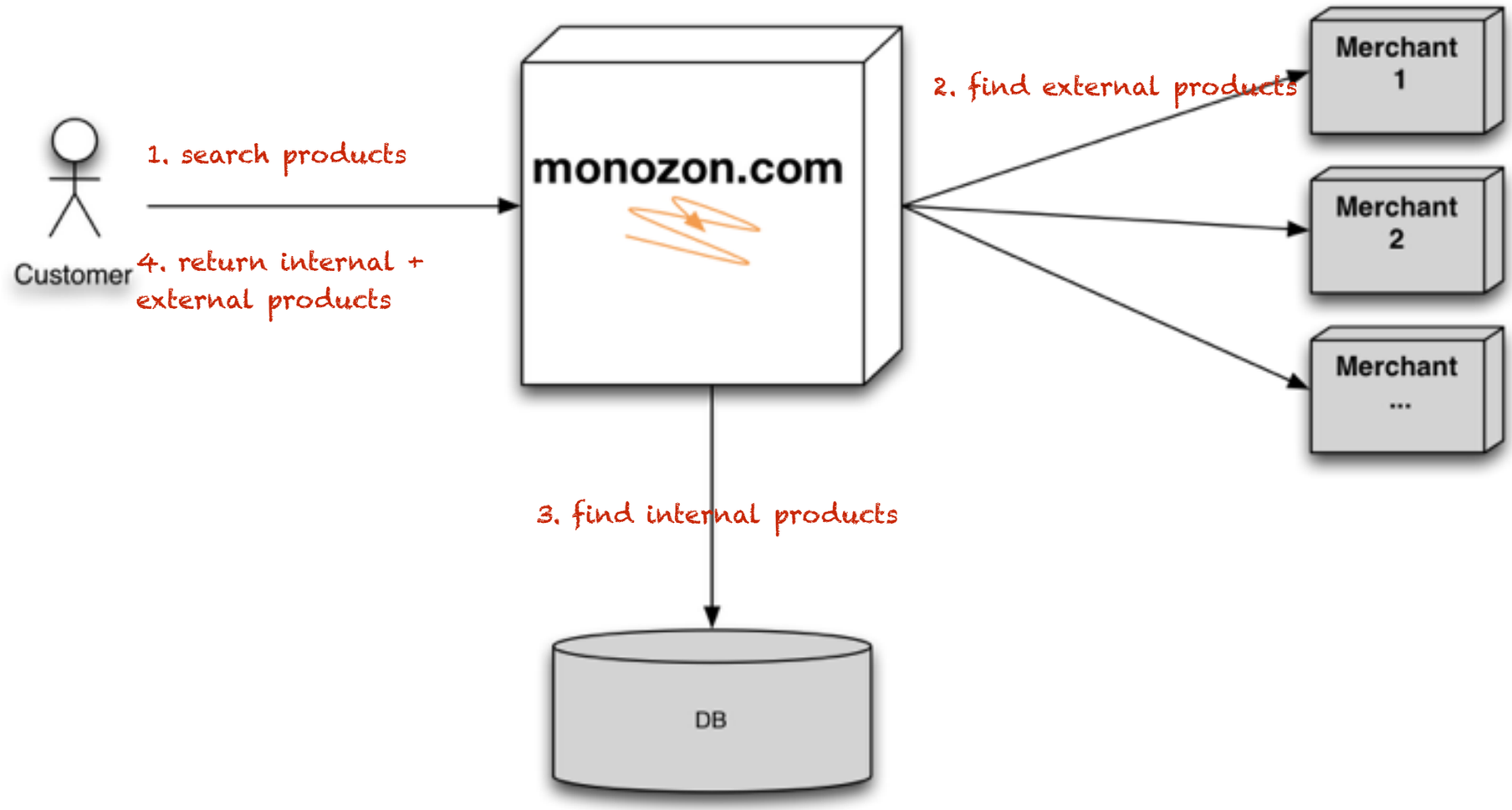
Hystrix

- Library from Netflix
- Resilience Library
- Command Pattern
- Metrics
- Dashboard

Use Cases


Monozon
login
search products
purchase products
...
Customer
let's pick this one!

Search Products



Search Products

```
➜ /  http GET http://monozon:8080/products | jq '.'
[
  "internalProduct_1",
  "internalProduct_2",
  "merchant_1",
  "merchant_2",
  "merchant_3",
  "merchant_4"
]
```

Call without Hystrix

```
private List<Product> findProducts(String query) {
    ClientResponse clientResponse =
            Client.create()
                    .resource("http://merchant1/products/")
                    .queryParam("query", query)
                    .get(ClientResponse.class);

    return toProduct(clientResponse);
}
```

cascading failures incoming!

Simple Command

```
public class GetMerchant1Products
        extends HystrixCommand<List<Product>> {

    private final String query;

    public GetMerchant1Products(String query) {
        super(HystrixCommandGroupKey.Factory.asKey("merchant1"));
        this.query = query;
    }

    @Override
    protected List<Product> run() throws Exception {
        return findProducts(query);
    }
}
```

Execute it!

```
public List<Product> findExternalProducts(String query) {

    List<Product> productList =
            new GetMerchant1Products(query).execute();

    List<Product> merchant2Products =
            new GetMerchant2Products(query).execute();

    productList.addAll(merchant2Products);

    return productList;
}
```

Execute it asynchronously

```
public List<Product> findExternalProducts(String query) {

    Future<List<Product>> merchant1ProductsFuture =
            new GetMerchant1Products(query).queue();

    Future<List<Product>> merchant2ProductsFuture =
            new GetMerchant2Products(query).queue();

    List<Product> productList = new ArrayList<>();
    productList.addAll(merchant1ProductsFuture.get());
    productList.addAll(merchant2ProductsFuture.get());

    return productList;
}
```

Static Fallback

```
@Override
protected List<Product> run() throws Exception {
    return findProducts(query);
}

@Override
protected List<Product> getFallback() {
    return Collections.emptyList();
}
```

In case Merchant 2 is down

```
➜ /  http GET http://monozon:8080/products | jq '.'
[
  "internalProduct_1",
  "internalProduct_2",
  "merchant_1",
  "merchant_3",
  "merchant_4"
]
```

something is missing here

Stale Caches

```
@Override
protected List<Product> run() throws Exception {
    List<Product> products = findProducts(query);
    Cache.put(query, products);
    return products;
}

@Override
protected List<Product> getFallback() {
    List<Product> staleProducts = Cache.get(query);
    return (staleProducts != null)
            ? staleProducts
            : Collections.emptyList();
}
```

Nested Commands

```
@Override
protected Payment run() throws Exception {
    return authorizePayment();
}

@Override
protected Payment getFallback() {
    BackupPaymentGatewayCommand command = new BackupPaymentGatewayCommand();
    return command.execute();
}
```

Configuration

- Netflix archaius used internally
- Configuration at runtime is possible
- allows a faster PDCA cycle and could prevent server restarts
- overall highly configurable and well documented

Command Groups

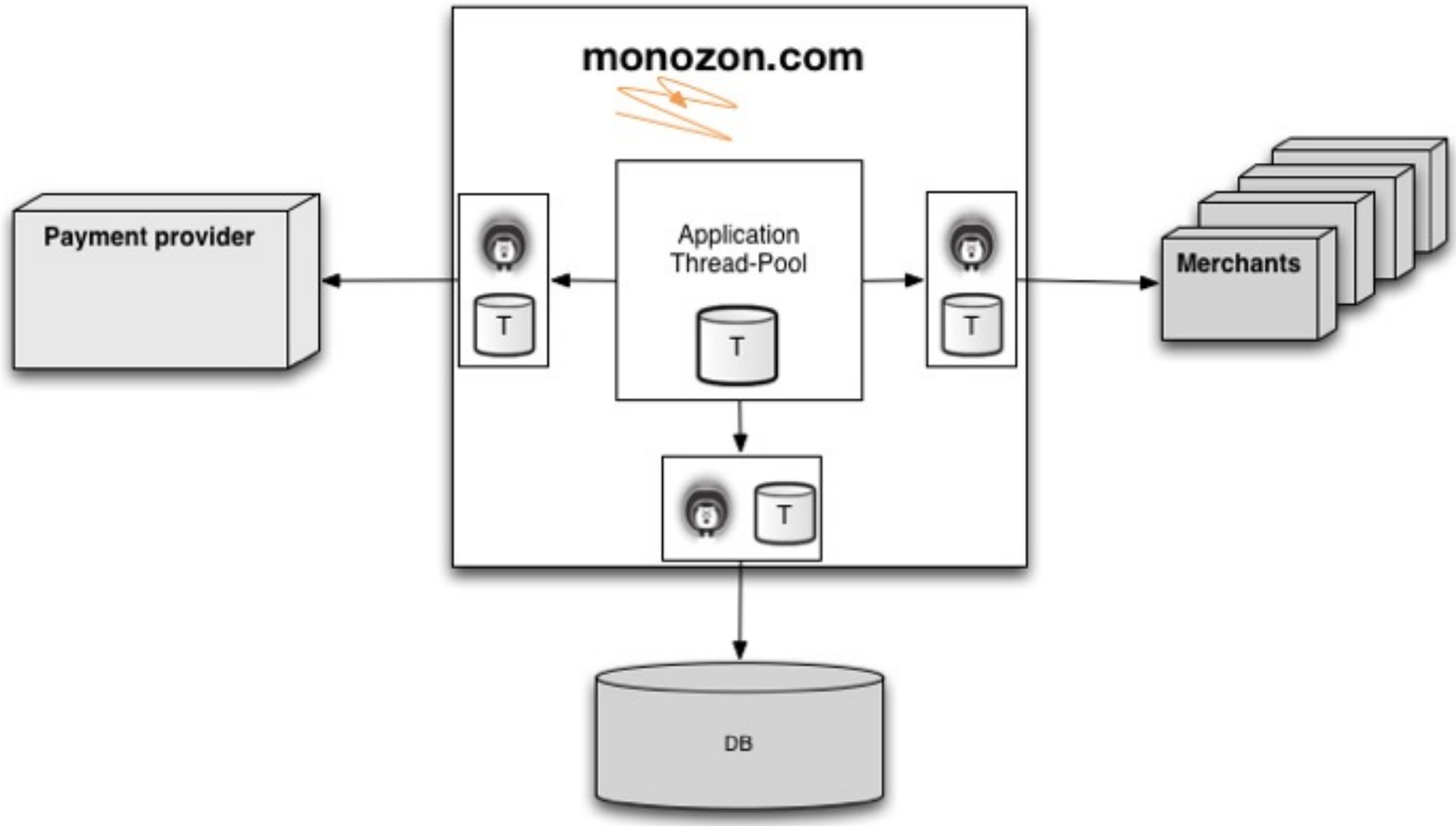
- every Command has a name
- you can combine commands to groups
- groups aggregate commands for metrics, dashboards etc.
- per default a group determines the thread pool

Thread Pool sizing

requests per second at peak when healthy × 99th percentile latency in seconds + some breathing room

Little's law

The stabilized system



Demo

More advanced features

- Request Collapser
- Request Cache
- Semaphore

Let’s recap

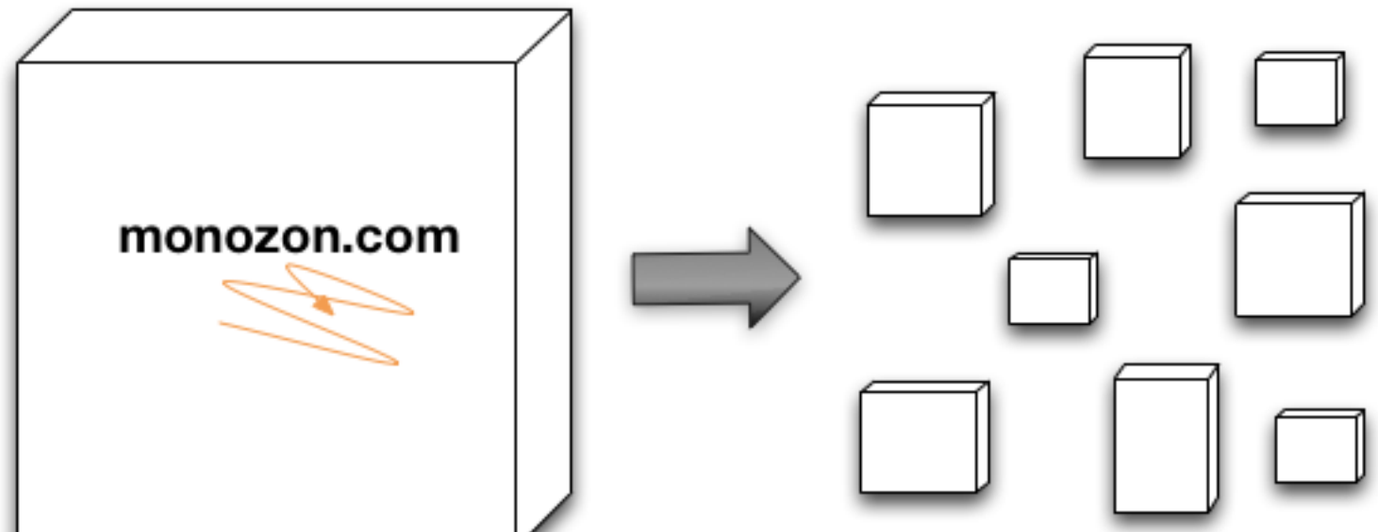
- Stabilize first to enable further distribution
- Hystrix is easy to integrate in existing systems
- it includes Bulkhead, Timeouts and Circuit Breakers out of the box
- Application monitoring as a side benefit

And now?

Current Problems

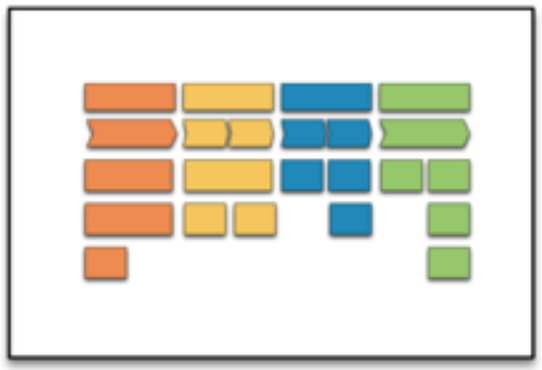
- Maintenance is difficult
- New features need a lot of time
- ~~Very unstable~~ => enables further distribution
- Outdated technology
- Doesn’t scale

Microservices!



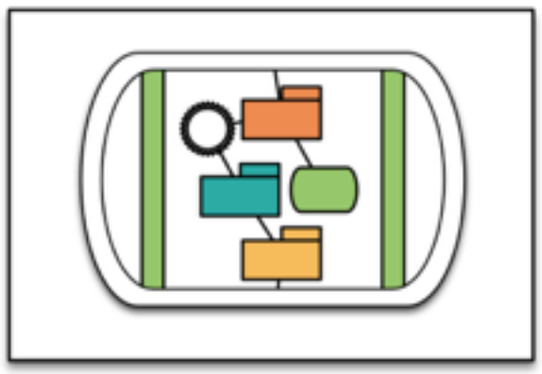
but how to get started ????

