AgileDev

v. 0.1.9-SNAPSHOT.002

Guidelines

2015-06-18

Table of Contents

1 Software Designing Principles

Software Design Principles

Software design principles represent a set of guidelines at the highest level. The design principles define the overall generic shape and structure of software applications. A level lower are the (architectural) patterns which focus on a specific problem.

There are 4 primary symptoms that indicate that the design is rotten:

- *Rigidity* - Rigidity is the tendency for software to be difficult to change even in case of simple changes. Every change causes a cascade of subsequent changes in dependent modules.
- *Fragility* - Closely related to rigidity is fragility. Fragility is the tendency of the software to break (in many places) every time it is changed. Often the breakage occur in places that have no conceptual relationship with the area that was changed.
- *Immobility* - Immobility is the inability to reuse software from other projects or from parts of the same project.
- *Viscosity* - Viscosity comes in 2 forms: viscosity of the design, and viscosity of the environment. When changes in which the design is preserved are harder to employ than the hacks, then the viscosity of the design is high. Viscosity of environment comes about when the development environment is slow and inefficient. In this case developers are tempted to make changes that prevent a lot of changes, which are usually not optimal from a design point.

These four symptoms are the signs of poor architecture. Within this part we will focus on software design principles by which we can prevent this.

2 Behavior Driven Development

Behavior Driven Development

Behavior Driven Development is a software development process that emerged from Test Driven Development. Behavior Driven Development combines the general techniques and principles of TDD with ideas from Domain Driven Design. TDD states that for each unit of software (class in case of OO), a software developer must:

1. Define a test set for the unit *first*.
2. Then implement the unit.
3. Verify that the implementation of the unit makes the tests succeed.

Extension to this is that within BDD the tests should be defined in terms that has business value; e.g. the tests should be written in a form that is using business terminology (defined within the ubiquitous language).

On top of that BDD defines how the desired behavior should be specified. Business analysts and developers should work together and specify the desired behavior in the form of user stories that have the following structure:

```
Title: Explicit and clear title

Narrative: Short introduction that specifies
    * Who is the driver or primary stakeholder of the story, (business actor).
    * What effect the story would have.
    * What benefits the stakeholder will derive from this effect.

Scenarios: Description of each specific case according to the following structure

    * Given: Specifies the initial situation. This may consist of
             single clause, or several (combined by AND).
    * When : States which event triggers the start of the scenario.
    * Then : States the expected outcome in one or more clauses.
```

2.1 Related Patterns

- Test Driven Development - Behavior Driven Development emerged from Test Driven Development.
- Domain Driven Design - Behavior Driven Development borrows the concept of the ubiquitous language from DDD to define the tests in business terminology.

2.2 See also

- Cucumber - BDD testing framework for several languages, (Ruby, Java, …).
- An Introduction to BDD Test Automation with Serenity and JUnit

3 Defensive Programming

Defensive Programming

Defensive Programming is the practice of anticipating all possible ways that an end user could misuse an software system, and designing the system in such way that this is impossible, or to minimise the negative consequences. Goal of defensive programming is:

- Reducing the number of software bugs.
- Making software understandable; the source code should be readable and understandable and verified during code audit.
- Making the software behave in a predictable manner despite unexpected inputs or user actions.

Overly defensive programming could introduce code to prevent errors that can’t happen, but needs to be executed on runtime and to be maintained by the developers. There is also the risk that the code catches or prevents too many exceptions. In those cases, the error would be suppressed and go unnoticed while the result would still be wrong.

Some defensive programming techniques are:

- Code reuse - Existing code is tested and known to work, reusing it may reduce the change of bugs being introduced.
- Legacy problems - Before reusing old libraries, APIs, and so forth, it must be validated whether the old work is valid for reuse. The library might for example have a much lower quality than the newly designed system.
- Low tolerance against ‘potential’ bugs - Assume that code constructs that appear to be problem prone (for example reported by a source code analyzer) are bugs and potentially security flaws.
- Encrypt/authenticate all data transmitted over networks. Do not implement your own encryption scheme, but use a proven one instead.
- Design by Contract - Use Design by Contract methodology to ensure that provided data (and the state of the program as a whole) is verified.
- Error codes - Prefer exceptions to return error codes, see error handling for more details.

3.1 See also:

- PMD - ‘Free’ source code analyzer to detect potential bugs.

4 Design by Contract

Design By Contract

Design by Contract was first explored by Bertrant Meyer. He has invented a language named Eiffel in which contracts are explicitly stated for each method, and checked at each invocation. The contracts define preconditions, postconditions and invariants.

- Preconditions are certain expectations that need to be guaranteed before a client is allowed to perform certain functionality.
- Postconditions are guarantees on exit of certain functionality.
- An invariant is a certain constraint that must be true during the whole lifecycle of the entity.

Design by Contract advocates writing the conditions and invariants first. Constraints can be written by annotations and enforced by a test suite.

The contract will be applied on the method and will normally contain the following pieces of information:

- Acceptable and unacceptable input values/types and their meanings.
- Return values/types and their meaning.
- Exceptions that can occur during execution and their meaning.
- Side effects (for example notifications that are published, etc).
- Preconditions
- Postconditions
- Invariants

In rare cases performance guarantees are also part of the contract.

When applying design by contract a client should not try to verify that the contract conditions are satisfied. Instead the service will throw an exception and the client should be able to cope with that exception.

Design by contract will facilitate code reuse since the contract documents each piece of the code fully; e.g. the contract for a method can be regarded as a form of software documentation for the behavior of that module.

4.1 Related Patterns

- Defensive Programming - In case of multi-channel client server or distributed computing the defensive design approach should be taken, meaning that a server component tests (before or while processing a client’s request) that all relevant preconditions hold true.
- Behavior Driven Development - The enforcing of constraints can be applied by following the Behavior Driven Development methodology.

TODO: http://en.wikipedia.org/wiki/Design_by_contract

5 Intention revealing interfaces

Intention Revealing Interfaces.

If a developer must consider the implementation of a component in order to use it, the value of ancapsulation is lost. If someone other that the original developer must infer the purpose of an object or operation based on its implementation, that new developer may infer a purpose that the operation or class fulfills only by change. If that was not the intent, the code may work for that moment, but the conceptual basis of the design will have been corrupted, and the 2 developers will be working at cross purpose.

Therefore, name classes and operations to describe their effect and purpose without reference to the means by which they do what they promise. This relieves the client developer of the need to understand the internals. These names should conform to the ubiquitous language so that team members can quickly infer their meaning. Write a test for a behavior before creating it, to force your thinking into client developer mode.

6 Reactive System

Reactive System

Systems built as Reactive Systems are more flexible, loosely-coupled and scalable. This makes them easier to develop and amenable to change. They are significantly more tolerant of failure and when failure does occur they meet it with elegance rather than disaster. Reactive Systems are highly responsive, giving users effective interactive feedback.

6.1 What are reactive systems

- *Responsive*: The system responds in a timely manner if at all possible. Responsiveness is the cornerstone of usability and utility, but more than that, responsiveness means that problems may be detected quickly and dealt with effectively. Responsive systems focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service. This consistent behaviour in turn simplifies error handling, builds end user confidence, and encourages further interaction.
- *Resilient*: The system stays responsive in the face of failure. This applies not only to highly-available, mission critical systems — any system that is not resilient will be unresponsive after a failure. Resilience is achieved by replication, containment, isolation and delegation. Failures are contained within each component, isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole. Recovery of each component is delegated to another (external) component and high-availability is ensured by replication where necessary. The client of a component is not burdened with handling its failures.
- *Elastic*: The system stays responsive under varying workload. Reactive Systems can react to changes in the input rate by increasing or decreasing the resources allocated to service these inputs. This implies designs that have no contention points or central bottlenecks, resulting in the ability to shard or replicate components and distribute inputs among them. Reactive Systems support predictive, as well as Reactive, scaling algorithms by providing relevant live performance measures. They achieve elasticity in a cost-effective way on commodity hardware and software platforms.
- *Message Driven*: Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages. Employing explicit message-passing enables load management, elasticity, and flow control by shaping and monitoring the message queues in the system and applying back-pressure when necessary. Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host. Non-blocking communication allows recipients to only consume resources while active, leading to less system overhead.

Large systems are composed of smaller ones and therefore depend on the Reactive properties of their constituents. This means that Reactive Systems apply design principles so these properties apply at all levels of scale, making them composable.

7 Side effect free functions

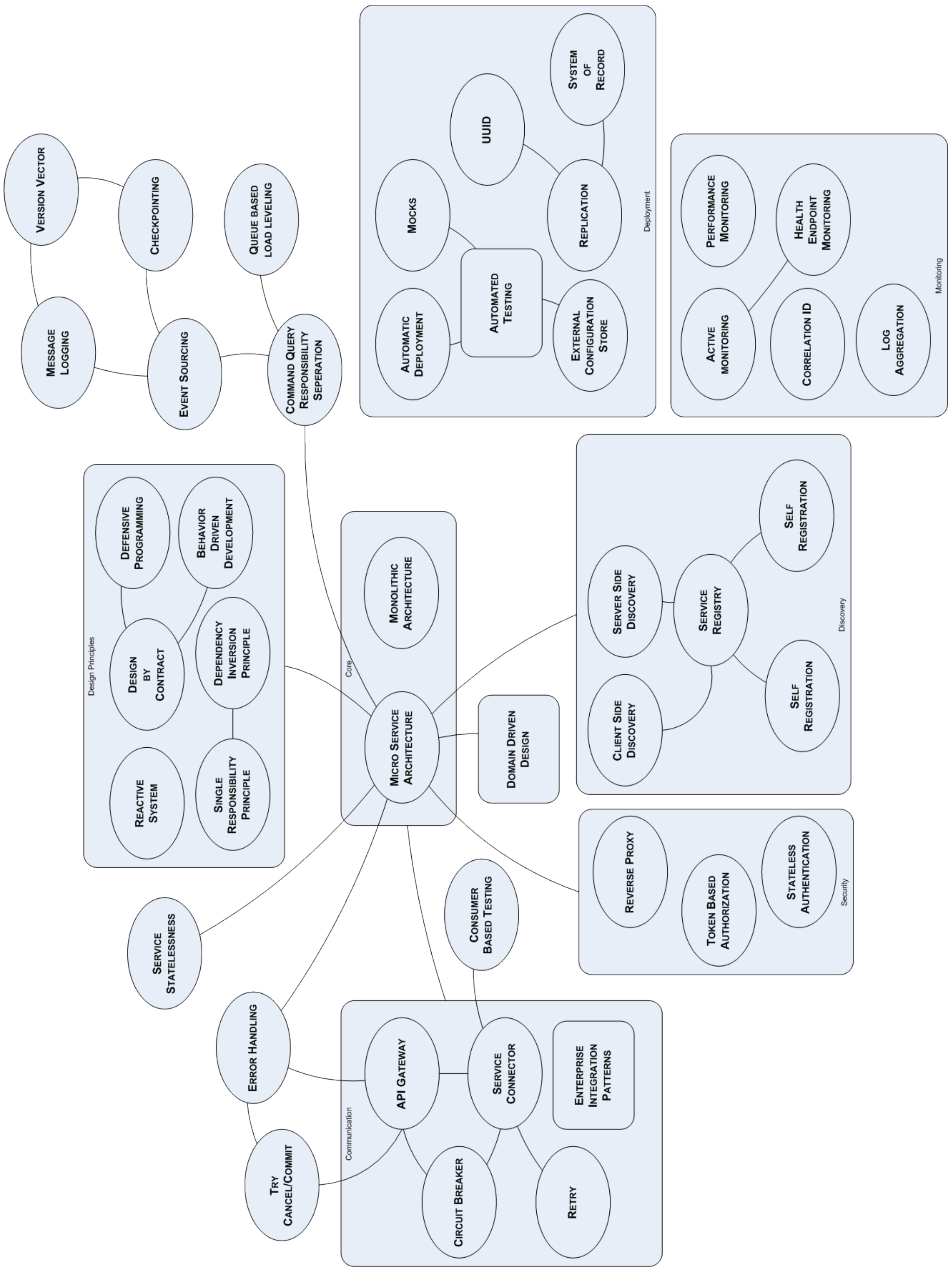
Side effect free functions.