Architectural Decisions



› Domain Architecture

› Macro Architecture

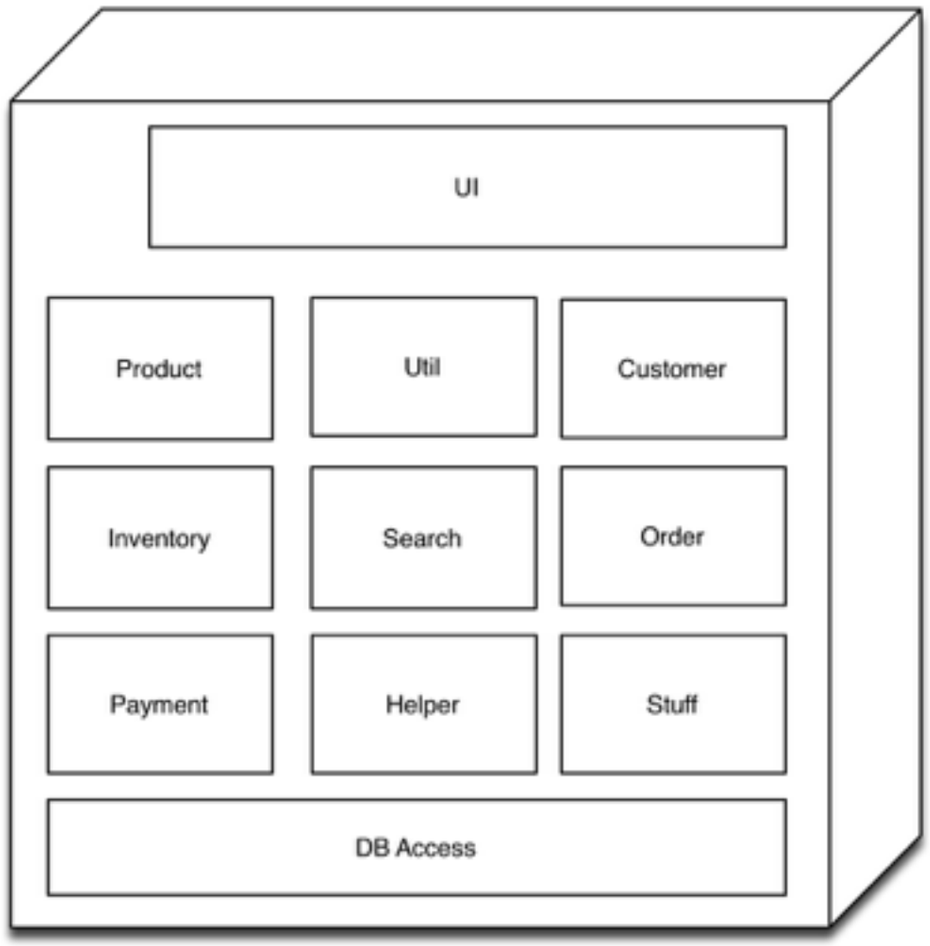


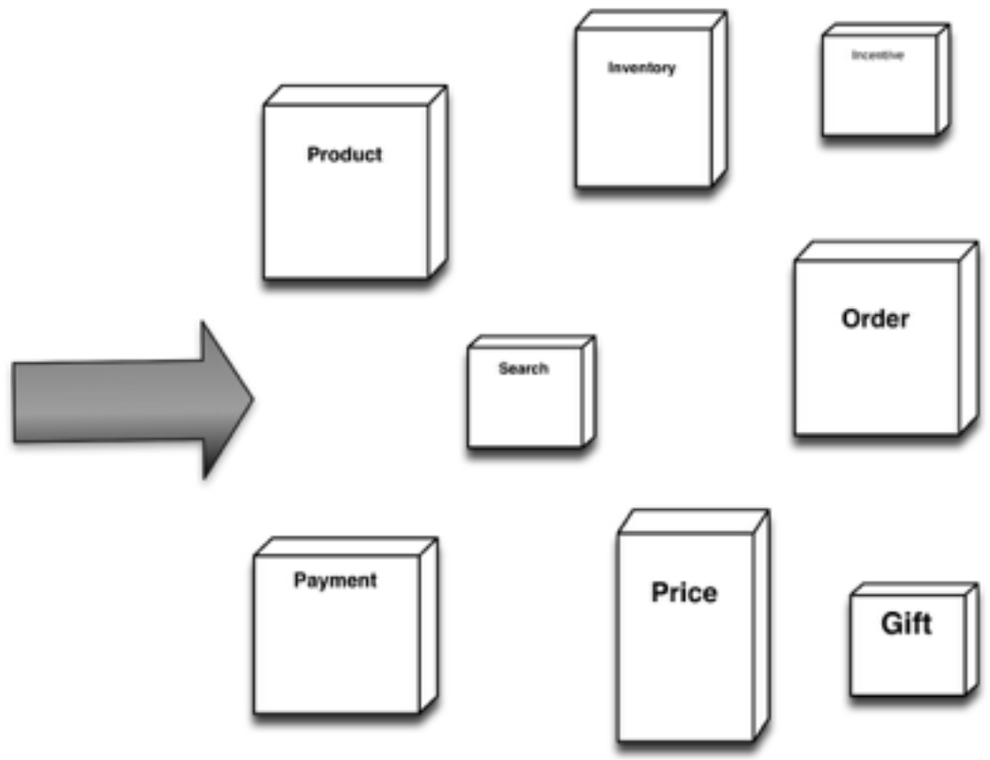
› Micro Architecture

https://speakerdeck.com/aheusingfeld/microservices-meet-real-world-projects-lessons-learned

Domain Architecture

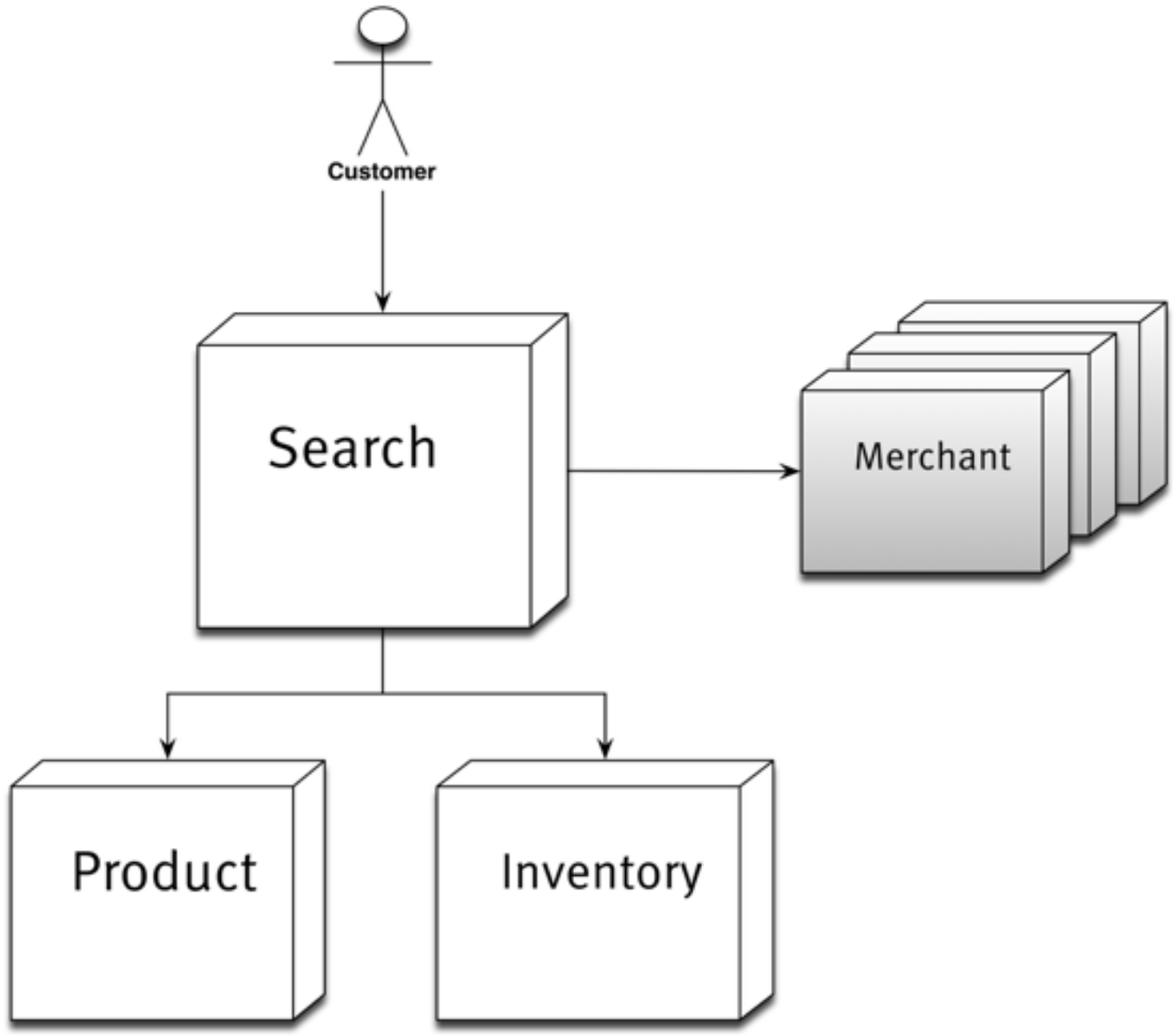
which boxes do we need ?





let the monolith guide you!

Domain: Search



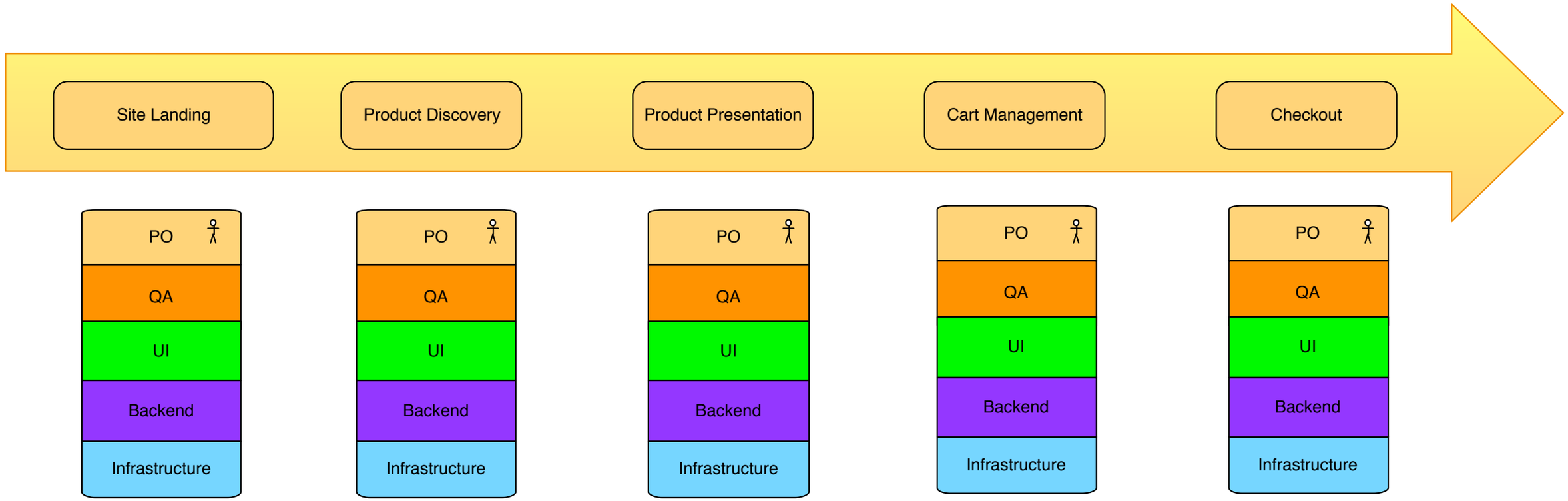
Cut you domain „MECE“

The 5 key phases of the model and the main driver per each phase

mutually exclusive and collectively exhaustive

***** Riccardo Mangiaracina, The eCommerce Customer Journey: A Model to Assess and , Compare the User Experience of the eCommerce Websites, Journal of Internet Banking and Commerce

Domains and Teams



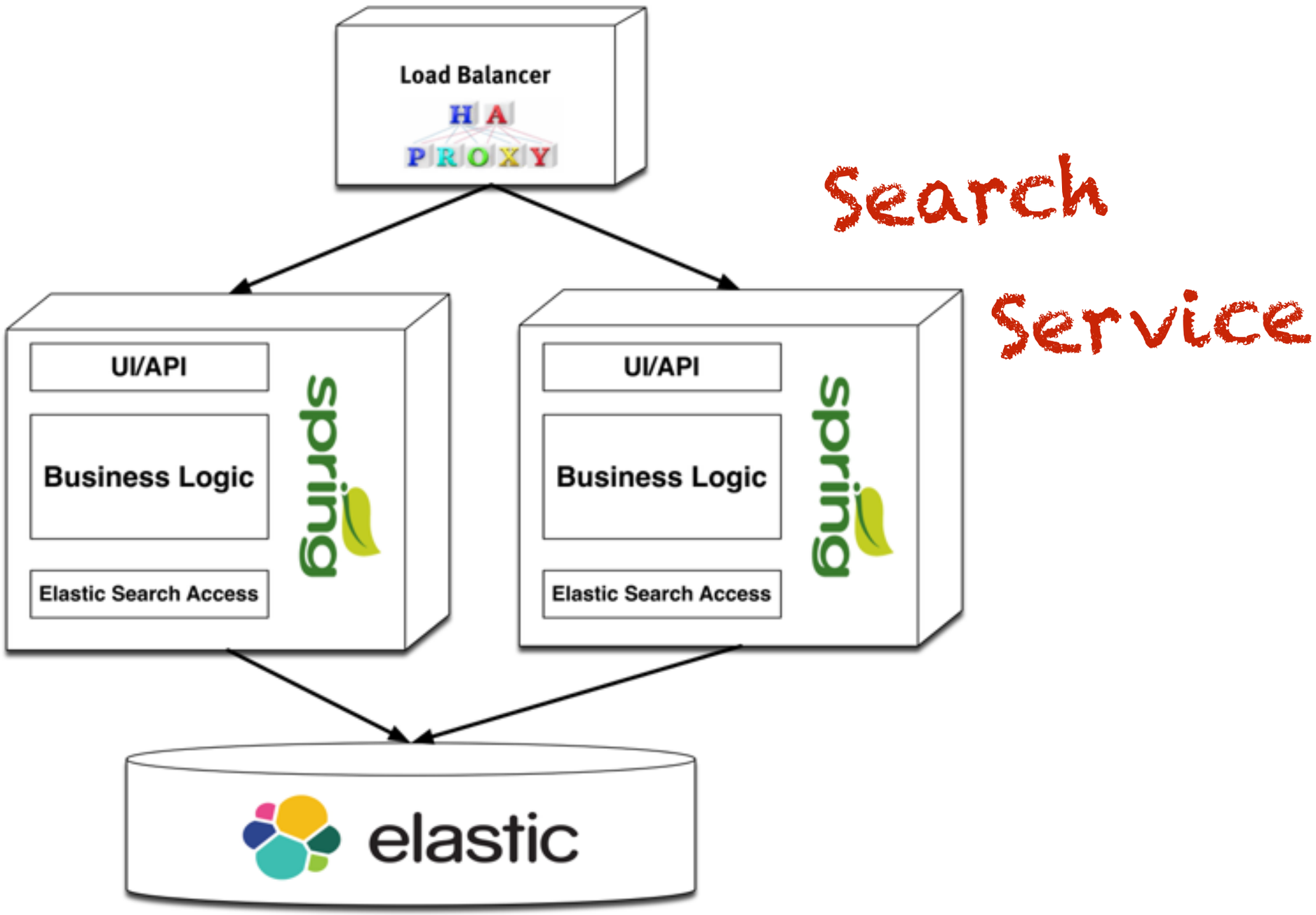
Macro Architecture

what's the same for all boxes ?

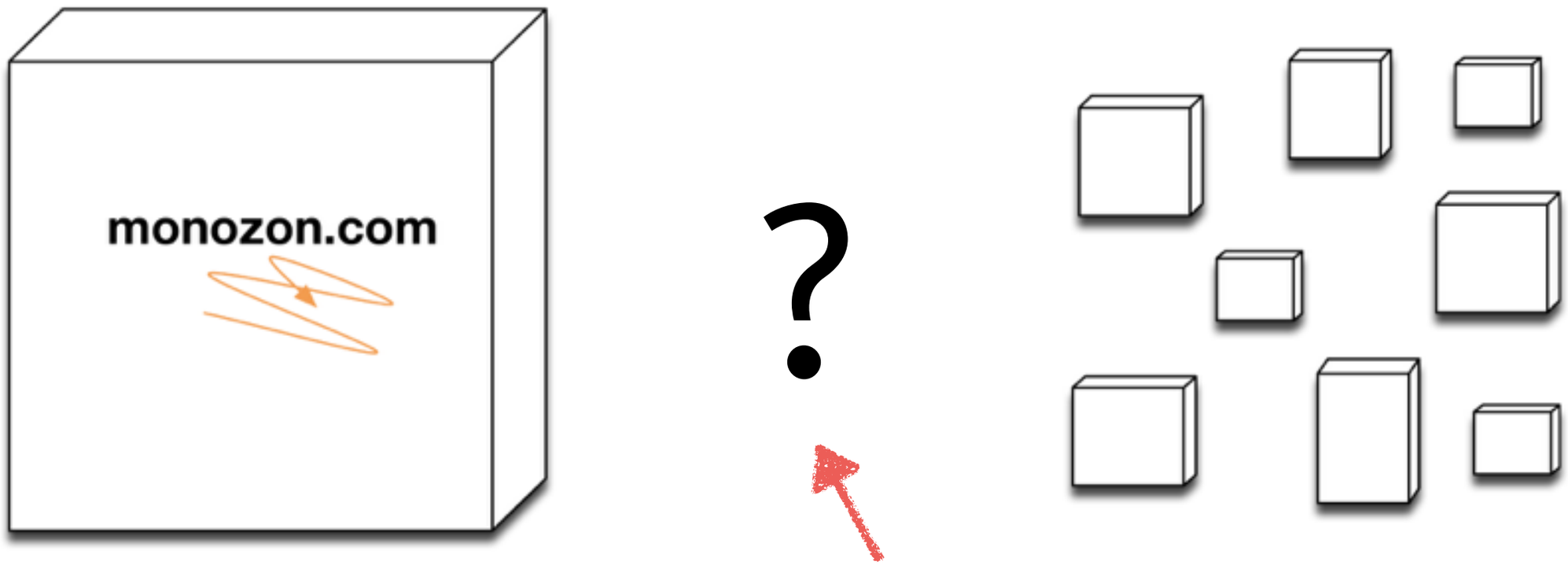
	Monozon:
› Integration	= API
› Deployment	= Docker
› Formats	= JSON
› Protocols	= HTTP + AMQP
› Reduce Choices	= Java, Go

pick your own!

Micro Architecture

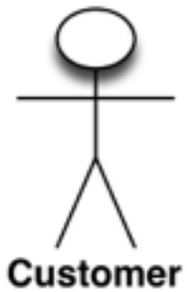


Migration Path



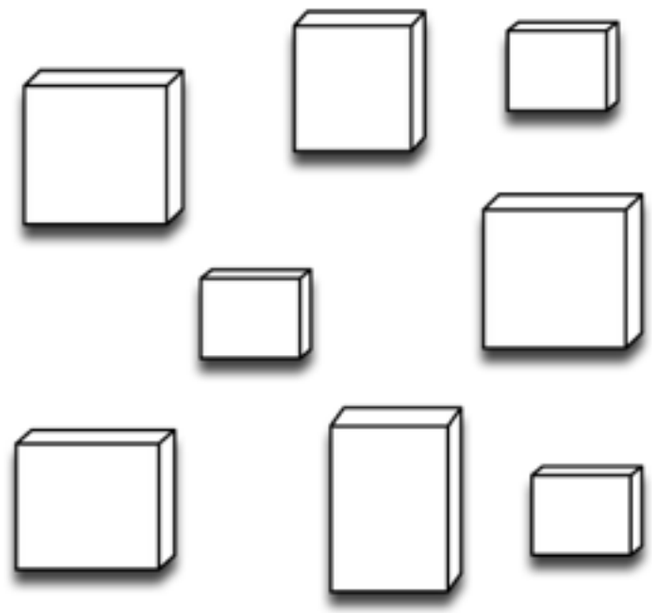
Big Bang

a.k.a REVOLUTION

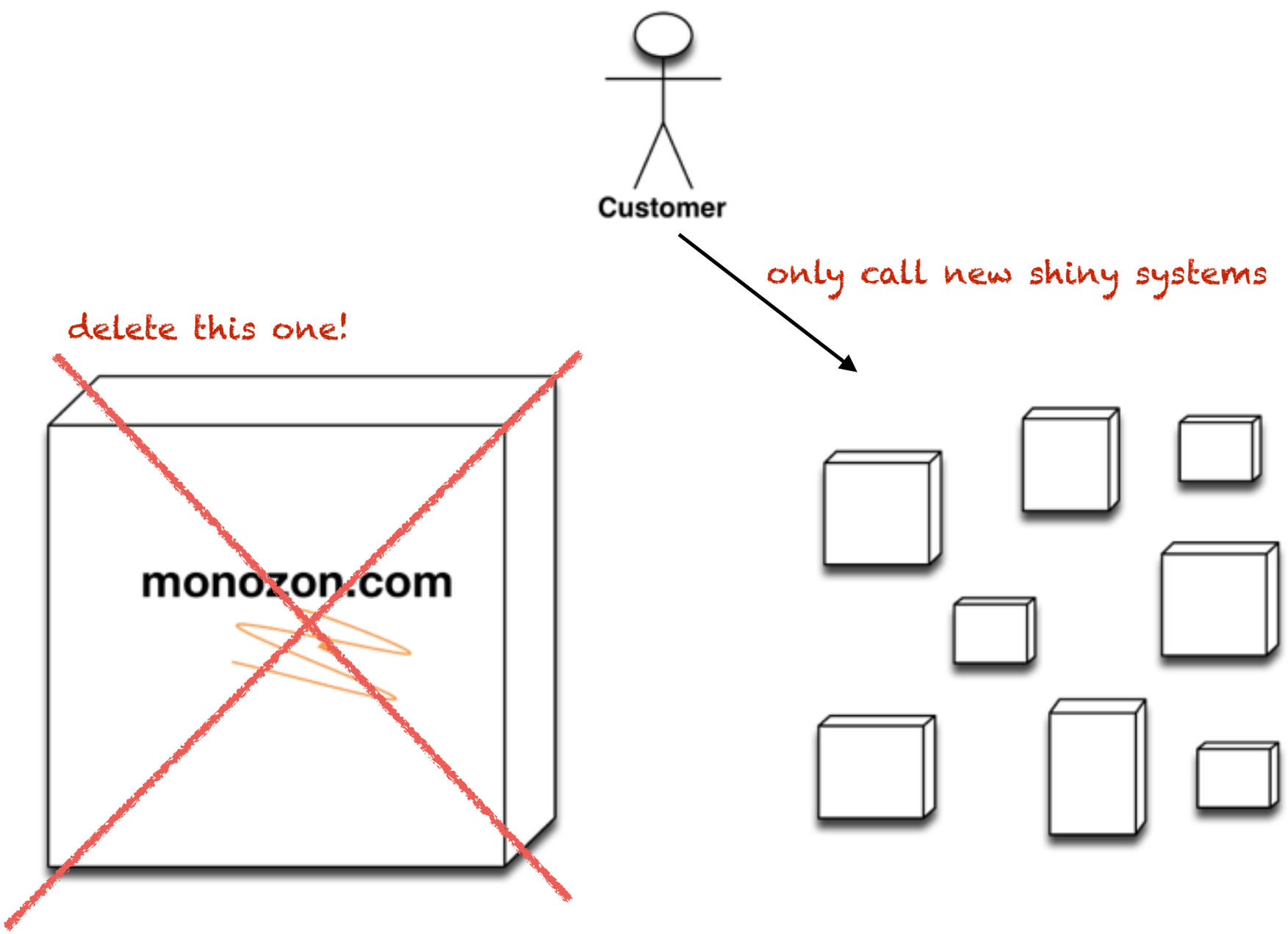


only call old monolith

meanwhile build new systems:

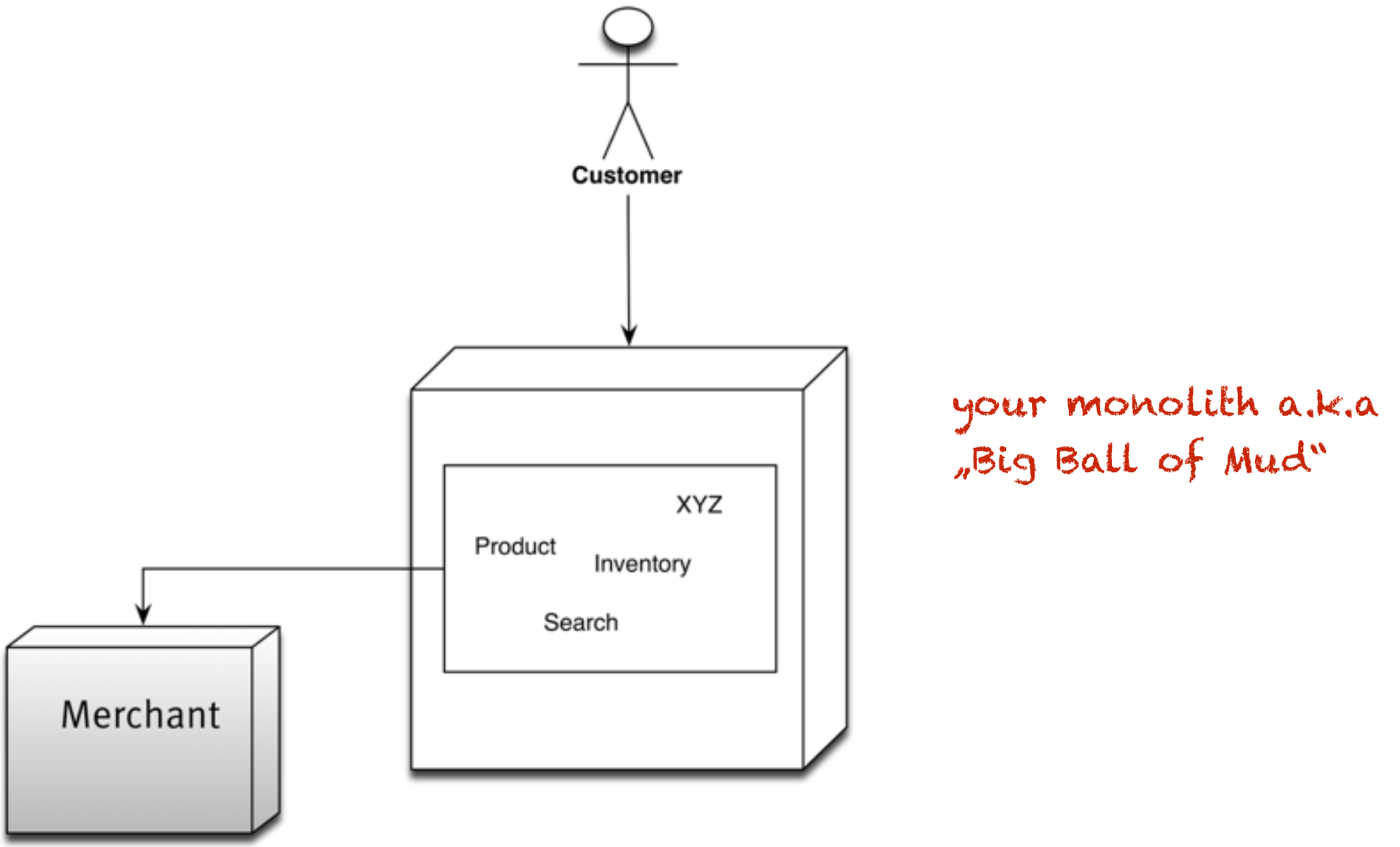


Big Bang


Customer
only call new shiny systems
delete this one!
monozon.com

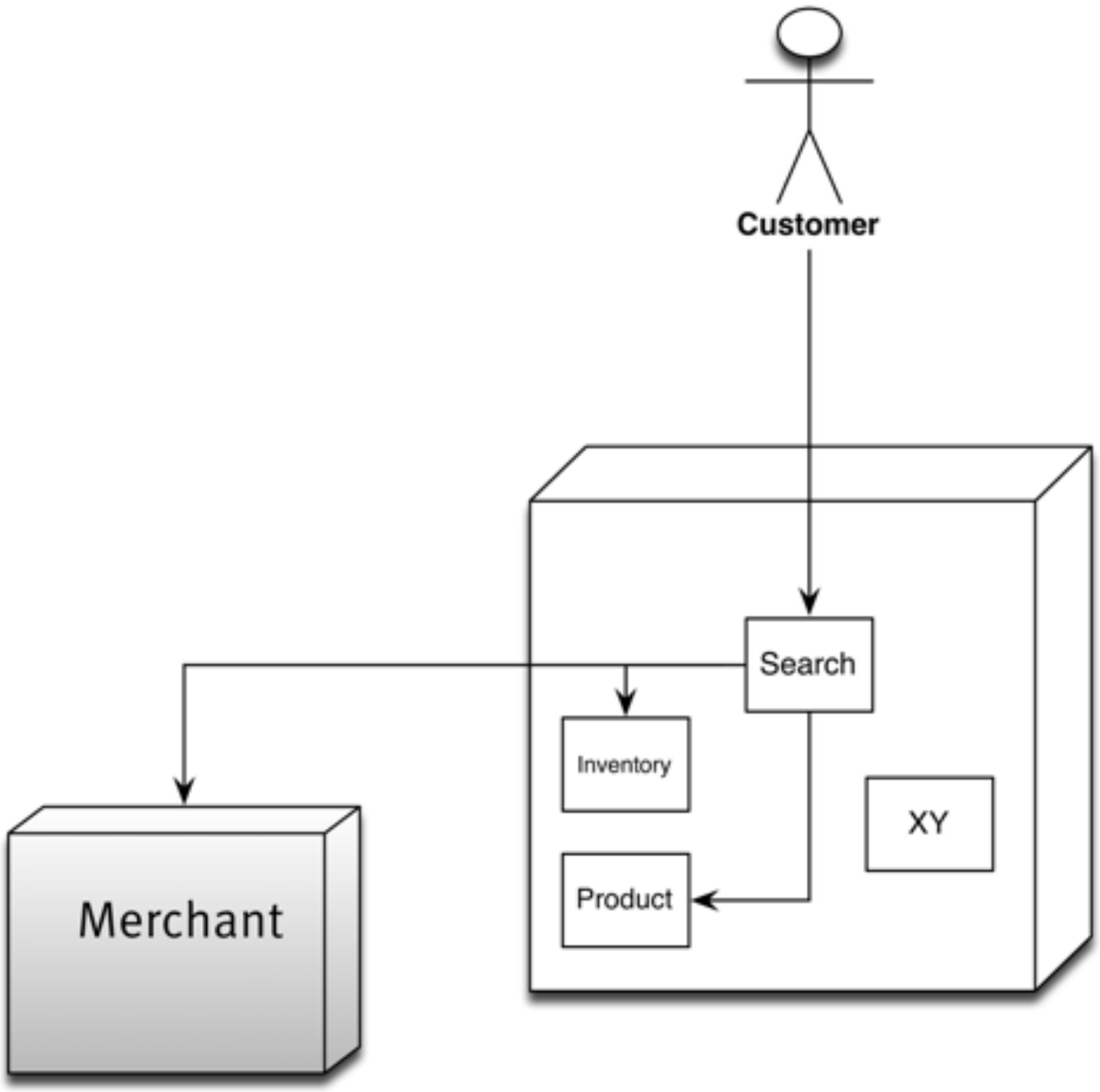
Strangler

a.k.a EVOLUTION



your monolith a.k.a
„Big Ball of Mud"