Interactions of multiple rules or compositions of calculations become extremely difficult to predict. The developer calling an operation must understand its implementation and the implementation of all its delegations in order to anticipate the result. The usefulness of any abstraction of interfaces is limited if the developers are forced to pierce the veil.

Therefore, place as much as possible of the program into functions/operations that return results with no observable side effects. Strictly segregate method which result in modifications to observable state into very simple operations that do not return domain information. Further control side effects by moving complex logic into `Value Objects` when a concept fitting the responsibility present itself.

8 Patterns

Patterns

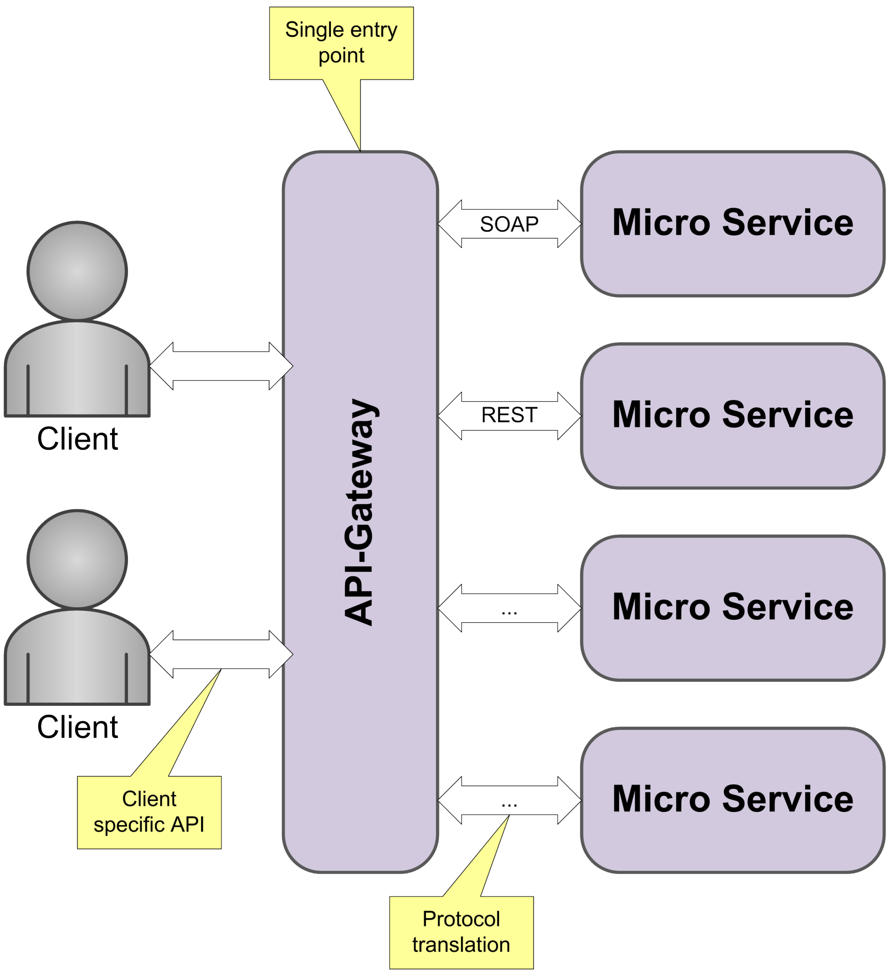

Version Vector
Checkpointing
Message Logging
Event Sourcing
Queue Based Load Leveling
Command Query Responsibility Seperation
Deployment
Mocks
UUID
System of Record
Replication
Automated Testing
Automatic Deployment
External Configuration Store
Monitoring
Performance Monitoring
Health Endpoint Monitoring
Active Monitoring
Correlation ID
Log Aggregation
Design Principles
Defensive Programming
Behavior Driven Development
Design by Contract
Dependency Inversion Principle
Reactive System
Single Responsibility Principle
Core
Monolithic Architecture
Micro Service Architecture
Domain Driven Design
Discovery
Server Side Discovery
Service Registry
Self Registration
Client Side Discovery
Self Registration
Security
Reverse Proxy
Token Based Authorization
Stateless Authentication
Service Statelessness
Consumer Based Testing
Error Handling
Communication
API Gateway
Service Connector
Enterprise Integration Patterns
Try Cancel/Commit
Circuit Breaker
Retry

9 API gateway

API Gateway

Your system must expose to clients functionality that they can use. However the granularity of the APIs provided by the microservices is often different from what the client needs. Microservice APIs typically provide fine-grained APIs, which means that clients need to interact with multiple services. Also the number of service instances and their locations (hoss, port, …) changes dynamically, and the partitioning of the services can change over time. All this should be hidden from the clients.

To solve this we expose an API gateway that is the single entry point for all clients.



The API gateway handles the requests in the following 2 ways:

1. Requests are simply proxied/routed to the appropriate service.
2. Requests are fanning out to multiple services

Rather than exposing a one size fits all style API, the API gateway can expose a different API for each client. The API Gateway might also verify that the client is authorized to perform the request.

Related Patterns

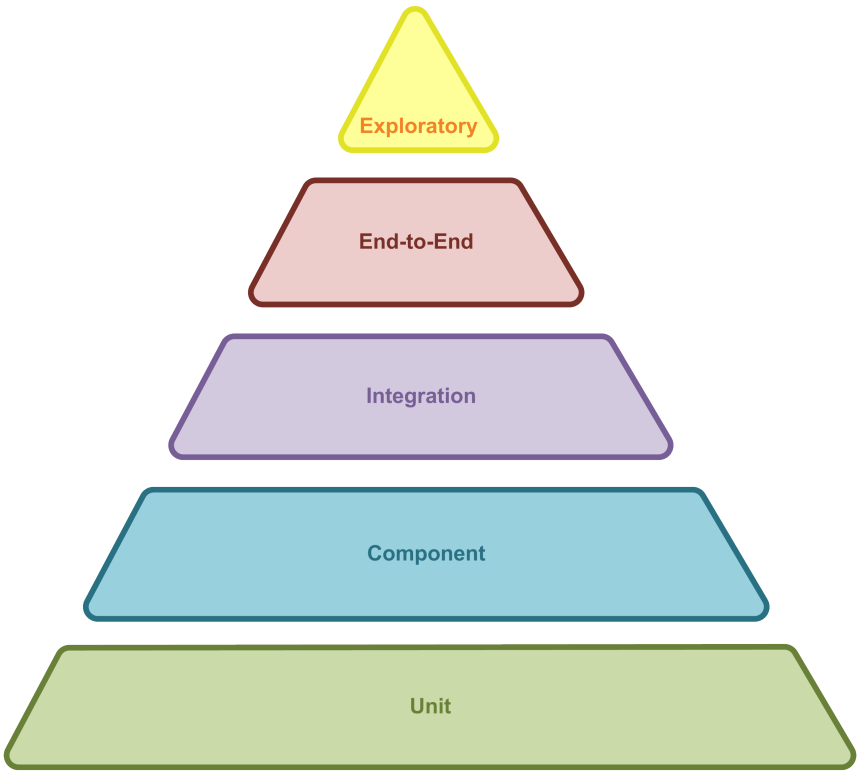
- Service Connector - Provide high level interface that hides implementation details regarding communication, thereby making the use of the microservice easier.

- Try-Cancel/Commit - By this pattern we can realize distributed transactions for microservices.

10 Automated testing

Automated testing

TODO



The testing pyramid essentially points out that you should have many more low level unit tests than high level end-to-end tests running. The layers we can distinct are:

- Exploratory -
- End-to-End
- Integration - By integration tests the interaction between two or more services is explicitly tested.
- Component -
- Unit - In this context, a 'unit' is often a function or a method of a class instance. See unit testing for more details.

10.1 Test Data

One of the harder challenges within automated testing is generating valuable test data. Hard coding assumptions about data availability can be a fragile approach as there are no guarantees that the data continue to exist. Furthermore, some tests may require data consistency across multiple services.

One of the robust strategies is to create the test data during the test run, as you guarantee the data exists before using it. This requires however that each service allows creating new resources, which isn't always the case. A solution for this could be to expose in the test environments test-only endpoints to facilitate test data creation. These end-points are of course not exposed in the production environment.

An alternative strategy is to have each service publish a cohesive set of test data that is guaranteed to be stable.

10.2 See also

- [Consumer-based testing] - To verify the integration between multiple services we could rely on consumer based testing.

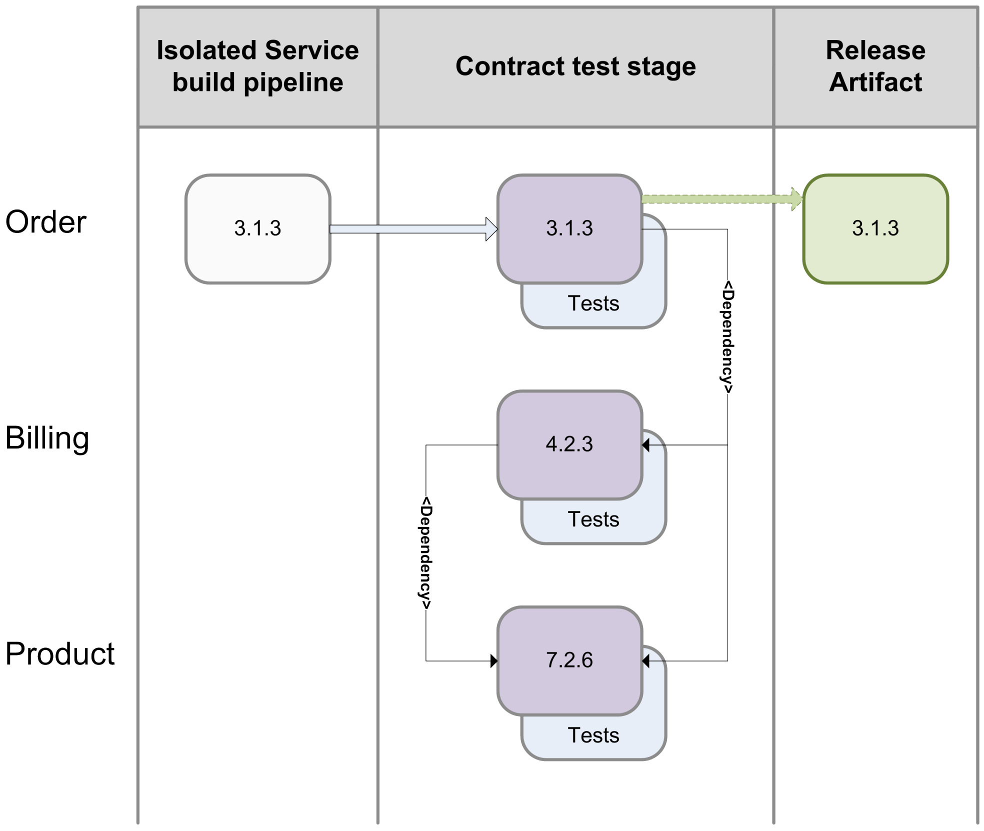
11 Consumer-based testing

Consumer-based testing

Consumer based testing is counter intuitive as it relies on the consumer writing tests for the producer. When writing contract tests, a consumer writes a test against a service it uses to confirm that the service contract satisfies the consumer needs.