Strangler

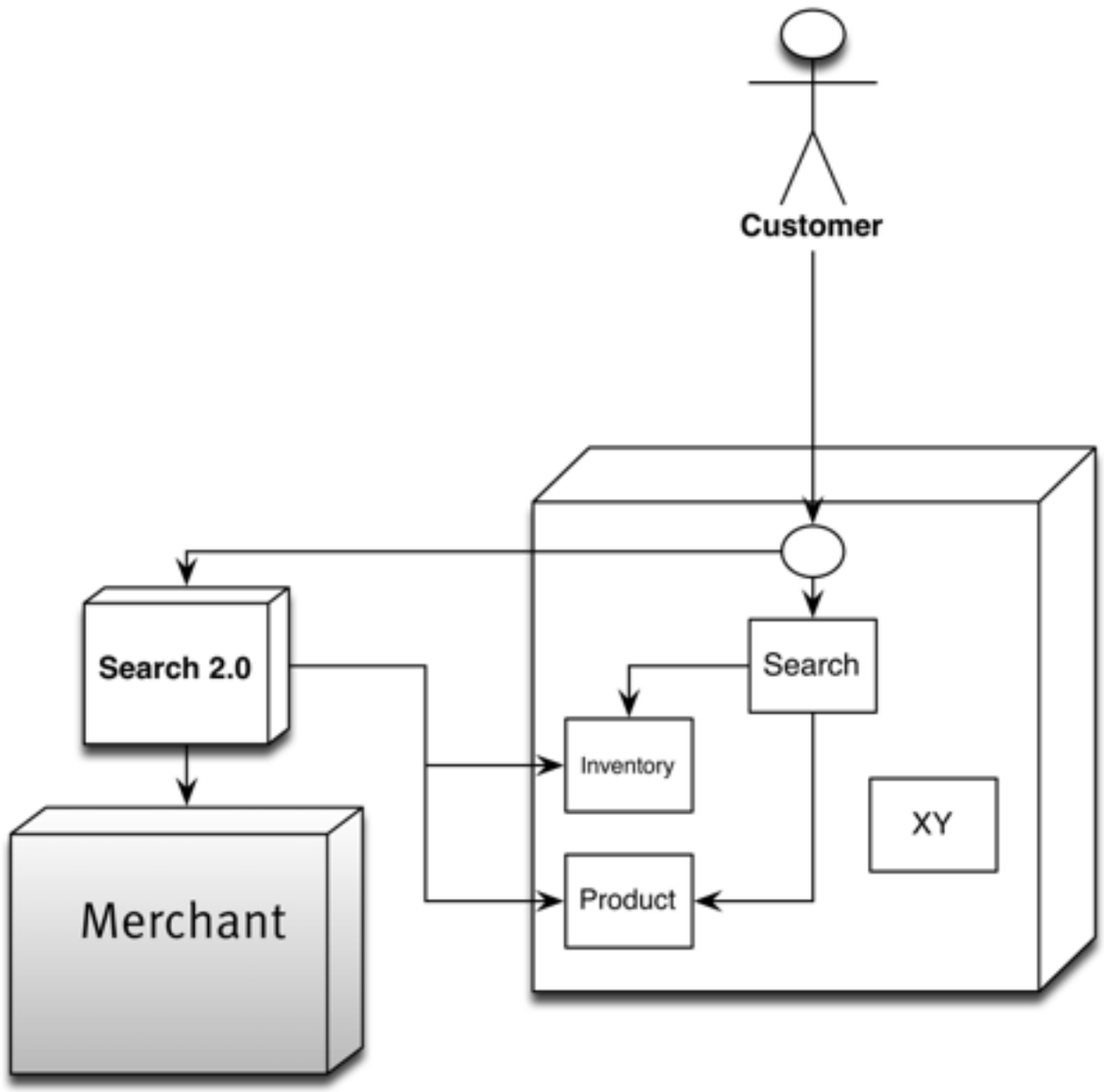


give your monolith
internal structure
and define interfaces

...

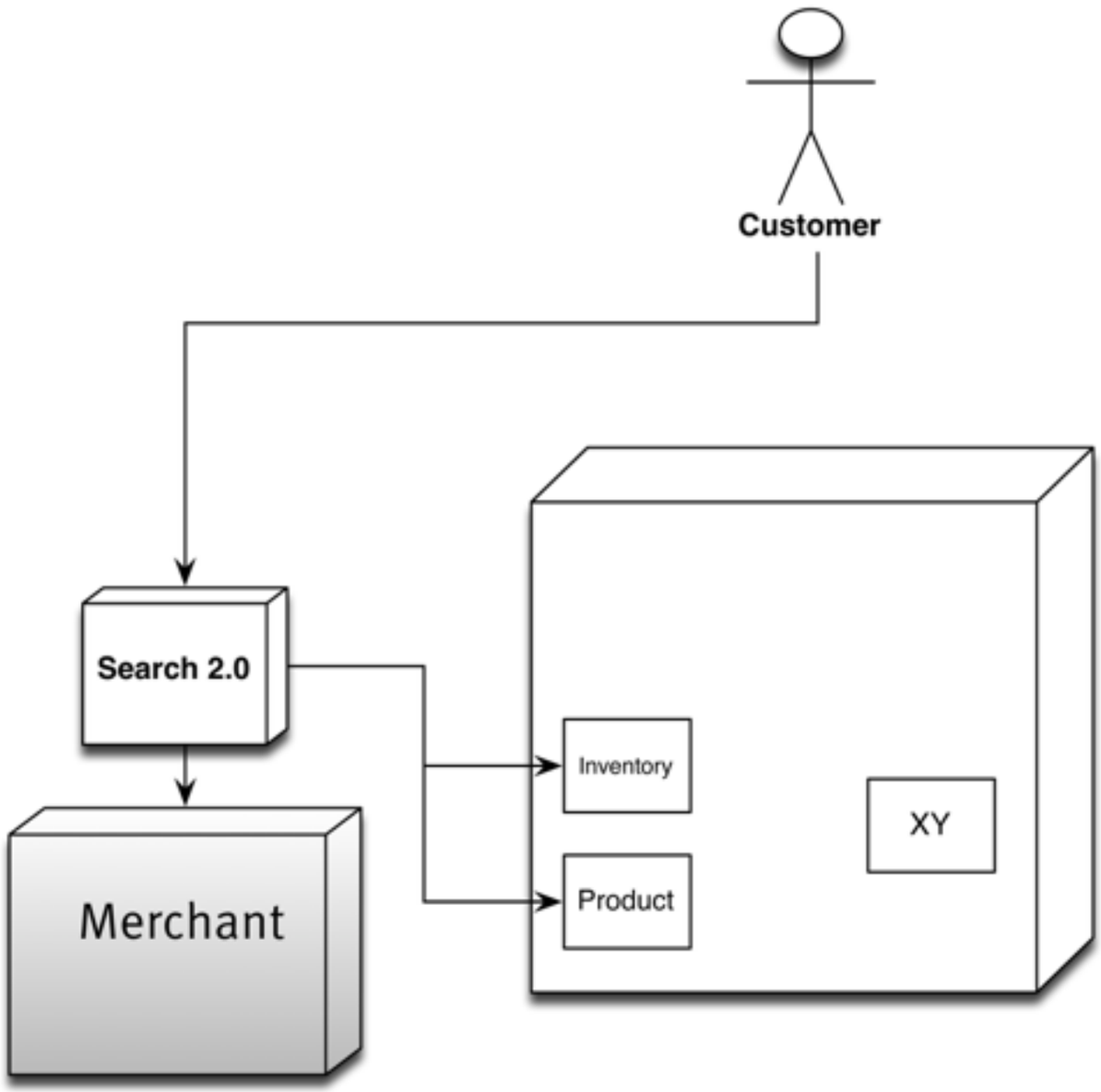
if possible.

Strangler



Extract your new service and redirect calls to it

Strangler

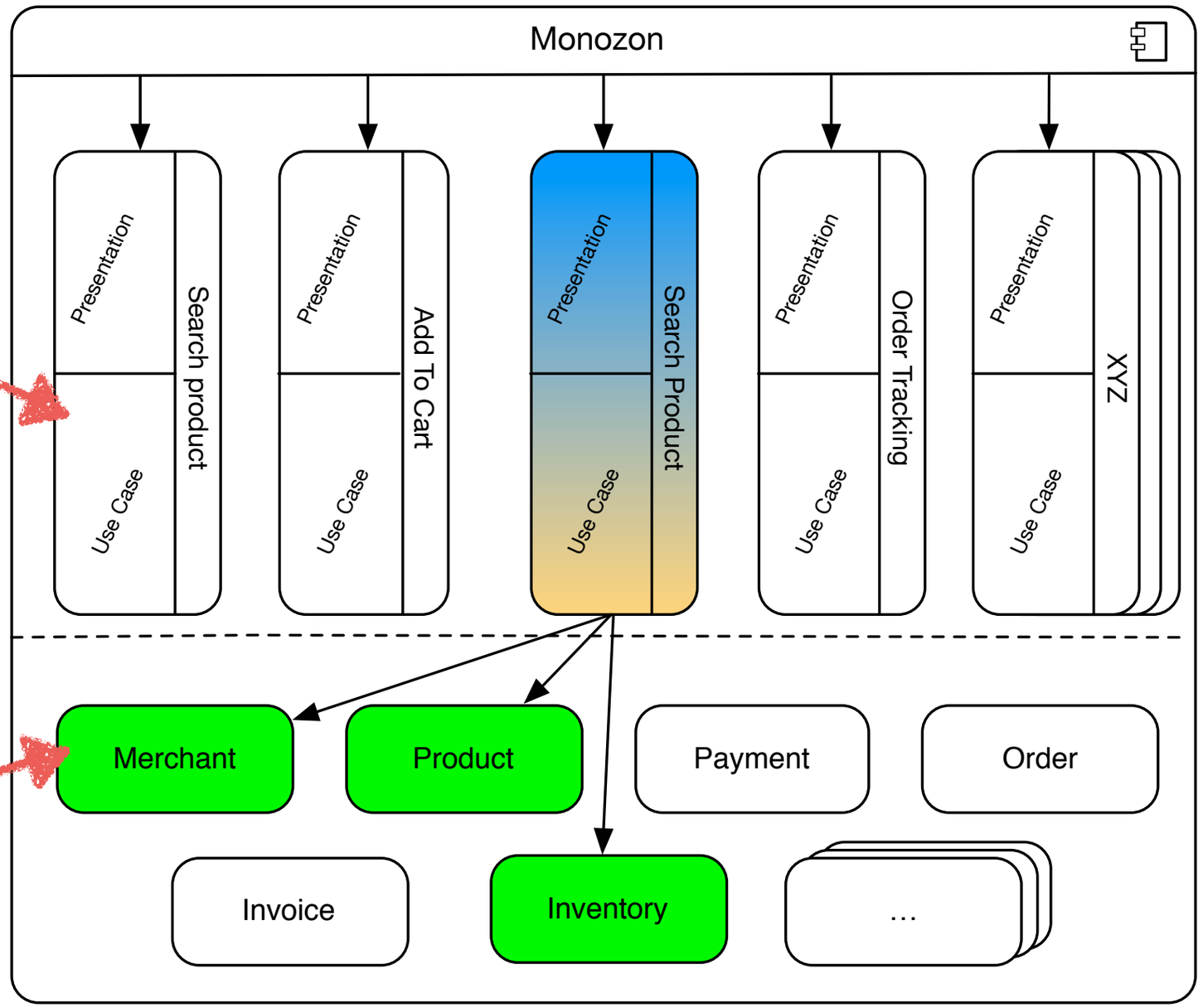


Once unused strangle the old code

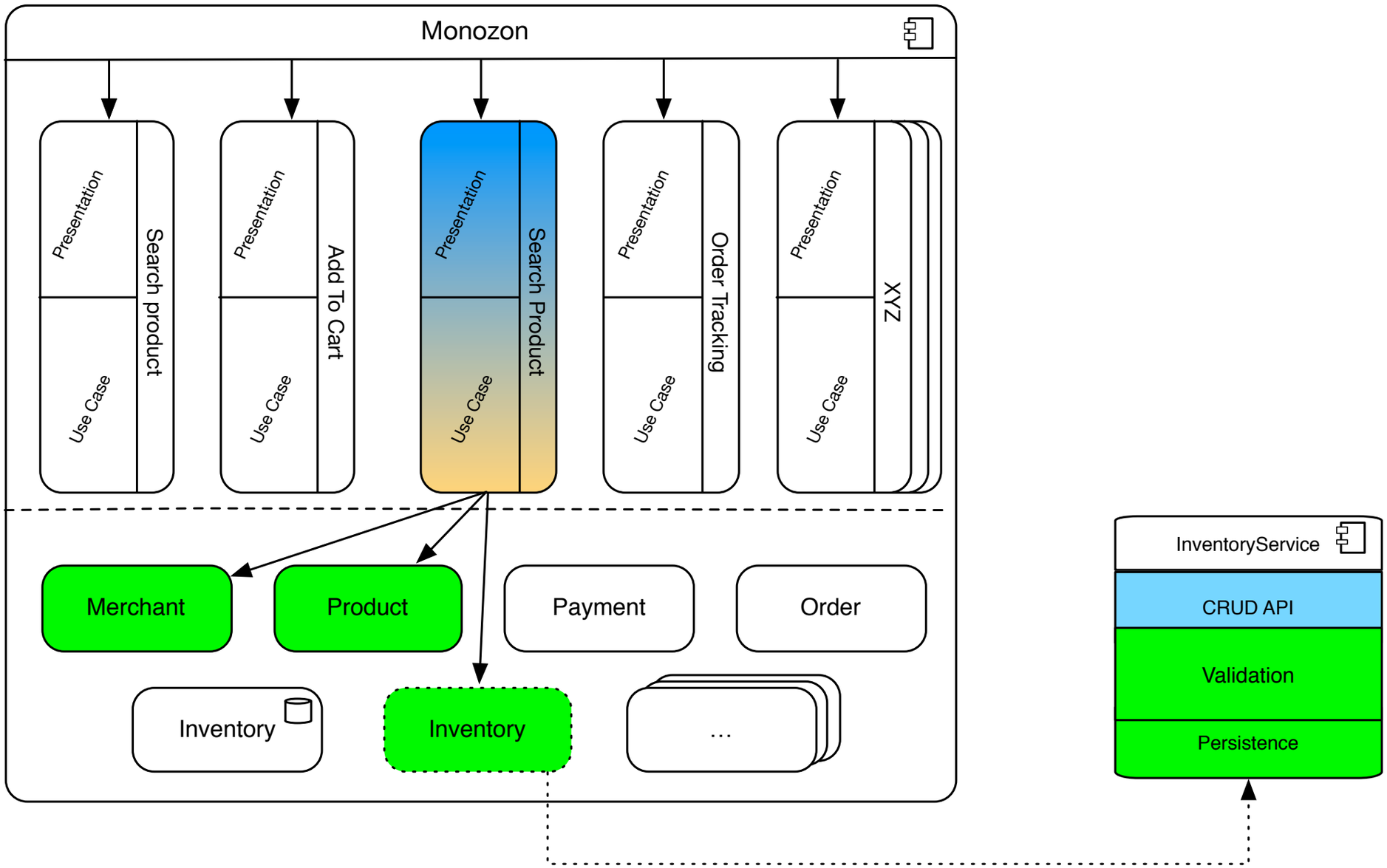
How much to cut out?