By doing this we enable a neat trick in the deployment pipeline. After a service passes its internal build and QA process, all services and consumers go through a unified integration test stage. It could be triggered by the consumer changing tests, or the producer committing changes to the service. During this stage each consumer runs its tests against the new version of the changed service. Any failure prevents the new version from progressing in the pipeline.

Now for example, if we would have a `Order` Service that depends on the `Product` and `Billing` service, a new build of `Order` would trigger the execution of its consumer based tests against the latest version of `Product` and `Billing` service to pass the integration test stage.



Triggering only tests that are associated with a particular change can get tricky, however you can go a long way by simply running all contract tests each time a new service is deployed to the integration test stage within the pipeline.

This would mean that the consumer based tests of billing would be included (gray arrow), which aren't relevant to the change that was introduced.

Assuming that all the tests pass, we have a set of services that have been proven to work together. We can record the set of them working together by creating a Deployable Artifact Set (DAS). This DAS can become the single deployable artifact for higher stages within the deployment pipeline, or it can become a compatibility reference.

The DAS could become the input for creating a container that could be deployed to other test environments.

11.1 See also

- [Service connector] - Since all services are accessed through a service connector the consumer based tests to verify whether the service satisfies the needs of the consumer will be included in the service connector.

12 Unit testing

Unit Testing

Unit testing exercises the smallest piece of testable software in the application to determine whether it behaves as expected. In this context, a 'unit' is often a function or a method of a class instance. Often, difficulty in writing a unit test can highlight when a module should be broken down into independent more coherent pieces and tested individually. Thus alongside being a useful testing strategy, unit testing is also a powerful design tool, especially when combined with behavior driven development.

Within unit testing we can make a distinction based on whether or not the unit under test is isolated from its collaborators:

- Sociable unit testing - focuses on testing the behaviour of units by observing changes in their state. This treats the unit under test as a black box tested entirely through its interface.
- Solarity unit testing - looks at the interactions and collaborations between an object and its dependencies, which are replaced by mocks.

These styles are not competing and are frequently used in the same codebase to solve different testing problems.

The domain logic within a microservice often manifests as complex complex calculations and a collection of state changes. Since this kind of logic is highly state based there is often little value in trying to isolate the units. This means that we should apply solarity unit testing to the domain model.

With plumbing code like repositories, and service connectors the purpose of the unit tests is to verify the logic used to produce requests or map responses from external dependencies. As such using mocks provides a way to control the request-response cycle in a reliable and repeatable manner. Advantage of testing these components at this level is that they provide faster feedback than integration tests and we can force error conditions by having the mocks replicate error conditions.

Coordination logic such as services care more about the messages passed between the modules and the domain than any complex logic. Using mocks allows the changes to the domain to be verified. If a service requires too many mocks, it is usually a good indicator that some concepts should be extracted and tested in isolation.

12.1 See also

- [ContiPerf] - A Java library for measuring performance by running JUnit tests. This could provide a early hint regarding performance bottle necks.

13 Caching

Cache

Repetitious access to remote resources or global value objects form a bottleneck for many services. Caching is a technique that can drastically improve the performance. For example by avoiding multiple read operations for the same data.

However there is a price, caching data that the application is accessing will increase the memory usage. Therefore it is very important to obtain a proper balance between the retrieval of the data and the memory usage. The quantity of data being cached and the moment when to load, either in the beginning when the application initializes or whenever it is required for the first time, depends on the requirements of the application.

The cache will identify the buffered resources using unique identifiers. When the resources stored in the cache are no longer required they could be released in order to lower the memory consumption.

Basically there are 2 main caching strategies: Primed and Demand cache.

A cache that is initialized from the beginning with default values is primed cache. A primed cache should be considered whenever it is possible to predict a subset or the entire set of data that the client will request, and to put it to the cache. In case the client request data with a key that matches one of the primed keys having no corresponding data in the cache it is assumed that there is no data in the datasource for that key as well. Disadvantage of primed cache is that it takes longer for the application to start-up and become functional. There is however a hidden disadvantage and that is that developers assume that the cache will always be populated and that it will always be populated with the entire data set. This assumption could lead to a disaster for your application because:

- the logic to detect cache misses and fetch individual items on a miss never gets written or is written badly.
- Nothing can ever be evicted from the cache.
- Operations has no options to cleanup when faced with memory or data consistency issues in production.

Draw back of not using a primed cache is that when the application is cold and the cache is empty grabbing data from the datastore is inefficient. A solution to bypass the problems as described above is to provide cache miss logic to operate on sets of identifiers beside the single identifier logic. Now, when the system starts, spawn a couple of workers that fetch ids from the resource (in configurable batch sizes) and insert those in the cache. This way you:

- Ensure that code no longer make assumptions about all data being in cache at all times.
- Caches still get filled close to application start, but clients aren't held back by it.
- Cache warming code and normal get-miss-fetch scenarios share the same code.

Demand cache loads information and stores it in cache whenever the information is requested by the system. A demand cache implementation will improve performance while running. A demand cache should be considered whenever populating the cache is either unfeasible of unnecessary.

Object which are applicable for caching are:

- Results of database queries
- XML message
- Results of I/O

- Any other object that is expensive to get.
- Any object that is mostly read.

A cache requires: * max cache size - Defines how many elements a cache can hold * eviction policies - Defines what to do when the number of elements in cache exceeds the max cache size. The Least Recently Used (LRU) works best. Policies like MRU, LFU, etc, are usually not applicable in most practical situations and are expensive from performance point of view. * Time to live - Defines time after that a cache key should be removed from the cache (expired). * statistics Professional caches like EHCache or Guava Cache provide this kind of functionality out of the box and are constantly tested.

13.1 See also

- Distributed cache - a distributed cache is an extension of the traditional concept of cache used in a single locale. A distributed cache may span multiple servers so that it can grow in size and in transactional capacity. Distributed cache might be interesting in case of demand cache since it will fill up quicker.
- Spring Cache Abstraction
- Hibernate 2nd level cache

14 Circuit Breaker

Circuit Breaker

One of the properties of a remote call is that it can fail, or hang without a response until some timeout limit is reached. If there are many callers of this unresponsive service this can result in the caller run out of critical resources leading to cascading failures across multiple systems. To prevent this kind of catastrophic cascade we can use a circuit breaker.

The basic idea behind the circuit breaker is that you wrap external calls in a circuit breaker object. The circuit breaker object monitors for failures and once the failures reach a certain threshold, the circuit breaker trips, and all further calls via the circuit breaker return an error without the call being made at all. Usually there is also some kind of monitoring whether the circuit breaker has tripped.

This simple form of circuit breaker would need an external intervention to reset it when things are well again. An improvement would be to have the breaker itself detect if the underlying cells are working again.

14.1 Related patterns

- Active Monitoring - Circuit breakers are a valuable place for monitoring and any change in the breaker state should be logged. Breaker behavior is often a good source of warnings about deeper troubles in the environment.
- API Gateway - An API gateway orchestrates the calls between the diverse number of microservices. Each of these calls should be via a orchestrator.

14.2 See also

- Hystrix - A sophisticated tool for dealing with latency and fault tolerance for distributed systems. It includes an implementation of the circuit breaker pattern
- JRugged - A Java library of robustness design patterns.

15 Client Side Discovery

Client Side Discovery

Services typically need to call one another. In a monolithic, services invoke one another through direct calls. In a traditional distributed system deployment, services run at fixed, well known locations (host, port, etc) and can easily call one another using HTTP/REST or some RPC mechanism. A modern microservice based application typically runs in a virtualized or containerized environment where the number of instances of a service and their locations change dynamically. As a consequence of this, you must implement a mechanism that enables clients of services to make requests to a dynamically changing set of service instances.

To overcome this the client obtains the location of a service instance by querying a Service Registry which knows the locations of all service instances.

Disadvantage of this approach is that it couples the client to the service registry and client side service discovery logic needs to be implemented for each language/framework used within the client.

15.1 Related patterns

- Service Connector - Logic that is required for obtaining the location of a service can be implemented within the service connector.
- Service Registry - queried by client, containing locations of all service instances.
- Server Side Discovery is an alternative solution for the lookup of services.
- System of record - To assure that changes for a record are only happening on a single site.

15.2 See also

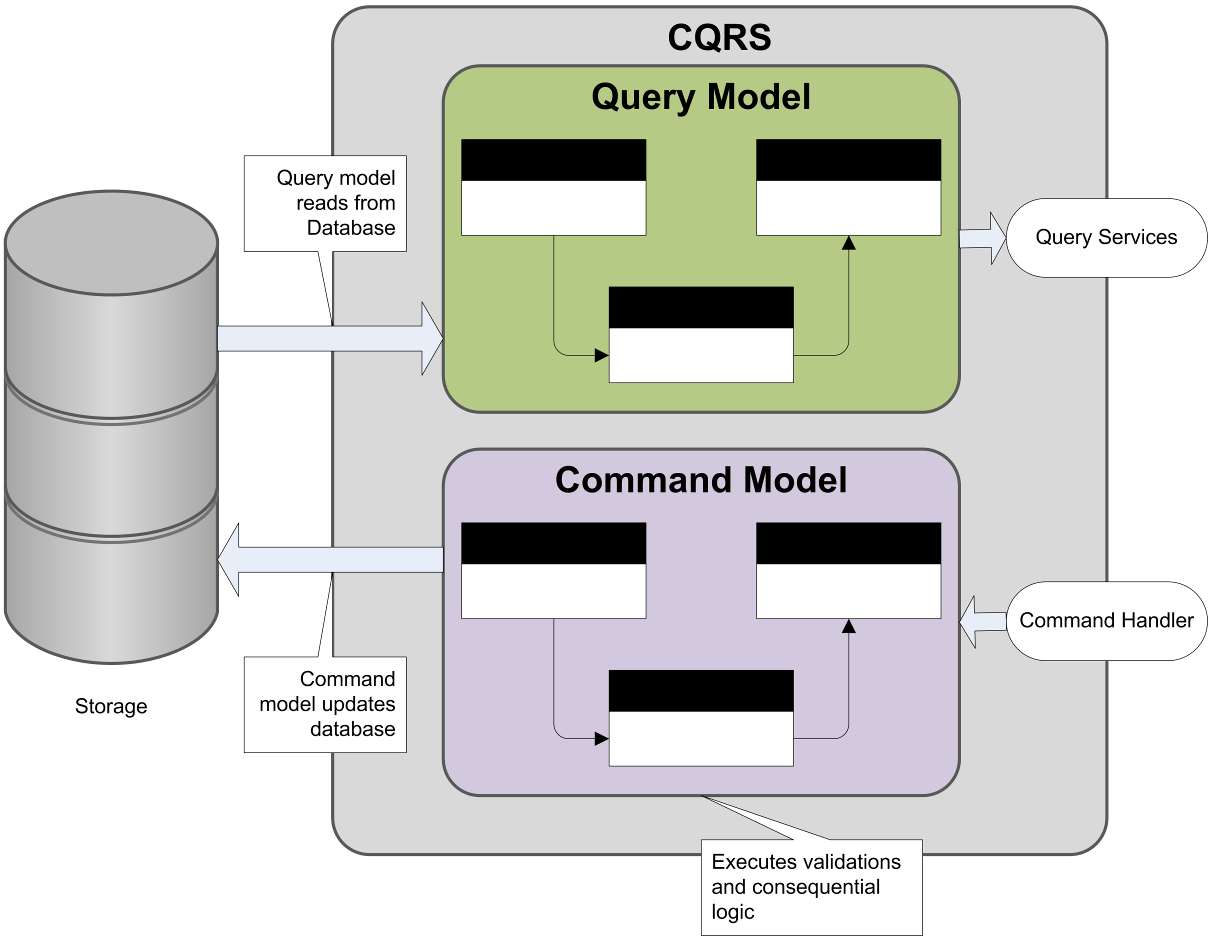
- Ribbon Client is an HTTP client that queries Eureka to route HTTP requests to an available service instance.

16 Command Query Responsibility Separation

Command Query Responsibility Segregation

CQRS stands for Command Query Responsibility Segregation and helps developers to develop scalable, extensible and maintainable applications. At its hearth the pattern makes a distinction between the model in which information is updated and the model from which information is read. Following the vocabulary of *Command Query Separation* the models are called Command and Query. The rationale behind this separation is that for many problems, particularly in complicated domains, having the same conceptual model for updating information and reading information leads to a complex model that does neither well.

The models may share the same database, in which case the database acts as the communication between the two models, but they may also use separate databases, making the query side database a real-time reporting database. In later case there needs to be some communication mechanism between the two models, or their databases. Commonly this communication is realized by events.


CQRS
Query Model
Query model reads from Database
Query Services
Command Model
Command Handler
Storage
Command model updates database
Executes validations and consequential logic

16.1 Querying

A query returns data and does not alter the state of the model. Once the data has been been retrieved by an actor, that same data may have been changed by another actor, in other words it is stale. If the data we are going to return to actors is stale anyway, is it really necessary to go to the master database and get it from there? Why transform the persisted data into entities if we just want data, not any rule preserving behavior? Why transform those entities to DTOs to transfer them across.

In short, it looks like we are doing a lot of unnecessary work. So why not creating a additional model, (whose data can be a bit out of sync), that matches the DTOs as they are expected by the requester. As data store you can use a regular database but that is not mandatory.

16.2 Commands

A command changes the state of an entity within the command model. A command may be accepted or rejected. An accepted command leads to zero or more events being emitted to incorporate new fact into the system. A rejected command leads to some kind of exception.

One should regard a command as a request to perform a unit of work which is not depending on anything else.

16.3 Domain event

In CQRS, domain events are the source all changes in the query model. When a command is executed, it will change state of one or more entities within the command model. As a result of these changes, one or more events are dispatched. The events are picked up by the event handlers of the query model and those update the query model.

16.4 When to use CQRS.

Like any pattern, CQRS is useful in some places and in others not. Many systems do fit a CRUD model, and should be done in that style. In particular CQRS should only be used on specific portions of a system (Bounded context in DDD lingo) and not to the system as whole; e.g. each bounded context needs its own decision on how it should be modeled.

So far there are the following benefits:

- Handling complexity - a complex domain may be easier to tackle by using CQRS.
- Handling high performance applications - CQRS allows you to separate the load from reads and writes allowing you to scale each independently.

17 Deployment microservices

Deployment of microservices

So you have applied the different patterns as described within these guidelines and architected your application as a set of services. Each service should be deployed as a set of service instances for throughput and availability.

Forces that (could) influence the decision are:

- Services are written using a variety of languages, frameworks and framework versions.
- Services are independently deployable and scalable.
- Services need to be isolated from each other.
- Need to be able to quickly build and deploy service
- Need to be able to restraint the resources (CPU/Memory) consumed by a service.
- Need to monitor behavior for each service instance.
- Deployment of application must be cost-effective

17.1 Multiple service instances per host

Multiple instances of different services are running on a single host (physical or virtual machine). There are various ways of deploying:

- Deploy each service instance as a JVM process, (for example a Tomcat instance per service instance).
- Deploy multiple service instances in the same JVM, (for example as OSGI bundles)

The resource utilization within this approach is more efficient than the service instance per host. The drawbacks of this approach include:

- Risk of conflicting resource requirements.
- Risk of conflicting dependency versions.
- Difficult to limit the resources consumed by a service instance.
- Difficult to monitor the resource consumption of each service.
- Impossible to isolate each service instance.

17.2 Single service instance per host

Within this approach we deploy each service instance on it’s own host. The benefits of this approach include:

- Service instances are isolated from one other.
- There is no possible conflict regarding resource requirements or dependency versions.
- A service instance can consume at most the resources or a single host.
- It is straight forward to monitor, manage, and to redeploy a service instance.

The drawback is less efficient resource utilization compared to multiple service instances per host.

17.3 Service instance per VM

Within this approach we package the service as a virtual machine image and deploy each service instance as a seperate VM.

The benefits of this approach include:

- Straightforward to scale the service by increasing the number of instances.
- VM encapsulates the details of the technology used to build the service.
- Each service instance is isolated.
- VM imposes limits on the CPU and memory consumed by the service instance.

The drawback of this approach include that building a VM image is slow and time consuming.

17.4 Service instance per container

Package the service as a container image (Docker) and deploy each service instance as a container. The benefits of this approach are:

- Straightforward to scale up and down a service by changing the number of container instances.
- Container encapsulates the details of the technology used to build the service. All services are, for example, started and stopped in exactly the same way.
- Each service instance is isolated.
- Container imposes limits on the CPU and memory consumed by a service instance.
- Containers are fast to build and start.

Drawback is that the infrastructure is not as rich as the infrastructure for deploying virtual machines.

TODO

18 Domain Driven Design

Introduction Domain Driven Design

The philosophy of Domain Driven Design (DDD), firstly described by Eric Evans, is about placing our attention at the heart of the application, focusing on the complexity that is intrinsic to the business domain itself.

Domain Driven Design consists of a set of patterns for building enterprise applications based on the domain model. Within this introduction we are going to run through some of concepts and terminology of DDD.

18.1 Domain Model

At the heart of DDD lies the concept of the domain model. This model is built by the team responsible for developing the system. The team consists of both domain experts from the business and software developers. The role of the domain model is to capture what is valuable or unique to the business. The domain model serves the following functions:

- It captures the relevant domain knowledge from the domain experts.
- It enables the team to determine the scope and verify the consistency of the knowledge.
- The model is expressed in code by the developers.
- It is constantly maintained to reflect evolutionary changes in the domain.

Domain models are typically composed of elements such as entities, value objects, aggregates, and described using terms from a ubiquitous language.

18.2 Ubiquitous language

The ubiquitous language is very closely related to the domain model. One of the functions of the domain model is to create a common understanding between the domain experts and the developers. By having the domain experts and developers use the same terms of the domain model for objects and actions within the domain, the risk of confusion or misunderstanding is reduced.

18.3 Entities, value objects and services

DDD uses the following concepts to identify some of the building blocks that will make up the domain model:

- Entities are objects that are defined by their identity and that identity continuous through time.
- Value objects are objects which are not defined by their identity. Instead they are defined by the values of their attributes.
- Services is a collection of stateless methods that doesn’t model properly to a entity or value object.

18.4 Aggregates and aggregate roots

DDD uses the term aggregate to define a cluster of related entities and value objects that form a consistency boundary within the system. That consistency boundary is usually based on transactional consistency. The aggregate root (also known as root entity) is the gatekeeper object for the aggregate. All access to the objects within the aggregate must occur through the aggregate root; external entities are only permitted to hold a reference to the aggregate root. In summary, aggregates are mechanism that DDD uses to manage the complex set of relationships between the many entities and value objects in a typical domain model.

18.5 Bounded context

For a large system, it may not be practical to maintain a single domain model; the size and complexity would make it difficult to keep it consistent. To address this, DDD introduces the concept of bounded context and multiple models. Within a system, you might choose to use multiple smaller models rather than a single large model, each focusing on a aspect or grouping of functionality within the overall system. A bounded context is the context for one particular domain model. Each bounded context has its own ubiquitous language, or at least its own dialect of the domain ubiquitous language.

18.5.1 Anti corruption layers

Different bounded contexts have different domains models. When your bounded contexts communicate with each other, you need to ensure that concepts specific to one domain do not leak into another domain model. An anti corruption layer functions as a gatekeeper between bounded contexts and helps you to keep the domain models clean.

18.5.2 Context maps

A large complex system can have multiple bounded contexts that interact with each other in various ways. A business entity, such as a customer, might exist in several bounded contexts. However, it may need to expose different facets or properties that are relevant to a particular bounded context. As a customer entity moves from one bounded context to another, you may need to translate it so that is exposes the relevant facets or properties for that particular context. A context map is the documentation that describes the relationships between these bounded contexts.

18.5.3 Bounded contexts and multiple architectures

Following the DDD approach, the bounded context will have its own domain model and its own ubiquitous language. Bounded contexts are also typically vertical slices through the system, meaning that the implementation of a bounded context will include everything from the data store, right up to the UI. One important consequence of this split is that you can use different implementation architectures in different bounded contexts. Due to this you can use an appropriate technical architecture for different parts of the system to address its specific characteristics.

18.5.4 Bounded contexts and multiple development teams

Clearly separating bounded contexts, and working with separate domain models and ubiquitous languages makes it possible to develop bounded contexts in parallel.

18.5.5 Maintaining multiple bounded contexts

It is unlikely that each bounded context will exist in isolation. Bounded contexts will need to exchange data with each other. The DDD approach offers a number of approaches for handling the interactions between multiple bounded contexts such as using anti-corruption layers or using shared kernels.