intertwined presentation and use case logic

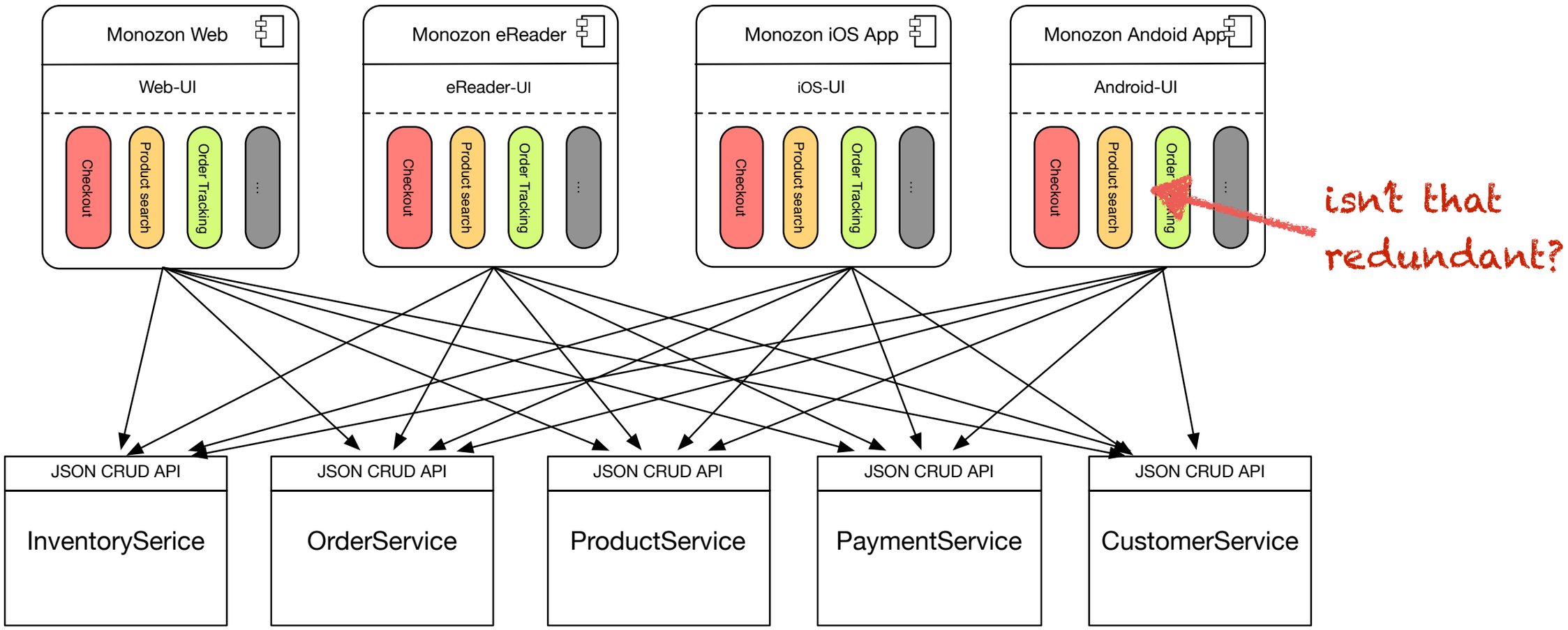
intertwined business entity and persistence logic



Reuse Business Entities?



Beware of multiple Clients

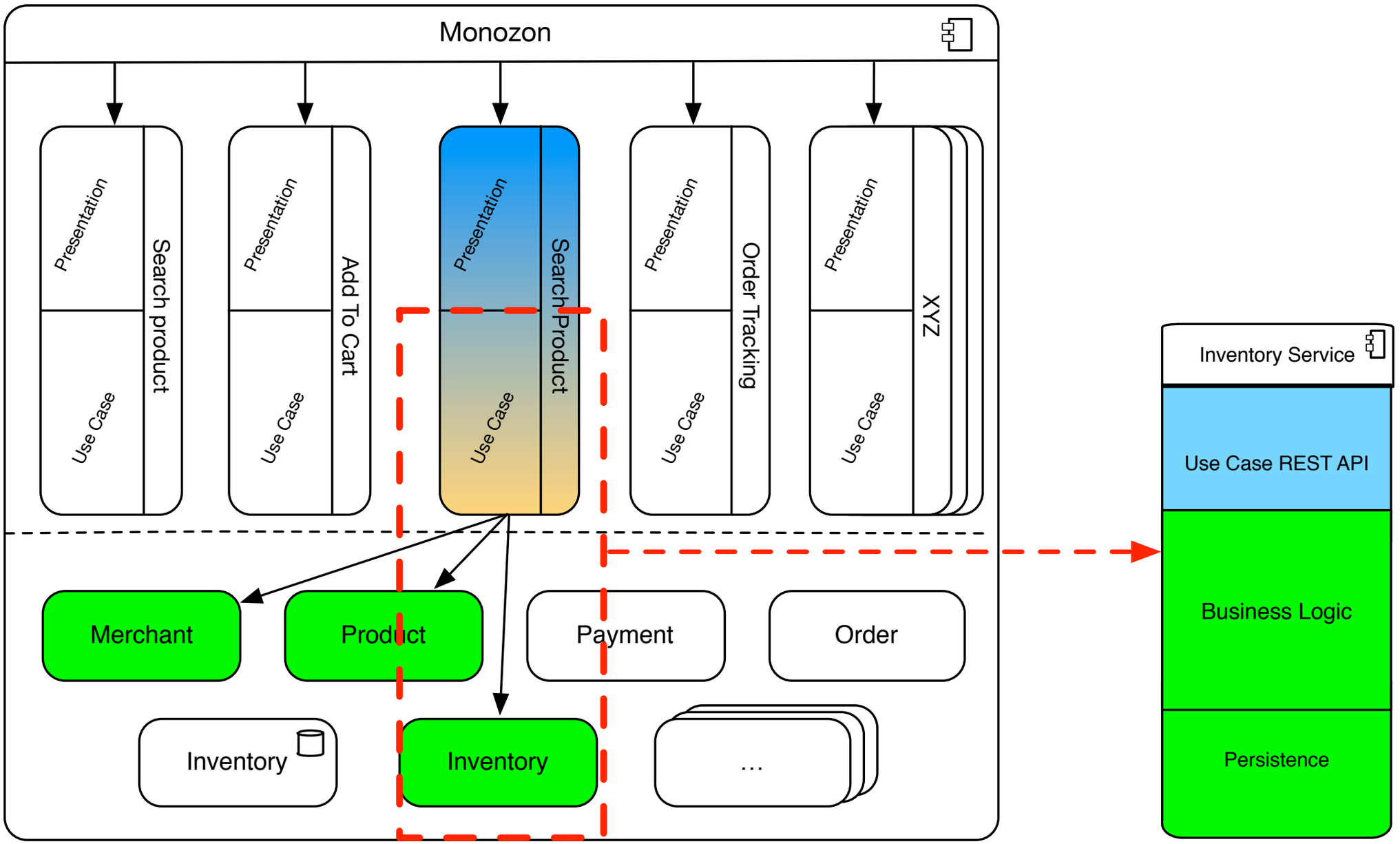


Downsides

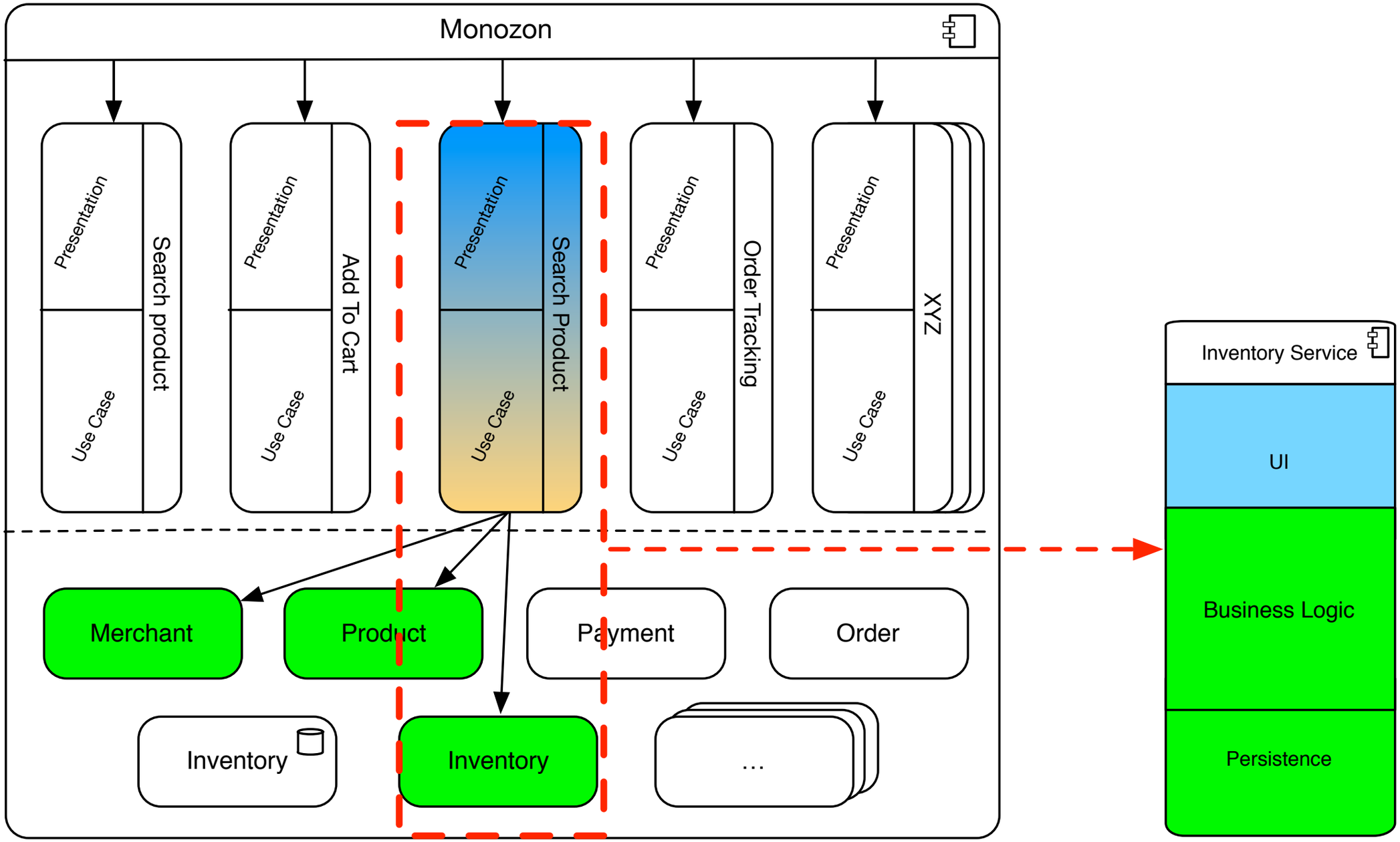
- CRUD APIs must match all needs
- only Use Case agnostic validation is possible
- every Clients must rebuild the Use Cases with by aggregating a lot of calls
- redundant and not reusable logic in all Clients

Aim for a higher cut that also influences your integration strategy

API Integration



Frontend Integration



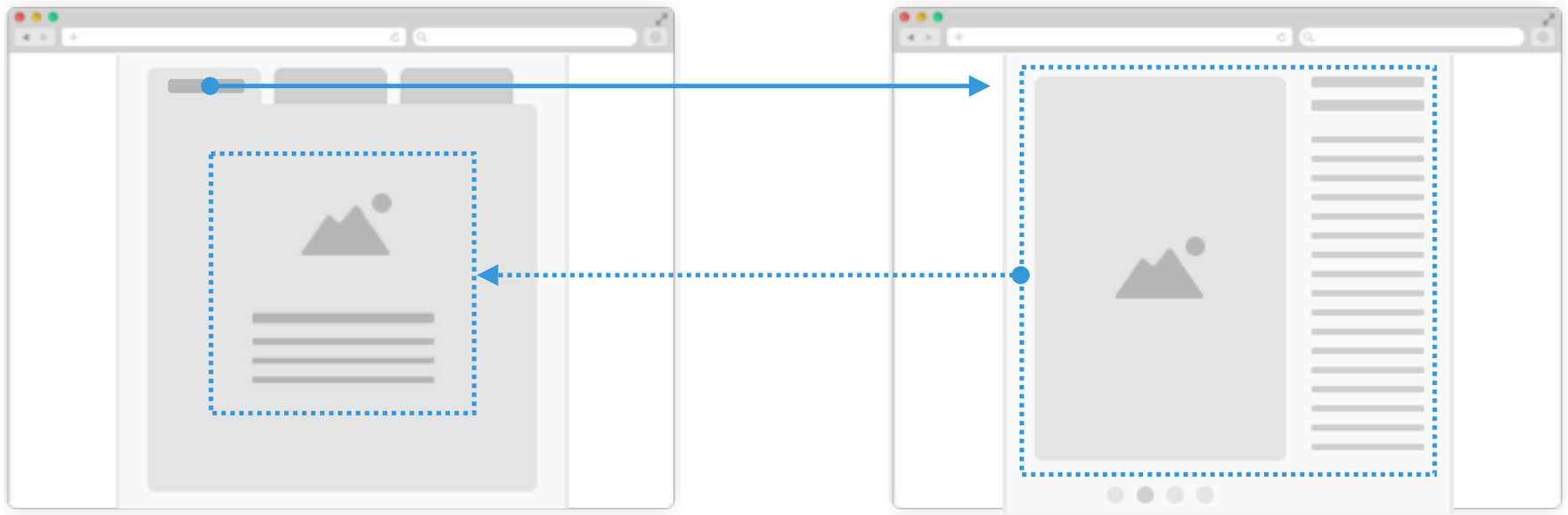
Frontend Integration

System 1

System 2

Simple **hyperlinks** can be used to navigate between systems.

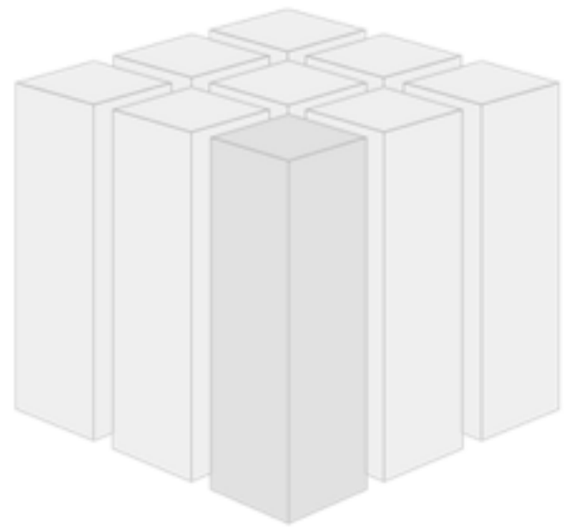
Frontend Integration



System 1

System 2

Hyperlinks can also support the
dynamic inclusion of content that is served by another application into the web interface of a self-contained system.