19 Domain Model

Domain Model

Domain Driven Design is based on the idea of solving problems the organisations face through code. This is achieved by focusing on the domain of the business you are working with and the problems they want to solve. This will typically involve rules, processes and existing systems that need to be integrated as part of your solution; e.g. the domain is the ideas, entities and knowledge of the problem you are trying to solve. All of the knowledge around the company and how it operates will form the domain.

The Domain Model is your solution to the problem. The Domain Model usually represents an aspect of reality or something of interest. The Domain Model is often a simplification of the bigger picture, the important aspects of the solution are included while everything else is ignored.

The Domain Model should represent the vocabulary and key concepts of the problem domain and it should identify the relationships among all of the entities within the scope of the domain. The important this is that the Domain Model should be accessible and understandable by everyone who is involved with the project.

So the Domain is the world of the business and the Domain Model is the structured knowledge of the problem, but why is this important to Domain Driven Design? Well there are 3 reasons:

1. The Domain Model and implementation shape each other - The code that is written should be intimately linked to the Domain Model.Anyone on the team (business, developers, etc) should be able to look at your code and understand how it applies to the problem you are solving. Whenever a decision needs to be made during the course of the project everyone should refer to the Domain Model to look for the answer.
2. The Domain Model is the backbone of the language used by all team members - Every member of the team should use the ubiquitous language. This ensures that technical and non-technical people have a common language so there is no loss of understanding between parties. The ubiquitous language should be directly derived from the Domain Model.
3. The Domain Model is distilled knowledge - The Domain Model is the outcome of an iterative discovering process where everyone on the team is involved to discuss the problem and how it should be solved. The Domain Model should capture how all think about the project and all of the distilled knowledge that has been derived from those collaboration sessions.

19.1 How do you create at a Domain Model?

Creating a Domain Model is an iterative process that attempts to discover the real problem that is faced and the correct solution that is required. It's important to focus a Domain Model on one important problem. Trying to capture the entire scope of a business inside a single Domain Model will be far too overcomplicated and most likely work contradicting as concepts and ideas move around within the organisation, (see bounded context on how to cope with this).

The problem should be mapped out through talking to business experts to discover the problems they face. This should form the conceptual problem that you are looking to tackle. Business experts won't talk in terms of technical solutions, so during the process important aspects of the problem should be picket out from the language and given concrete definitions to form the ubiquitous language.

There should be a tight feedback loop where everyone on the project discuss the proposed Domain Model to get closer to the solution. This is an iterative approach that will likely encompass code and diagrams to really understand the problem and discover the correct solution.

20 Building blocks DDD

Building blocks Domain Driven Design

The diagram below is a navigational map. It shows the patterns that form the building blocks of Domain Driven Design and how they relate to each other.

By using these standard patterns we bring order in the design and make it easier for team members to understand each other’s work. Using standard patterns also adds to the *ubiquitous language* which all team members can use to discuss model and design discussions.

21 Aggregates

Aggregates.

It is difficult to guarantee the consistency of changes to objects in a model with complex associations. Invariants need to be maintained that apply to closely related groups of objects, not just discrete objects. Yet cautious locking schemes cause multiple users to interfere pointlessly with each other and make a system unusable.

Therefore, cluster the `Entities` and `Value Objects` into `Aggregates` and define boundaries around each. Choose one `Entity` to be the root of each `Aggregate`, and control all access to the object inside the boundary through the root. Allow external objects to hold references to the root only. Transient references to internal members can be passed out for use within a single operation only. Because the root controls access, it cannot be blindsided by changes to the internals. This arrangement makes it practical to enforce all invariants for objects in the `Aggregate` and for the `Aggregate` as a whole in any state change.

22 Entities

Entities

Many object are not fundamentally defined by their attributes, but rather by a thread of continuity and identity.

Some objects are not defined primarily by their attributes. They represent a thread of identity that runs through time and often across distinct representations. Sometimes such an object must be matched with another object even though attributes differ. Mistaken identity can lead to data corruption.

Therefore, when an object is distinguished by its identity, rather than its attributes, make this primary to its definition in the model. Keep the class definition simple and focused on life cycle continuity and identity. Define a means of distinguishing each object regardless of its form or history. Be alert to requirements that call for matching objects by attributes. Define an operation that is guarenteed to produce a unique identity for each object. The model must define what it *means* to be the same thing.

23 Factories

Factories

When creation of an object, or an entire `Aggregate`, becomes complicated or reveals too much of the internal structure `Factories` provide encapsulation.

Creation of an object can be a major operation in itself, but complex assembly operations do not fit the responsibility of the created object. Combining such responsibilities can produce ungainly designs that are hard to understand. Making the client direct construction muddies the design of the client, breaches encapsulation of the assembled object or `Aggregate`, and overly couples the client to the implementation of the created object.

Therefore, shift the responsibility for creating instances of complex objects and `Aggregates` to a separate object, which may itself have no responsibility in the domain model but is still part of the domain design. Provide an interface that encapsulates all complex assembly and that does not require the client to reference the concrete classes of the object being instantiated. Create entire `Aggregates` as a piece, enforcing their invariants.

24 Layered Architecture

Layered Architecture

In an object oriented program UI, database, and other support code often gets written directly into the business objects. Additional business logic is embedded in the behavior of UI widgets and database scripts. This happens because it is the easiest way to make things work, in the short run. When the domain related code is diffused through such a large amount of other code, it becomes extremely difficult to see and reason about. Superficial changes to the UI can actually change business logic. To change a business rule may require meticulous tracing of UI code, database code, or other programming elements. Implementing coherent model driven objects impractical and automated testing is awkward. With all the technologies and logic involved in each activity, a program must be very simple or it becomes impossible to understand.

Therefore:

- Partition a complex program into ’layer’s.
- Develop a design within each ‘layer’ that is cohesive and that depends only on the layers below.
- Follow standard architectural patterns to provide loose coupling to the layers above by means of a mechanism such as *Observer* or *Mediator*; there is never a direct reference from lower to higher.
- Concentrate all the code related to the domain model in one layer and isolate it from the user interface, application, and infrastructure code. The domain objects, free of the responsibility of displaying themselves, storing themselves, managing application tasks, and so forth, can be focused on expressing the domain model. This allows a model to evolve to be rich and clear enough to capture essential business knowledge and put it to work.

A typical enterprise application architecture consists of the following four conceptual layers:

- *Presentation Layer* - Responsible for presenting information to the user and interpreting user commands.
- *Application Layer* - Layer that coordinates the application activity. It doesn’t contain any business logic. It does not hold the state of business objects, but it can hold the state of an application task’s progress.
- *Domain Layer* - This layer contains information about the business domain. The state of business objects is held here. Persistence of the business objects, and possibly their state is delegated to the infrastructure layer.
- *Infrastructure Layer* - This layer acts as a supporting library for all the other layers. It implements persistence for business objects, contains supporting libraries, etc.

24.1 Application layer.

The application layer:

- Is responsible for the navigation between the UI screens in the bounded context as well as the interaction with application layers of other bounded contexts.

- Can perform the basic (non business related) validation on the user input data before transmitting it to the other (lower) layers of the application.
- Doesn’t contain any business or domain related logic.
- Doesn’t have any state reflecting a business use case but it can manage the state of the user session or the progress of a task.
- Contains application services.

24.2 Domain layer.

The domain layer:

- Is responsible for the concepts of business domain, and the business rules. Entities encapsulate the state and behavior of the business domain.
- Manages the state of a business use case if the use case spans multiple user requests, (e.g. loan registration process which consists of multiple steps: user entering loan details, system returning products and rates, user selecting particular product, system locking the loan for selected rate).
- Contains domain services.
- Is the heart of the bounded context and should be well isolated from the other layers. Also, it should not be dependent on the application frameworks used in the other layers, (Hibernate, Spring, etc).

24.3 CRUD operations

TODO: see Domain Driven Design (DDD) architecture layer design for CRUD operation

25 Modules

Modules

Everyone uses `Modules` but few treat them as full fledged part of the model. Code gets broken down into all sort of categories, from aspects of the technical architecture to developers work assignments. It is a truism that there should be low coupling between `Modules` and high cohesion within them. Explanations of coupling and cohesion tend to make them sound like technical metrics, to be judged mechanically based on the distributions of associations and interactions. Yet it isn't just code being divided into `Modules`, but concepts. There is a limit to how many things a person can think about at once (hence low coupling).

Therefore, choose `Modules` that tell the story of the system and contain a cohesive set of concepts. Seek low coupling in the sense of concepts that can be understood and reasoned about independently of each other. Refine the model until it partitions according to the high level domain concepts and the corresponding code is decoupled as well. Give the `Modules` names that become part of the ubiquitous language. `Modules` and their names should refelect insight into the domain.

26 Repositories

Repositories.

A client needs a practical means of acquiring references to preexisting domain objects. If the infrastructure makes it easy to do so, the developer of the client may add more traversable associations, muddling the model. On the other hand, they may use queries to pull the exact data they need from the database, or to pull a few objects rather than navigating from the ‘Aggregate’ roots. Domain logic moves into queries and client code, and the `Entities` and `Value Objects` become mere data containers. The sheer technical complexity of applying most database access infrastructure quickly swamps the client code, which leads developers to dumb down the domain layer, which makes the model irrelevant.

Therefore, for each type of object that needs global access, create an object that can provide the illusion of an in memory collection of all objects of that type. Setup access through a well known global interface. Provide methods to add and remove objects, which will encapsulate the actual insertion and removal of data in the store. Provide methods that select objects based on some criteria and return fully instantiated objects of object whose attribute values meet the criteria, thereby encapsulating the actual storage and query technology. Keep the client focused on the model, delegating all object storage and access to the `Repositories`.

27 Services

Services.

Some concepts from the domain aren't natural to model as objects. Forcing the required domain functionality to be the responsibility of an `Entity` or `Value Object` either distorts the definition of a model based object or adds meaningless artificial objecs.

Therefore, when a significant process or transformation in the domain is not a natural responsibility of an `Entity` or `Value Object`, add an operation to the model as a standalone interface declared as a `Service`. Define the interface in terms of the language of the model and make sure the operation name is part of the ubiquitous language. Make the service stateless.

27.1 Service types.

Services exist in most layers of the DDD layered architecture:

- Application
- Domain
- Infrastructure

An infrastructure service would be something that communicates directly with external resources, such as file systems, databases, etc.

Domain services are the coordinators, allowing higher level functionality between many different smaller parts. Since domain services are first-class citizens of the domain model, their names and usage should be part of the ubiquitous language. Meanings and responsibilities should make sense to the stakeholders or domain experts.

The application service will be acting as façade and accepts any request from clients. Once the request comes, based on the operation, it may call a Factory to create domain object, or calls repository service to re-create existing domain objects. Conversion between, DTO (Data Transfer Object) to domain objects and Domain objects to DTO will be happening here. Application service can also call another application service to perform additional operations.

The differences between a domain service and an application service are subtle but critical:

- Domain services are very granular where as application services are a facade with as purpose providing an API.
- Domain services contain domain logic that can't naturally be placed in an entity or value object, whereas application services orchestrate the execution of domain logic and don't themselves implement any domain logic.
- Domain service methods can have other domain elements as operands and return values whereas application services operate upon trivial operands such as identity.
- Application services declare dependencies on infrastructural services required to execute domain logic.
- Command handlers are a flavor of application services which focus on handling a single command typically in a CQRS architecture.

28 Value Objects

Value Objects

Many objects have no conceptual identity. These objects describe some characteristic of a thing.

Tracking the identity of `Entities` is essential, but attaching identity to other objects can hurt system performance, add analytical work, and muddle the model by making all object look the same. Software design is a constant battle with complexity. We must make distinctions so that special handling is applied only where necessary. However if we think of this category of objects as just the absence of identity , we haven’t dded much to our toolbox or vocabulary. In fact, these objects have characteristics of their own, and their own significance to the model. These are the objects that describe things.

Therefore, when you care only about the attributes of an element of the model, classify it as a `Value Object`. Make it express the meaning of the attributes it conveys and give it related functionality. Treat the `Value Object` as immutable. Don’t give it any identity and avoid the design complexities necessary to maintain `Entities`.

29 Context mapping

Context Mapping

TODO:

Include documentation regardign AcL Anti corruption Layer, and other approaches (hexa pattern?).

30 Enterprise Integration Pattern

Enterprise Integration Pattern

TODO

31 Correlation ID

Correlation ID

Microservices call each another and it will be difficult to figure out how one particular request got transformed and which services where called. By using a correlation ID that is passed within the calls between services we enable the tracking of requests and its route.

TODO: determine whether we should use (custom) HTTP Header or URI parameter. Advantage of URI parameter is that it is easier log-able, however caching would become complexer.

31.1 See also

- Implementing Correlation IDs in Spring Boot

32 Exclusive Consumer

Exclusive Consumer

If there are multiple message consumer instances consuming from the same queue you will loose the guarantee of processing the messages in order since the messages will be processed concurrently in different threads. Sometimes its important to guarantee the order in which messages are processed.

A common approach in this case is to pin one particular JVM in the cluster to have one consumer on the queue to avoid loosing ordering. The problem with this is that if that particular JVM goes down, no one is processing the queue any more.

With a exclusive consumer we avoid to pin a particular JVM. Instead the message broker will pick a message consumer to get all the messages for that queue to ensure ordering. If that consumer fails, the broker will automatically failover and choose another consumer.

The effect is a heterogeneous cluster where each JVM has the same setup and configuration; the message broker is choosing one consumer to be the master and send all the messages to it until it dies; then you get immediately fail-over to another consumer.

32.1 Parallel exclusive consumer

By including the concept of message groups we can create a kind of parallel exclusive consumers. Rather that all messages going to a single consumer, a message group will ensure that all messages for the same message groups will be sent to the same consumer while that consumer stays alive. As soon as the consumer dies another will be chosen.

When a message is being dispatched to a consumer, the message group is checked. If there is a message group present the broker checks to see if a there is a consumer that owns the message group. If no consumer is associated with a message group, a consumer is chosen. That message consumer will receive all further messages with the same message group until:

- The consumer closes.
- The message group is closed.

32.2 Resources

- Exclusive Consumer within ApacheMQ
- Parallel exclusive consumers within ApacheMQ

33 Message Router

Message Router

A message router consumes a Message from a Message Channel and republish it to a different message channel depending on a set of conditions.

A key property of the message router is that it does not modify the message content. It only concerns itself with the destination of the message.

34 Exception Handling

Exception Handling

TODO: Write nice intro.

34.1 Exception handling and logging

…

34.2 Best practices

- Use checked exceptions for recoverable errors and unchecked exceptions for programming errors - Checked exceptions ensure that you handle certain error conditions, but at the same time it also adds a lot of clutter in the code and might make it unreadable. Also it is only reasonable to catch exceptions if you have alternatives or recovery strategies.
- Avoid unnecessary exception handling - Exceptions are costly and can slow down your code. Don’t just throw and catch exceptions, if you can use standard return values to indicate result of an operation. Also avoid unnecessary handling by fixing the root cause.
- Never swallow the exception in catch block.
- Declare specific checked exceptions that method can throw - Declare specific checked exceptions that can be thrown by your method. If there are too much checked exceptions, you should wrap them in your own exception and add information in the exception message.
- Never catch the `Exception` class - The problem with catching `Exception` is that if the method that you are calling adds a new checked exception to its method signature you will never know about it and the fact that your code now might be wrong and might break at any point in time.
- Never catch `Throwable` class - JVM errors are also subclass of `Throwable` and these errors are irreversible conditions that cannot be handled by the JVM itself. The JVM might even not invoke your `catch` clause on an error.
- Always wrap exceptions correctly - the original exception should always be included within the exception that wraps it.
- Either log the exception or throw it but never do both - Logging and throwing will result in multiple messages in the log for a single problem in the code.
- Never throw any exception form finally block - if we would throw an exception from the finally block we might hide the original first exception and correct reason would be lost forever.
- Only catch exception you can handle - Catch an exception only if you want to handle it or to provide additional contextual information in that exception.
- Don’t print stack traces to the console.
- Use finally blocks instead of catch blocks if you are not going to handle exception. If a method throws some exception which you do not want to handle, but still want to perform some cleanup, then do this cleanup in the `finally` block. Do not use `catch` block.
- Prefer exceptions to return codes - it is preferable to throw understandable messages that are part of your API contract and guide the client programmer.
- Throw early catch late - This principle implicitly says that you will throw an exception in the low level methods and make the exception climb the stack trace until you reached a sufficient level of abstraction to be able to handle the problem.
- Always cleanup after handling exception - If you are using resources like databases make sure you clean them up. You can use the new java 7 auto-cleanup via try-with-resources statement.
- Exception names must be clear and meaningful - Use for specific exceptions clear names that states the cause of the exception. For example `AccountLockedException` instead of `AccountEcxception`.

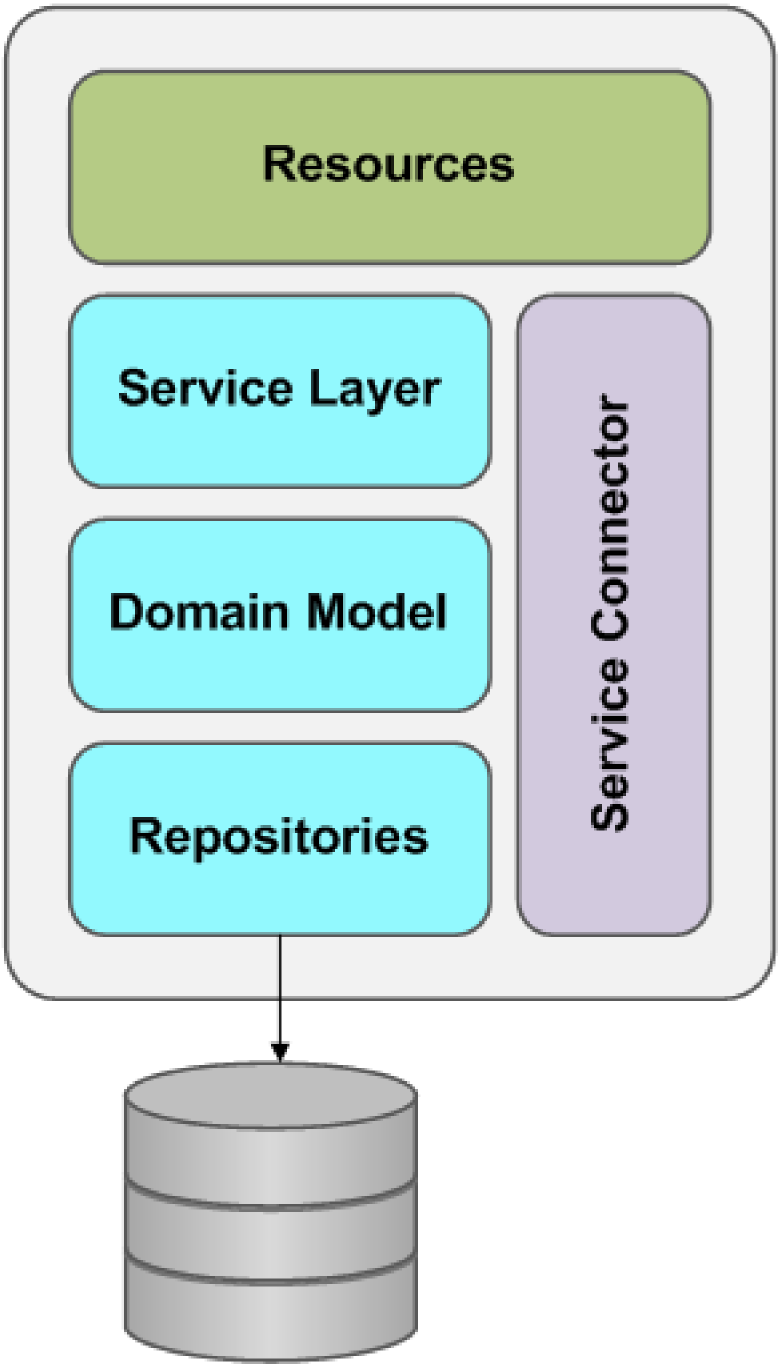
- Never use exceptions for flow control - it makes code hard to understand.
- Do not handle exceptions inside loops. Surround the loop with exception block instead.
- Always include all information about an exception in a single log message.
- Document all exceptions in your application in javadoc.

35 Microservice Architecture

Microservice architecture

A microservice architecture is the natural consequence of applying the single responsibility principle at architectural level. Within a microservice architecture the functionality is decomposed in a set of collaborating services and the scale cube is applied. Services communicate with each other either using synchronous protocols like HTTP/REST or asynchronous protocols such as AMQP/JMS. In this way, business domain concepts are modelled as resources with one or more of these managed by a microservice. Since a business request can span multiple microservices separated by network partitions, it is important to consider possible failures in the system. Techniques such as timeouts, circuit breakers, bulkheads can help to maintain overall system uptime in spite of outage.

Often microservices display similar internal structure consisting of some or all of the layers as shown below.



- Resources act as a mapper between the application protocol as exposed by the micro service and messages to the services/ entities that are representing the domain. Typically, they are thin, responsible for sanity checking the request, and providing a protocol specific response according to the outcome of the request. If we for example expose the resources via REST and follow internally the CQRS approach, this layer would be responsible for converting the HTTP REST request into a `Command`.

- The microservice logic resides in the domain model. Services coordinate across multiple domain activities, whilst repositories act on collections of domain entities and are often persistent. When a resource receives a request and has validated it, it either directly access the domain model, or, if many entities must be coordinated, it delegates the request to a service.
- If a service has another service as a collaborator, some logic is needed to communicate with the external service. A service connector encapsulates message passing with a remote service, marshalling requests and responses from and to other services. Service connectors should be resilient to outage of remote components.

Normally services are developed independently from another. Typically, a team will act as guardian to one or more microservices,

Each service has its own storage in order to decouple from other services. When necessary, consistency between services is maintained using application events.