Self-Contained Systems

ASSEMBLING SOFTWARE FROM INDEPENDENT SYSTEMS

http://scs-architecture.org/

API Integration Review

Pro:

- simple „classic“ way
- no need to restructure backend and frontend teams
- could support wide variety of clients

Con:

- harder to extract complete business logic
- no real product teams
- cut between frontend and backend

Frontend Integration Review

Pro:

- nice way to extract complete logic
- systems are highly autonomous
- real product teams possible

Con:

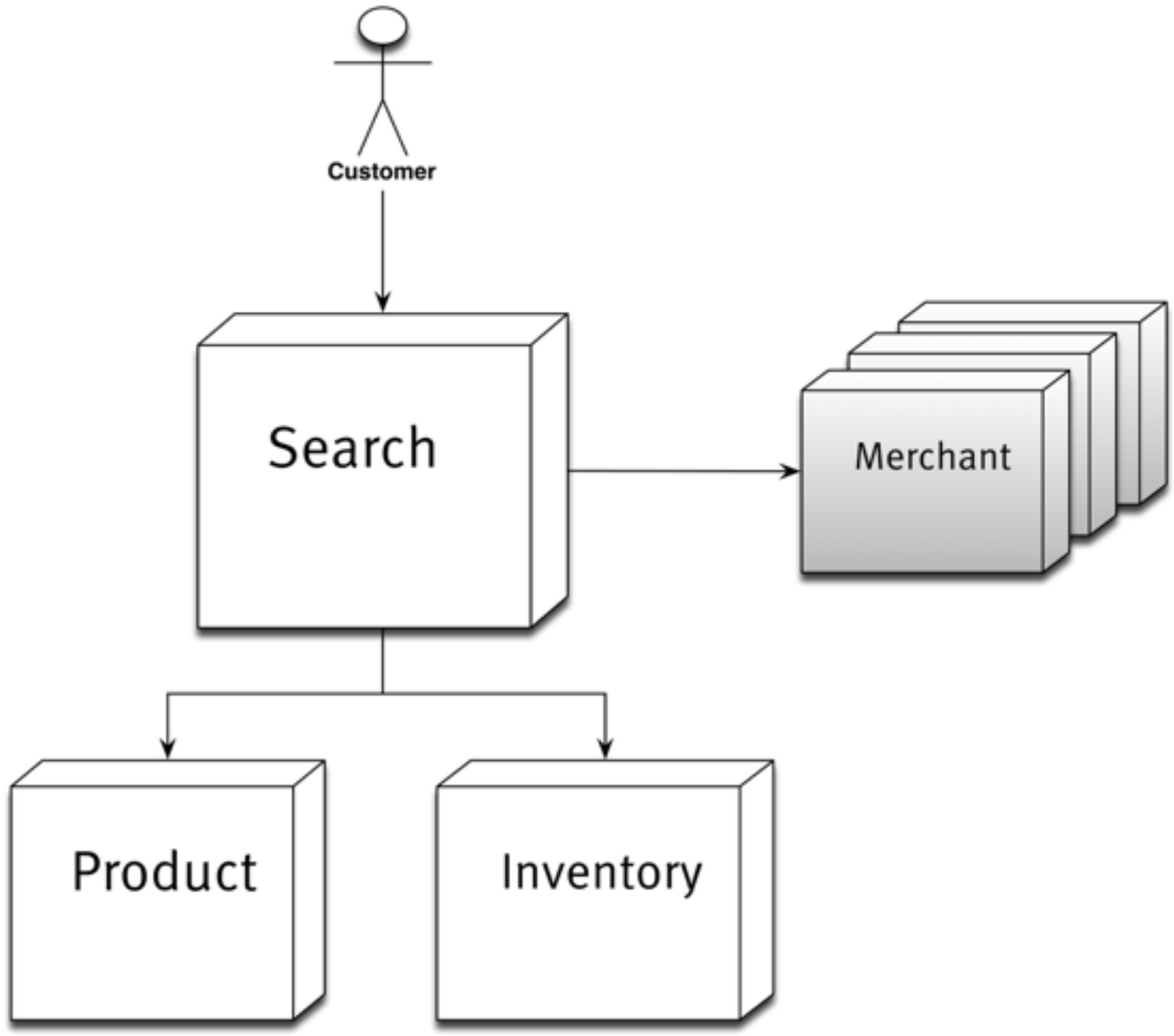
- not the stuff most devs are used to
- callbacks could be ugly
- reorganization of frontend and backend teams could be needed

Let’s recap

1. stabilize your monolith and enable distribution
2. establish and communicate your macro architecture
3. use you domain architecture to extract new services
4. prefer to „strangle“ over „big bangs“
5. always extract all business logic (even if it’s harder)
6. have a vision

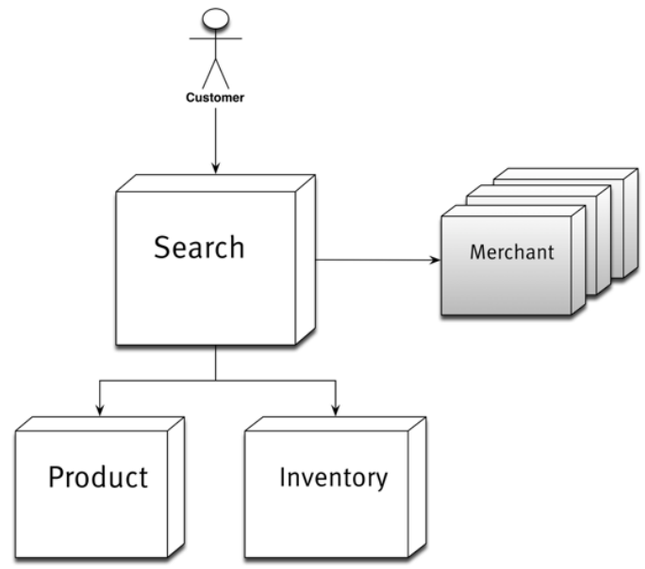
Assume we picked „api integration“ as a strategy

Time to connect boxes



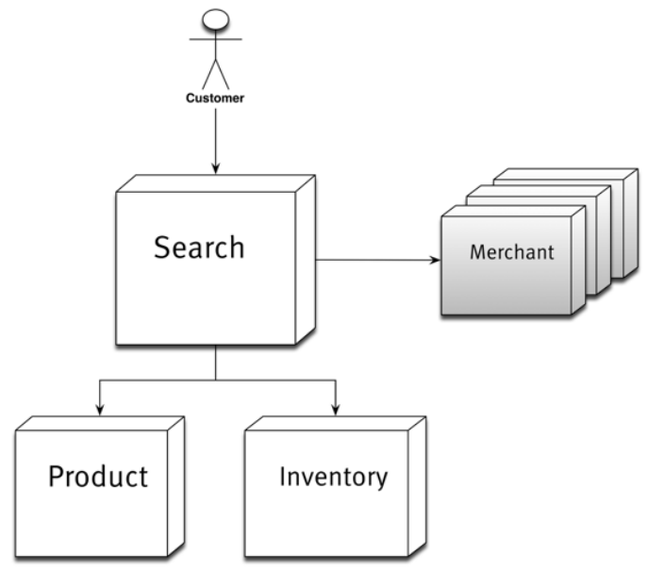
What has to be done?

- Retrieve internal products that match search criteria
- Retrieve external products from merchants that match search criteria



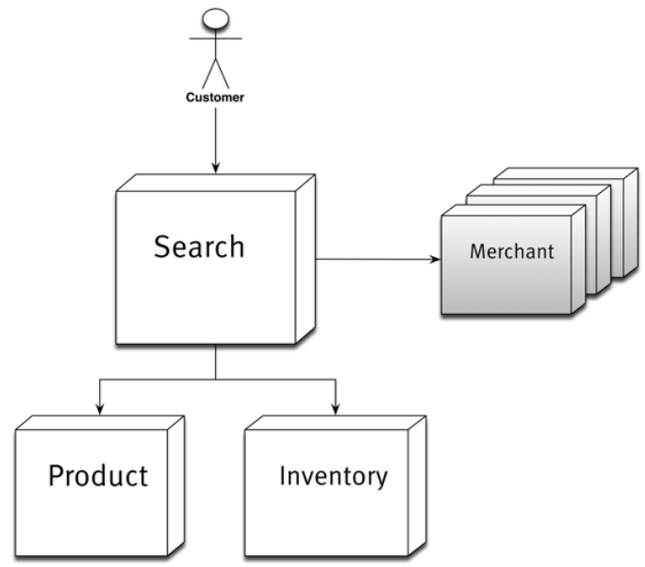
Internal Product Search

- Retrieve product ids from Search
- Retrieve product details from Product
- Retrieve number of matching products in stock from Inventory
- Combine all data into one format called SearchResult

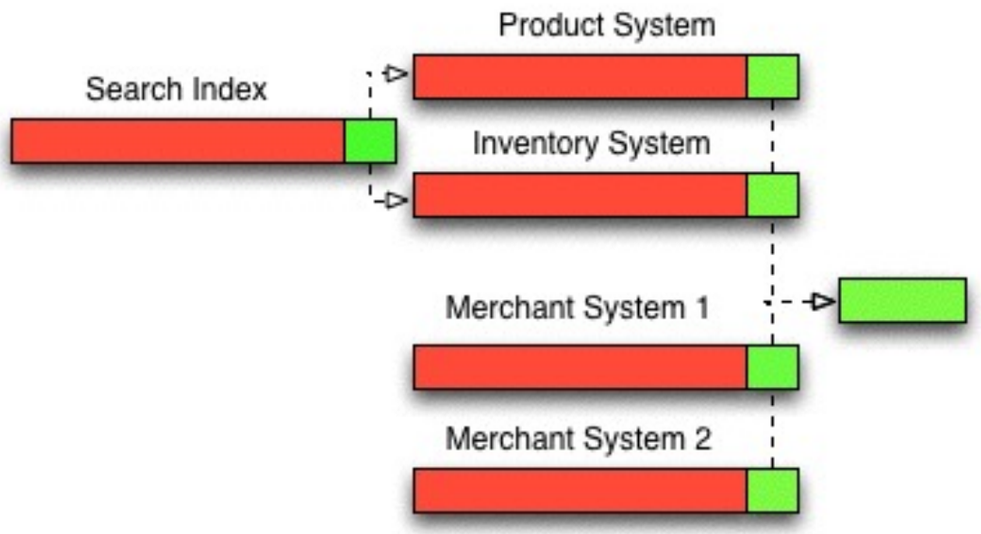


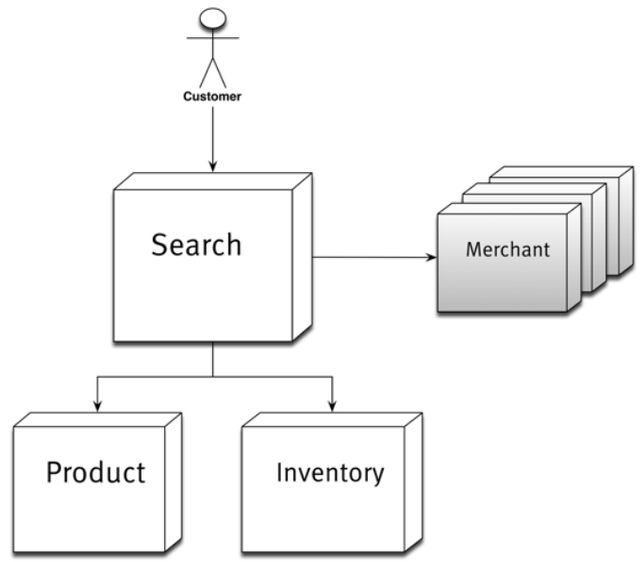
External Product Search

- Retrieve matching products from different merchants in different data formats
- Combine all data into one format called SearchResult