Following this approach has a number of benefits:

- Each service is relatively small
  - Easier for developers to understand.
  - The web container starts fast, which makes developers more productive and speed up deployments.
- Each service can be deployed independently of other services; e.g. easier to deploy new versions of services frequently.
- Easier to scale development. It enables to organize the development effort around multiple teams. Each team can develop, deploy, and scale their service independently of all other teams.
- Improved fault isolation.
- Each service can be developed and deployed independently.
- Eliminates long term commitment to a technology stack.

There are however also a number of drawbacks:

- Developers must deal with the additional complexity of creating distributed systems.
  - Testing is more difficult.
  - Developers must implement the inter-service communication mechanism
  - Implementing use cases that span multiple services without using distributed transactions is difficult (See Try-Cancel/Confirm)
  - Implementing uses cases that span multiple services requires careful coordination between the teams.
- Deployment complexity in production; there is additional operational complexity of deploying and managing a system comprised of many different services.
- Increased memory consumption. The microservice architecture replaces a monolithic application with N service instances. If each service runs in its own

JVM (which is usually necessary to isolate instances) there is a overhead of N-1 JVM runtimes. Moreover, if each service runs on its own VM the overhead is even higher.

A challenge is deciding how to partition the system into microservices. One approach is to partition services by use case. We could for example have a micro service shipping within a partitioned e-commerce application that is responsible for shipping completed orders. Another approach is to partition by entity. This kind of service is responsible for all operations that operate on entities of a given type.

Ideally, each service should have only a small set of responsibilities. The Single Responsible Principle states that every class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class. It makes sense to apply this principle to microservice design as well.

35.1 MicroService naming guidelines

There isn't a really straight forward approach regarding the naming of microservices but the following guidelines should help in creating a consistency in the naming of the microservices.

- *Use camel case for microservice name*
- *Don't reveal implementation details in the microservice name* - This not only has the potential to lead to confusion when you change the implementation of the microservice, but is also a security risk as it gives the microservice consumer an insight in how the microservice may be implemented which they may be able to exploit.
- *Don't include protocol information in the microservice name* - This is generally unnecessary as the service advertise itself at a particular endpoint which clearly defines the protocol to be used.
- *Don't include the word service in the microservice name*
- *Don't include a version in the microservice name*
- *Name microservice to entity at which it operates* - Microservices which operate on a specific entity should be named after the entity. For example if service that operates on customers may be simple named `Customer`.
- *Name microservice to functionality it performs* - Certain microservices are taking care of certain processes and therefore the use of verbs in the service name is common. For example a service that orchestrates the device registration could be called `RegisterDevice`.

Naming conventions may seem trivial at first, but as the number of microservices grow, so will the potential to reuse. In larger organizations, this means that more and more architects, analysts, and developers are discovering and then incorporating foreign services within their solution designs. The effort required to establish a consistent level of clarity across all microservices pays off quickly when interoperability and reuse opportunities are more easily recognized and seized.

35.2 TODO

- Check whether OSGI could help with decreasing memory consumption.

35.3 Related Patterns

- API Gateway - defines how clients access the services in a microservice architecture.
- Client side discovery and Service side discovery - Patterns used to route requests for a client to an available microservice.
- Command Query Response Seperation - Helps develop scalable, extensible and maintainable applications.

- Domain Driven Design - Set of patterns for building enterprise applications based on the domain model.
- Monolithic Architecture - alternative to the monolithic architecture.
- Service Connector - Provide high level interface that hides implementation details regarding communication, thereby making the use of the microservice easier.
- Service Statelessness - To have services scalable we should attempt to make them statelessness.

35.4 See also

- JBOSS OSGI User guide - Guide explaining how to deploy application as OSGI bundle on JBOSS server.

36 Mocks

Mocks

Mocks are simulated objects/services that mimic the behavior of real objects/services in a controlled way. A mock is typically created to test the behavior of some other parts of the system.

36.1 See also

- WireMock - WireMock is a flexible library for stubbing and mocking web services.

37 Monolithic Architecture

Monolithic Architecture

The application has either a layered of heaxagonal architecture and consists of different types of components:

- Presentation components - responsible for handling HTTP requests and responding with either HTML or JSON/XML.
- Business logic- the application’s business logic
- Database access logic
- Application integration logic - messaging layer

Problem is how we will deploy this application. Forces that influence the solution are:

- There is a team of developers working on the application.
- New team members must become quickly productive.
- The application must be easy to understand and modify.
- The application should be deployed using the continuous integration practice.
- Multiple instances of the application are running to satisfy requirements regarding scalability, and availability.

Most straightforward solution is to build a application with a monolithic architecture, (for example a single java WAR file). This approach has a number of benefits:

- Simple to develop
- Simple to deploy - you simply need to deploy the WAR file on the appropriate runtime.
- Simple to scale - You can scale the application by running multiple copies of the application behind a load balancer.

However once the application becomes large and teams grows in size, this approach has a number of drawbacks:

- The large monolithic code base intimidates developers and the application can be difficult to understand and modify. As a result, development typically slows down.
- The startup of the application will take longer. This will have a impact on developer productivity because of time waisted waiting for the web server to start. This is especially a problem for user interface developers since they usually need to iterative rapidly and redeploy frequently.
- A large monolithic application makes continuous deployment difficult because in case of an update the entire application has to be redeployed.
- A monolithic architecture can only scale in one dimension. This architecture can’t scale with an increasing data volume. Each instance of the application will access all of the data, which makes caching less effectively and increases memory consumption and I/O traffic. Also different application components have different resource requirements - one might be CPU intensive while another might be I/O intensive. With a monolithic architecture we cannot scale each component independently.

- A monolithic architecture is also an obstacle for scaling development. Once the application gets to a certain size it is useful to divide the development into several teams that focus on specific functional areas. The trouble with monolithic architecture is that it prevents teams from working independently. The teams must coordinate their development efforts and redeployments.
- A monolithic architecture requires a long term commitment to a technology stack.

37.1 Related Patterns

- Microservice Architecture is an alternative to the monolithic architecture.

38 Self registration

Self Registration

Service instances must be registered with the service registry on startup so that they can be discovered and unregistered on shutdown.

Registration of the service should happen with the service registry on startup and unregister on shutdown. If a service instance crashes the service must be unregistered from the service registry. The same applies for service instances that are running but are incapable of handling requests.

Within the self registration solution, a service instance is responsible for registering itself with the service registry. On startup the service instance register itself (location) with the service registry and makes itself available for discovery. The service instance must periodically renew it's registration so that the registry knows it is still alive.

Eureka is an example of a service registry which provides a service registry API and a client library that service instances use to (un)register themselves.

The benefit of the self registration pattern is that a service instance knows best it's own state so a state model that's more complex than `UP`/ `DOWN` should be possible.

There are also a couple of drawbacks:

- Couples the service to the service registry
- Service registration logic must be implemented for all languages that you are using within your environment/client.
- A service which is running but unable to process requests will often lack the self awareness to unregister from the service registry.

38.1 Related patterns

- Active monitoring - if a service crashes it might be that service is not properly unregistered. By actively monitoring the environment we might detect this and remove service instance from registry.
- Service registry - registry at which service instance registers itself, and un-registers when service is shutdown, crashes, or becomes unavailable.
- 3rd party registration - alternative method to register services.

39 Server Side Discovery

Server Side Discovery

Services typically need to call one another. In a monolithic, services invoke one another through direct calls. In a traditional distributed system deployment, services run at fixed, well known locations (host, port, etc) and can easily call one another using HTTP/REST or some RPC mechanism. A modern microservice based application typically runs in a virtualized or containerized environment where the number of instances of a service and their locations change dynamically. As a consequence of this, you must implement a mechanism that enables clients of services to make requests to a dynamically changing set of service instances.

Within a server side discovery approach a request to a service is going via a message router that runs at a well known location. The router queries a service registry, which might be built into the router, and forward the request to an available service instance.

The AWS Elastic Load Balancing is an example of a server side discovery solution. A client makes HTTP(S) requests to the ELB, which load balances the traffic amongst a set of EC2 instances. Within the Amazone solution ELB also functions as a service registry. EC2 instances are either registered with the instance explicitly via an API call or automatically as part of an auto-scaling group.

Server side discovery has a number of benefits:

- Compared to client side discovery, the client code is simpler since it does not have to deal with discovery. Instead a client simply makes a request to a router.

It however also has some drawbacks:

- Unless part of the cloud environment, the router is another system component that must be installed and configured. It will also need to be replicated for availability and capacity.
- More network hops are required compared to client side discovery.

39.1 Related patterns

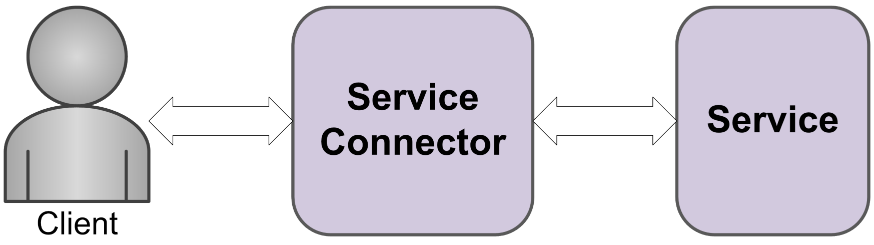
- Client side discovery is an alternative solution.
- Message Router
- Service Registry
- System of record - To assure that changes for a record are only happening on a single site.

40 Service Connector

Service Connector

Clients must know a lot about services in order to use them. REST APIs require clients to use specific media types, HTTP Headers, and specific request methods. Some web services require data to be encrypted or demand that clients submit authentication tokens. The client must also know the service's address, how to serialize requests, and parse responses. Each client could develop a custom solution to meet the specific requirements of the service but that would often result in duplicate code.

Instead we should create a service connector that encapsulates the logic a client can use to call a certain service.



Service connectors make services easier to use by hiding the communication specifics. The service connector encapsulate the generic communication logic required to use the service and also include the logic that is specific go the given service. Service connectors are typically responsible for service discovery and connection management, request dispatch, response handling, and some error handling

40.1 See also

- Inter-Service communication - There are multiple forms in which services communicate with each other and this describes the various forms.
- Consumer-based testing - To ensure that the service contract satisfies the consumer needs the Service Connector contains tests to confirm this.

41 Service Registry

Service Registry

Clients of a service user either client side discovery or service side discovery to determine the location of a service instance to send the requests. By providing a service registry, which is a database of service instances and their locations, the discovery mechanisms are able to find the available instances of a service.

The benefits of the service registry pattern is that the client of the service and/or message router can discover the location of service instances. There are also some drawbacks; Unless part of the cloud environment, the router is another system component that must be installed and configured. Moreover the service registry is a critical component. Although clients should cache data provided by the service registry, if the service registry fails that data will eventually become outdated. Consequently the service registry must be highly available.

Beside the service registry one also need to decide how service instances are registered with the service registry. There are two options:

1. Self registration pattern - service instances register themselves.
2. 3rd party registration pattern - a 3rd party registers the service instances with the service registry.

The clients of the service registry need to know the location(s) of the service registry instances. THis implicitly means that service registry instances must be deployed on fixed and well known locations. Clients are configured with those locations.