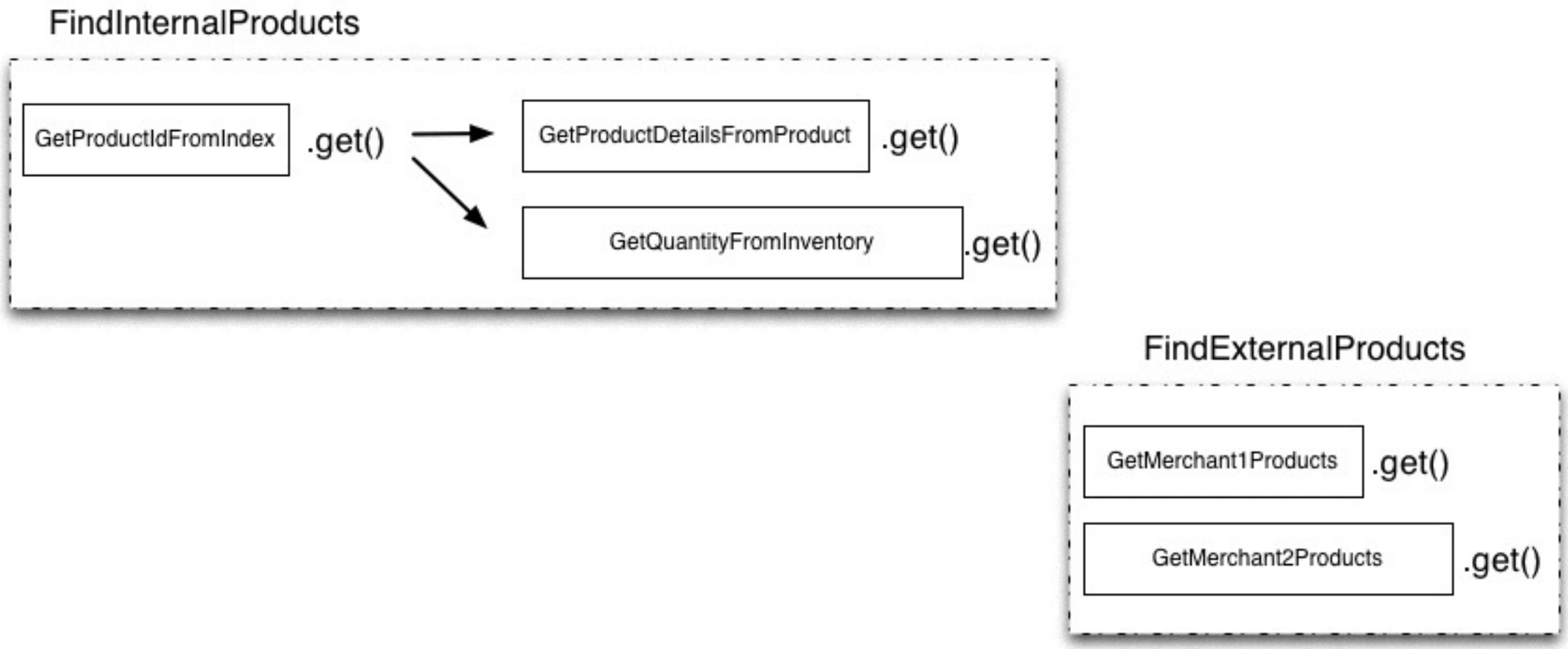


Sequential vs. Parallel

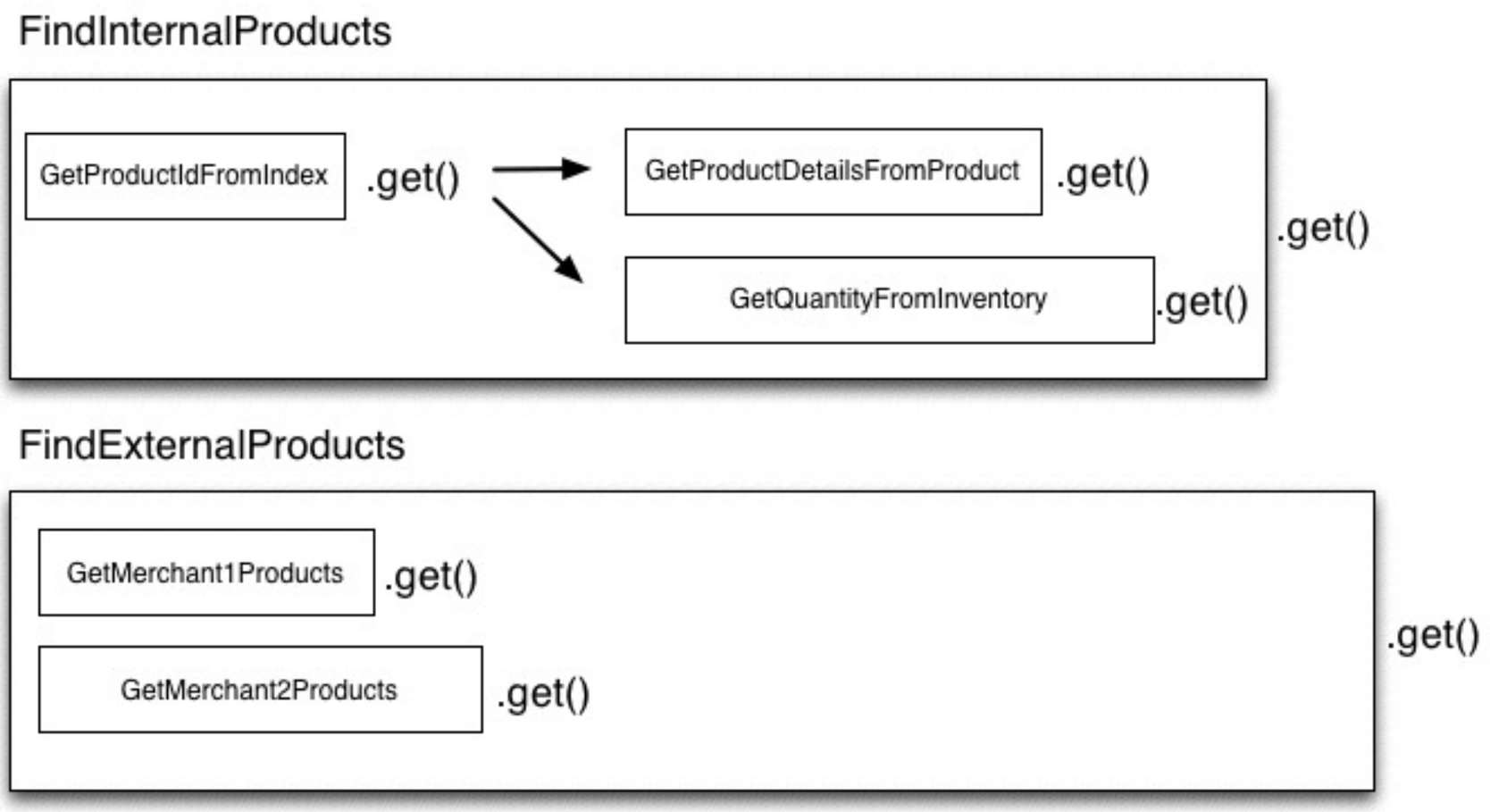




Future‹SearchResult›



Future<Future<SearchResult>



Internal products with Futures

```
Future<List<Long>> productIndexFuture =
        getProductIndexFuture(query);

List<Long> productIdList = productIndexFuture.get();

List<FuturePair> futurePairList = new ArrayList<>();
for (Long productId : productIdList) {
    Future<Product> productFuture = retrieveProductFromProductSystem(productId);
    Future<Long> quantityFuture = retrieveQuantityFromInventoryService(productId);
    futurePairList.add(new FuturePair(productFuture, quantityFuture));
}

List<SearchResult> searchResultList = new ArrayList<>();
for (FuturePair futurePair : futurePairList) {
    Product product = futurePair.getProductFuture().get();
    Long quantity = futurePair.getQuantityFuture().get();
    SearchResult searchResult = new SearchResult(product, quantity);
    searchResultList.add(searchResult);
}
```

External products with Futures

```
Future<List<Merchant1Product>> searchResultM1Future =
        new GetMerchant1ProductsCommand(query).queue();

Future<List<Merchant2Product>> searchResultM2Future =
        new GetMerchant2ProductsCommand(query).queue();
```

blocking!

```
List<Merchant1Product> searchResultM1 = searchResultM1Future.get();
List<Merchant2Product> searchResultM2 = searchResultM2Future.get();

List<SearchResult> externalSearchResults = new ArrayList<>();
for (Merchant1Product merchant1Product : searchResultM1) {
    externalSearchResults.add(toSearchResult(merchant1Product));
}

for (Merchant2Product merchant2Product : searchResultM2) {
    externalSearchResults.add(toSearchResult(merchant2Product));
}
```

Combine products with Futures

```
ExecutorService executorService = Executors.newFixedThreadPool(2);

Future<List<SearchResult>> internalResultFuture =
        executorService.submit(() -> findInternalProducts(query));
Future<List<SearchResult>> externalResultFuture =
        executorService.submit(() -> findExternalProducts(query));

List<SearchResult> result = internalResultFuture.get();
result.addAll(externalResultFuture.get());
```

blocking!

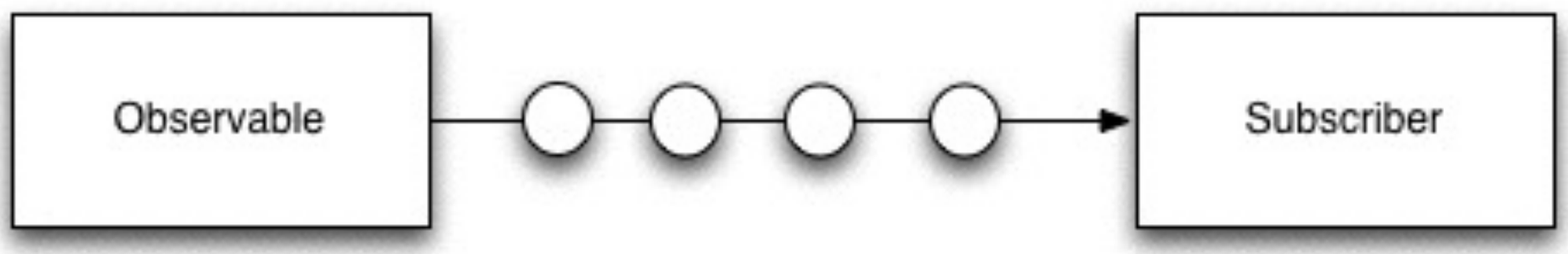
Time for RxJava

- Reactive Extensions for the JVM
- Asynchronous streams

- Elements of
  - Observable pattern
  - Iterator pattern
  - Functional programming

The Observable part

- An Observable emits items
- An Subscriber consumes these items



The Interator part

Iterable	Observable
pull	*push*
T next()	onNext(T)
throws Exception	onError(Exception)
returns;	onCompleted()

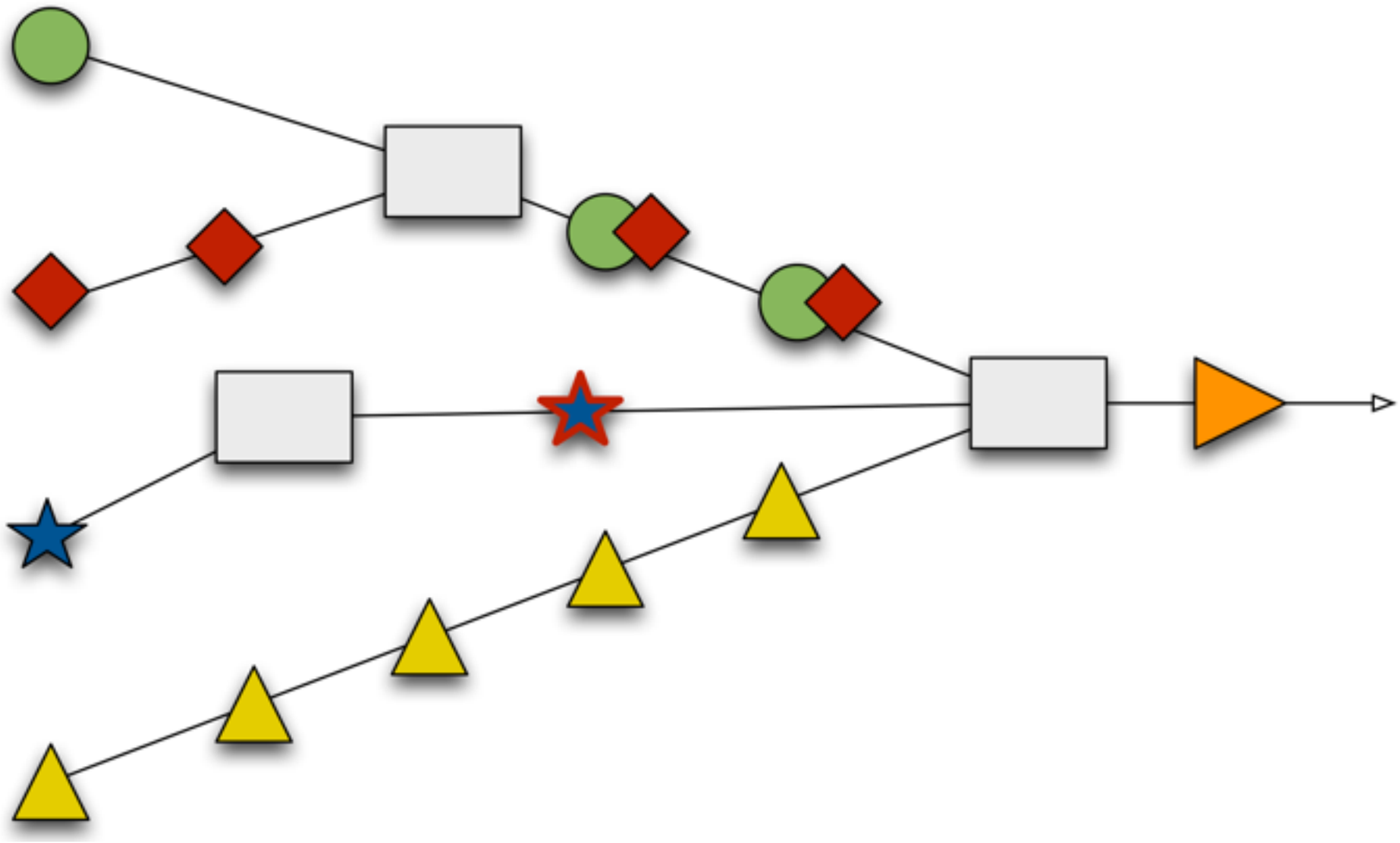
https://speakerdeck.com/benjchristensen/functional-reactive-programming-with-rxjava-javaone-2013

The functional part

- **Combining**: merge(), zip(), join(), ...
- **Filtering**: filter(), skip(), distinct(), ...
- **Aggregating**: reduce(), max(), sum(), ...
- **Transformation**: map(), flatMap(), ...
- **Conditional**: contains(), amb(), all(), ...


Everything is a stream!

RxJava in one picture



https://speakerdeck.com/benjchristensen/functional-reactive-programming-with-rxjava-javaone-2013

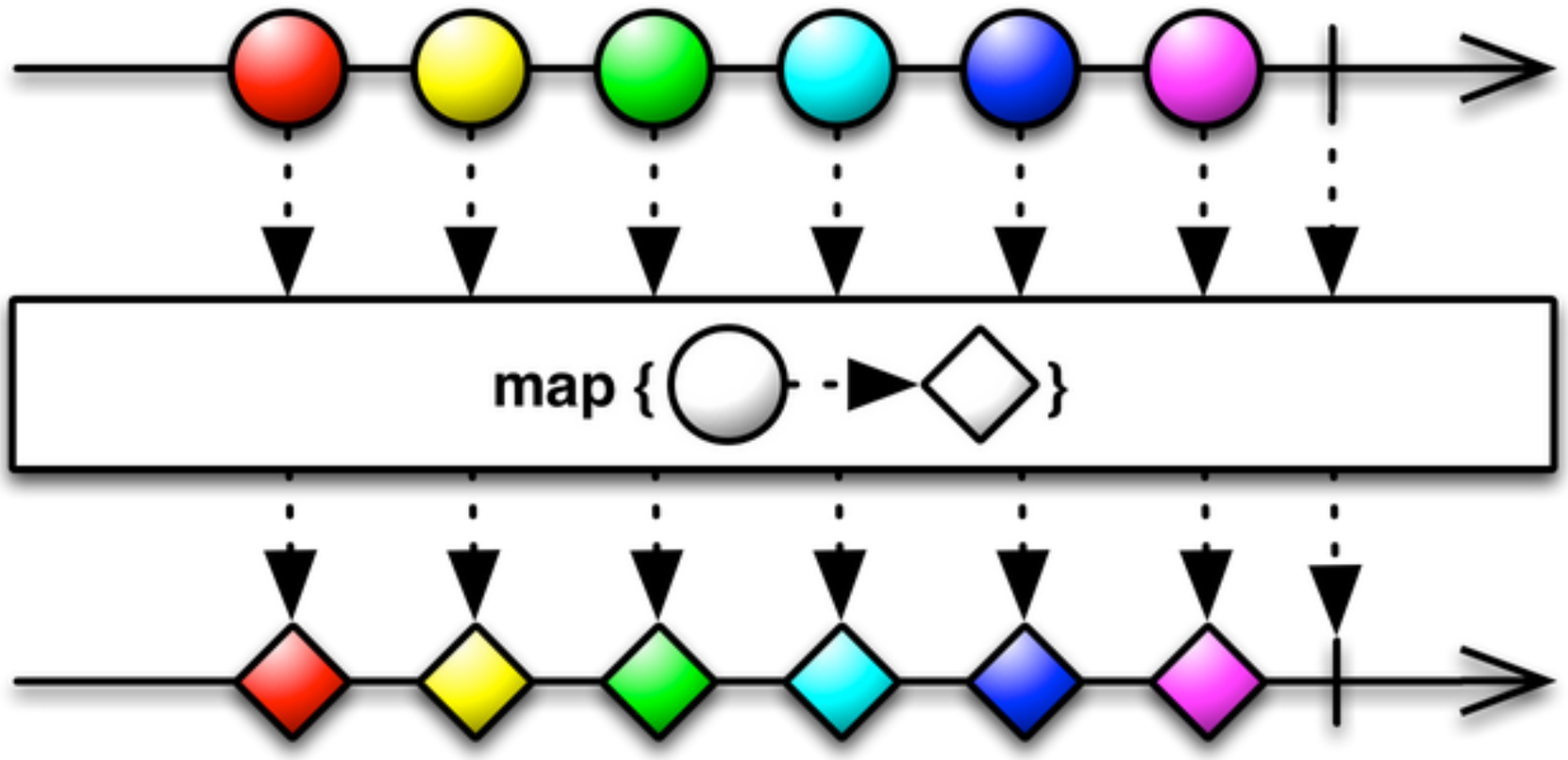
Creating Observables

```
Observable.just("Book A", "Book B", "Book C");
```

```
Observable.from(findProducts(query));
```

```
Observable.create(o -> {
    for (Merchant2Product product : findProducts(query)) {
        o.onNext(product);
    }
    o.onCompleted();
});
```

Transforming with map



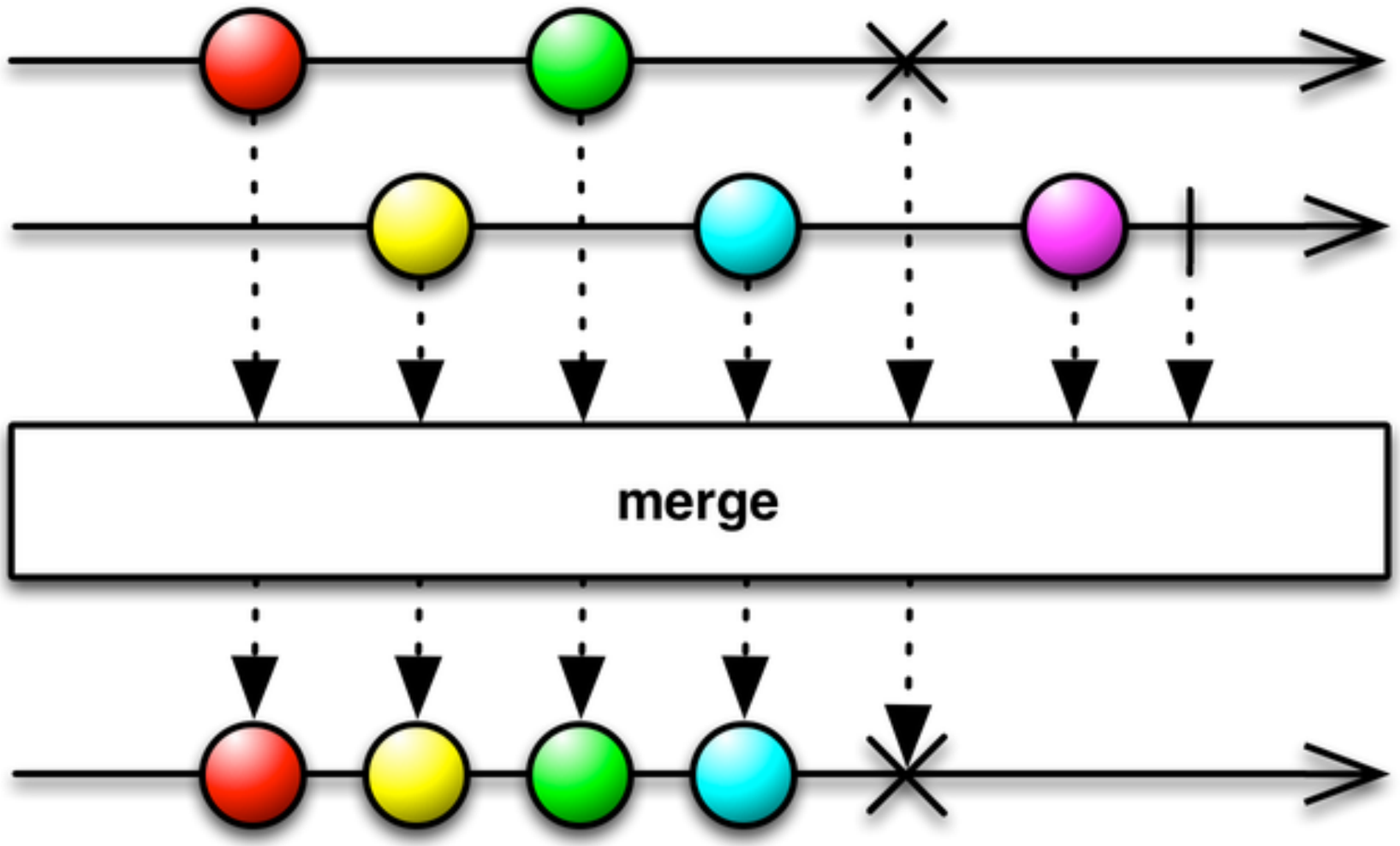
http://reactivex.io/RxJava/javadoc/rx/Observable.html

map in action

```
Observable<Merchant2Product> productM2Observable =
        Observable.from(findProducts(query));

Observable<SearchResult> searchResultM2Observable =
        productM2Observable.map(this::toSearchResult);
```

Combining with merge

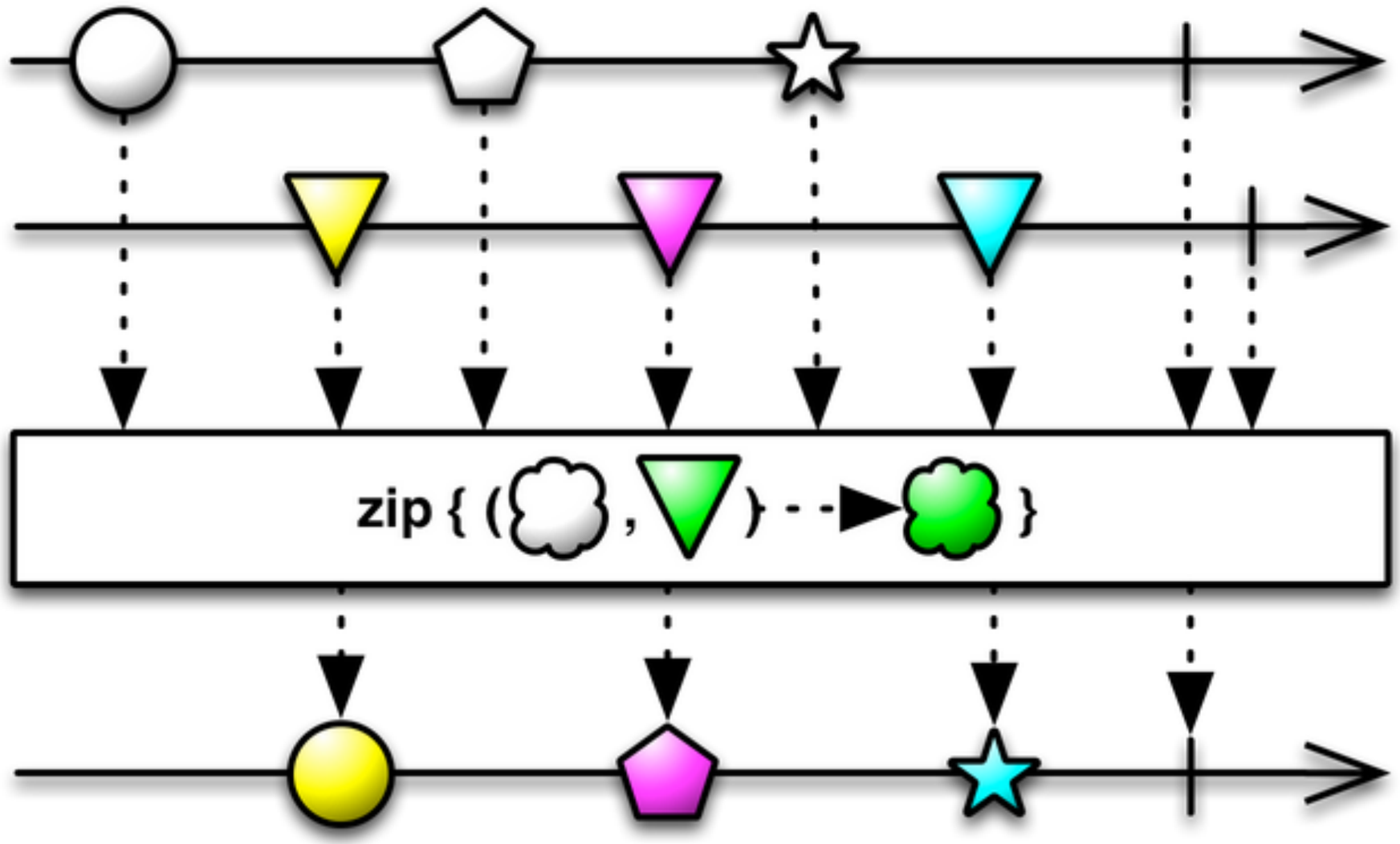


http://reactivex.io/RxJava/javadoc/rx/Observable.html

merge in action

```
Observable<SearchResult> mergedSearchResultObservable =
    searchResultM1Observable.mergeWith(searchResultM2Observable);
```

Combining streams with zip



http://reactivex.io/RxJava/javadoc/rx/Observable.html

zip in action

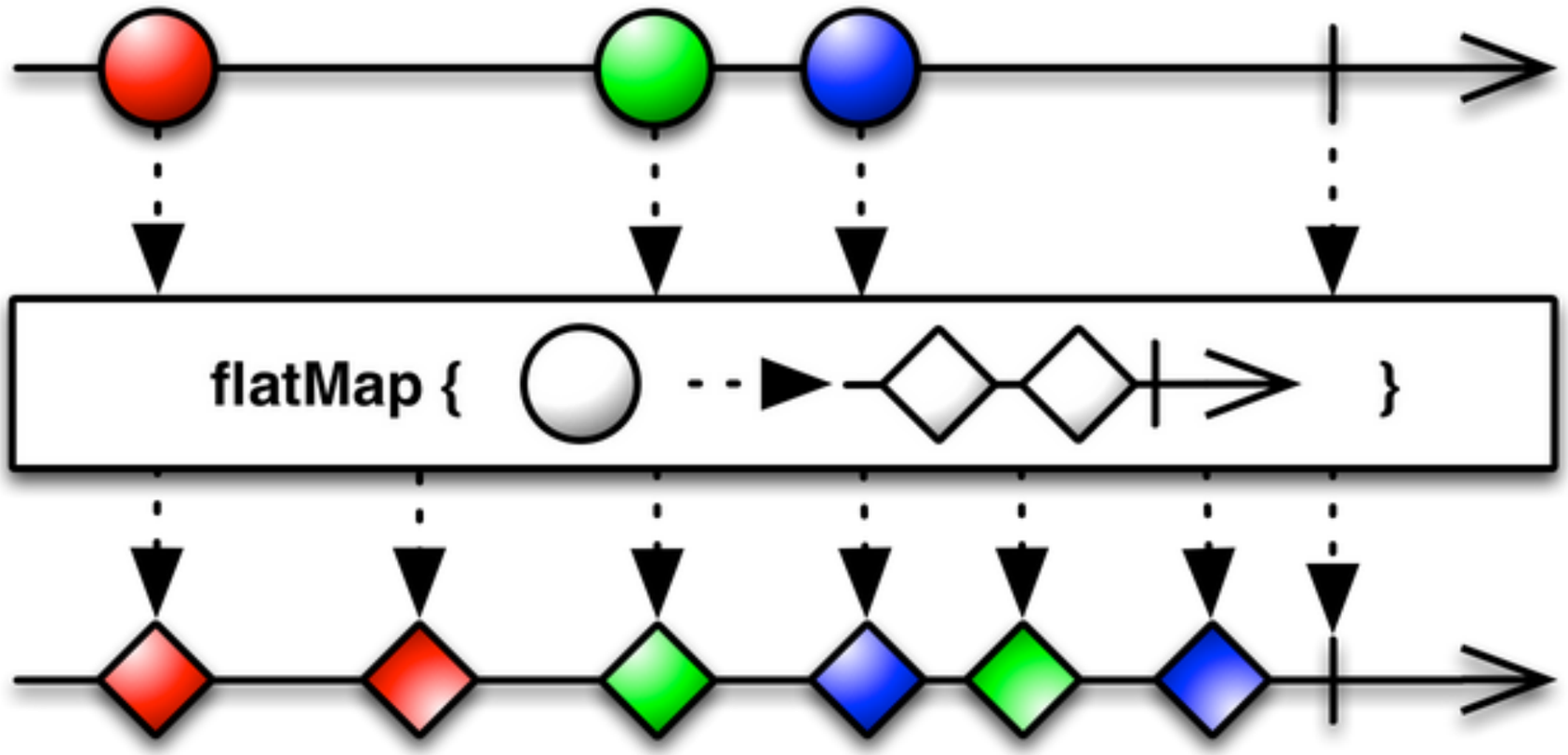
```
private Observable<SearchResult> productDetails(Long productId) {

    Observable<Product> productObservable =
            retrieveProductFromProductSystem(productId);

    Observable<Long> quantityObservable =
            retrieveQuantityFromInventoryService(productId);

    return Observable.zip(productObservable,
            quantityObservable, SearchResult::new);
}
```

Collecting details with flatMap



http://reactivex.io/RxJava/javadoc/rx/Observable.html

flatMap in action

```
public Observable<SearchResult> findInternalProducts(String query) {

    Observable<Long> productIndexObservable =
            getProductIndexObservable(query);

    return productIndexObservable
            .flatMap(this::productDetails);

}
```

Why not map?

```
public Observable<Observable<SearchResult>> findInternalProducts(String query) {

    Observable<Long> productIndexObservable =
            getProductIndexObservable(query);

    return productIndexObservable
            .map(this::productDetails);

}
```

Concurrency

- Observers run in the thread that runs the Observable

```
Observable<Merchant2Product> productM2Observable =
        Observable.from(findProducts(query))
                .subscribeOn(Schedulers.io());
```

Easier with Hystrix

```
public class GetMerchant2Products
        extends HystrixCommand<List<Merchant2Product>> {

    private final String query;

    public GetMerchant2Products(String query) {
        super(HystrixCommandGroupKey.Factory.asKey("merchant2"));
        this.query = query;
    }

    @Override
    public List<Merchant2Product> run() {
        return findProducts(query);
    }
```

Converting into a stream

```
Observable<List<Merchant2Product>> productM2ListObservable =
        new GetMerchant2Products(query).observe();

Observable<Merchant2Product> productM2Observable =
        productM2ListObservable.flatMap(Observable::from);
```

Synchronous result

```
@RequestMapping("/searchWithRx")
@ResponseBody
public List<SearchResult> findProducts(String query) {
    Observable<SearchResult> searchResults =
            searchService.findInternalProducts(query)
                    .mergeWith(searchService.findExternalProducts(query));

    Iterator<SearchResult> searchResultIterator
            = searchResults.toBlocking().getIterator();

    return Lists.newArrayList(searchResultIterator);
}
```

Asynchronous result

```
@RequestMapping("/searchWithRxDeferred")
@ResponseBody
public DeferredResult<List<SearchResult>> findProductsDeferred(String query) {
    Observable<SearchResult> searchResults =
            searchService.findInternalProducts(query)
                    .mergeWith(searchService.findExternalProducts(query));

    DeferredResult<List<SearchResult>> deferredResult = new DeferredResult<>();
    searchResults.subscribe(new SearchResultSubscriber(deferredResult));

    return deferredResult;
}
```

Subscriber

```
public class SearchResultSubscriber extends Subscriber<SearchResult> {

    final private List<SearchResult> searchResultList = new ArrayList<>();

    final private DeferredResult<List<SearchResult>> deferredResult;

    @Override
    public void onNext(SearchResult searchResult) {
        searchResultList.add(searchResult);
    }

    @Override
    public void onCompleted() {
        deferredResult.setResult(searchResultList);
    }
```

Error handling

- onError works like onCompleted, just with an error ;)
- UncheckedExceptions end up in OnError
- CheckedException have to be catched and rethrown as RuntimeExceptions
- Rx provides helpers to throw them

Reasons for RxJava

- Can be used with Java 6 and later versions
- Already part of Hystrix
- Lots of operators

Alternatives

- ListenableFuture (Google Guava)
- Reactive Streams (Akka-Streams, Reactor, Vert.x, etc.)
- CompletableFuture (Java 8)
- Flow (Java 9)

Summary

- Use Hystrix to stabilize your system!
- Use RxJava to increase the amount of parallel processes in an easy way!
- Introduce Microservices to get control over your system again!
- Have fun :)

Thank you!