Examples of service registries include:

- Eureka
- Zookeeper
- Consul

Related patterns

- Client side - and server side discovery create the need for a service registry.
- self registration and 3rd-party-registration are two different ways that service instances can be registered with the service registry.

42 Service Statelessness

Service Statelessness

The management of excessive state information can compromise the availability of a service and undermine its scalability principle. The services within a distributed application are deployed among multiple resources to benefit the scaling out functionality. The most significant factor complicating addition and removal of service instances is the internal state maintained by the service. In case of failure, this information might even be lost.

Services are therefore ideally designed to be stateless. Instead their state and configuration is stored externally in storage offerings or provided by the component with each request.

43 Single Responsibility Principle

Single Responsibility Principle

This principle states that, a subsystem, class or even function, should have only 1 reason to change. The classic example is a class that has methods that deal with business rules, reports, and persisting:

```
public interface Employee {
    public Money calculatePay();
    public int reportHours();
    public void save();
}
```

The problem with the interface shown above is that the functions change for entirely different reasons. The `calculatePay` function will change whenever the business rules for calculating pay change. The `reportHours` function will change whenever someone wants a different way of reporting hours. The `save` function will change whenever the database scheme is changed. These 3 reasons to change make `Employee` very volatile.

Applying the SRP principle means that we have to separate the interface into components that can be deployed independently. Independent deployment means that is we deploy 1 component we do not have to redeploy any of the others.

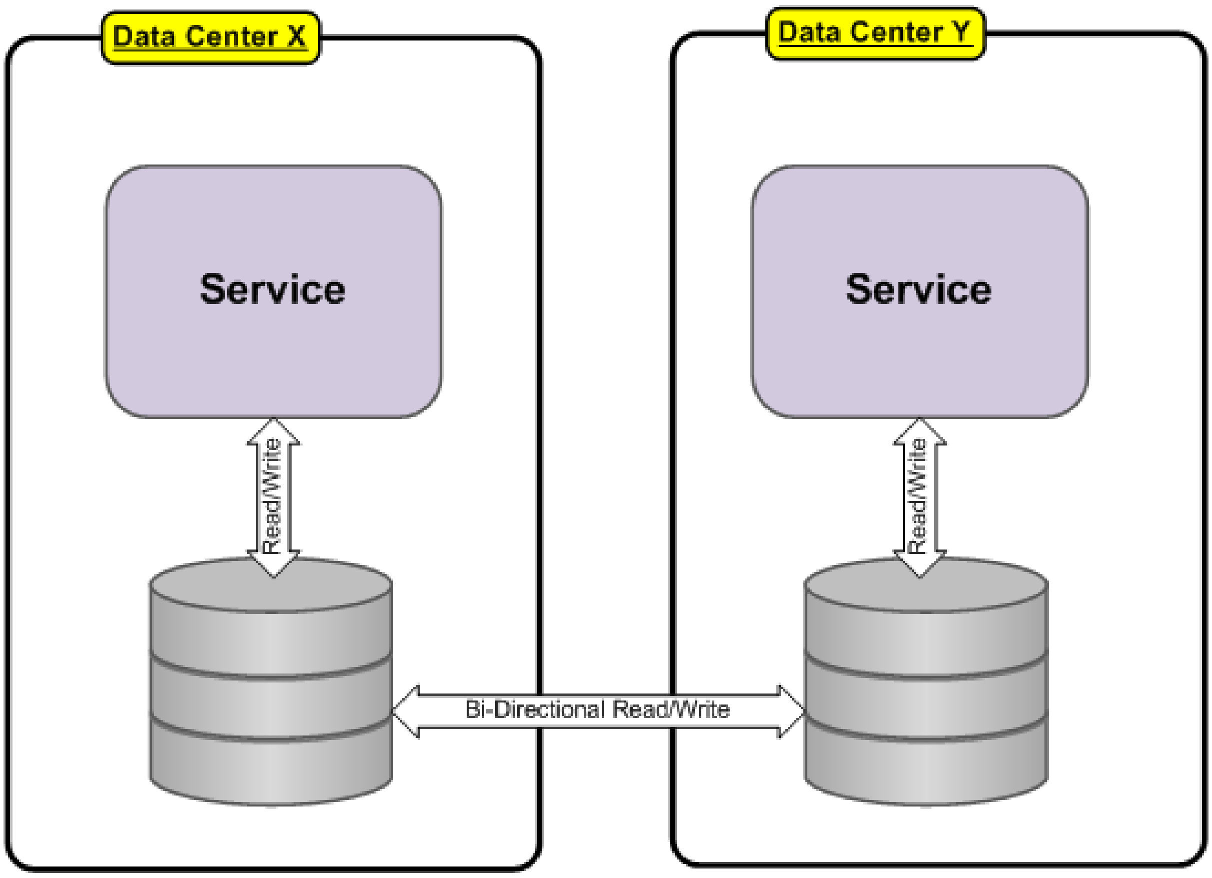
```
public interface Employee {
  public Money calculatePay();
}

public interface EmployeeReporter {
  public String reportHours(Employee e);
}

public interface EmployeeRepository {
  public void save(Employee e);
}
```

The simple partitioning shown above resolves the issue. We have to note that there is still a dependency from `Employee` to the other interfaces `EmployeeReporter` and `EmployeeRepository`. So if `Employee` is changed, the other classes will likely have to be recompiled and redeployed. We could prevent this through a careful use of the Dependency Inversion Principle.

44 System of Record

System of Record

Multi master database systems that span sites have similar data replicated over multiple locations and the data is updated all the time. Goal of these kind of setups is little or no data loss on failure.

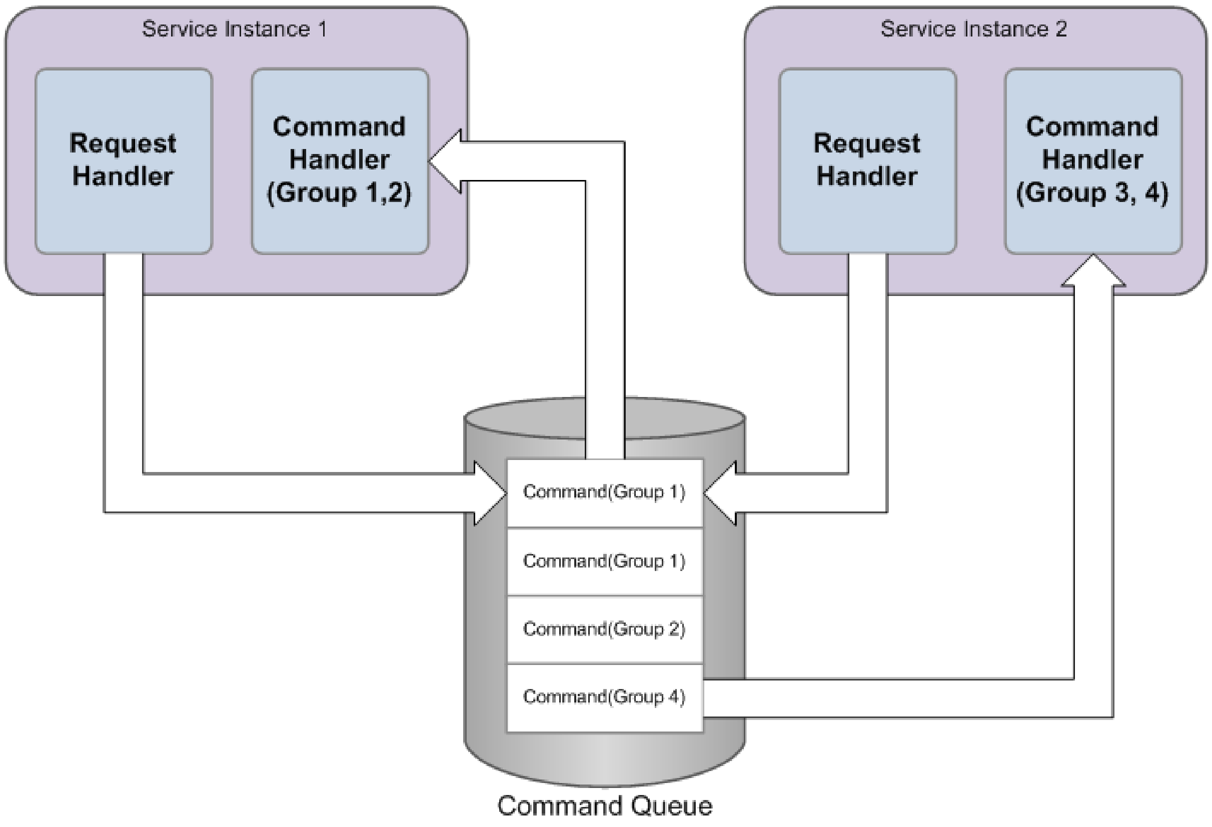


Problem with this solution is that it is impossible to build. The problem is the multi-master replication; updating the same record/table from two or more places on a LAN is already quite difficult, and the problem becomes unmanageable when you combine complex read/write operations, referential integrity, and high latency WAN connections.

Solution for this problem is system of record which states that individual records are updated in a single location only, but may have many copies elsewhere, (both locally as well as on other sites). When clients update particular information they do so on their ‘own’ master.

44.1 Related Patterns

- Exclusive consumer - To assure that only 1 instance is handling changes to certain entities we could use the exclusive consumer. In case of microservices this would mean that all instances of a particular service are listening to a particular queue and that the commands which are put on the queue are grouped by (for example) the ID of the entity at which the command applies. This would mean that the part that puts the command on the queue needs to know how to group the commands, (getting ID of entity at which the command applies from command). In the figure below we attempt to show how such a setup would look. Within this example there are 2 instances of the service whom both receive requests via their `Request Handler`. The `Request Handler` transforms the incoming requests into Commands, determine the group at which they apply, and put them on the queue. The `Command Handler` will listen on the queue for commands within certain groups (standard functionality ActiveMQ) and process the commands within that group in the order that they where posted. Since there is only 1 command handler listening for a certain group we assure the system of record.



45 Try-Cancel/Confirm

Try-Cancel/Confirm.

The Try-Cancel/Confirm pattern focuses on transactions for microservices. A transaction is a set of related interactions (or operations) that may need to be cancelled after they where executed.

When a client initiates a state transition the service will return a handle by which the client can confirm or cancel the state change. If the service does not hear anything after some service specific timeout, it will cancel automatically. Once the workflow has completed successfully the set of returned handles is used to confirm the state transitions. If the service fails, the set of handles that has been collected until the failure is used to cancel the state transitions.

The timeout after which the service will automatically cancel the pending state transitions should be specified by the service.

Related Patterns

- [API Gateway] - some use cases executed by/via an API gateway require multiple services. The API Gateway will orchestrate the sequence of service instantiations.

46 UUID

UUID

In an environment with multiple master relational databases, each master must be able to take updates, while also syncing correctly with others. One common concern is that unique identifiers must not conflict acros master databases. Therefore server features such as table level auto-increment can not be used unless each table is only updated on one master database, or each server is given a distinct range of available auto-increment values.

One solution is for each database server or client to generate a UUID (Universal Unique IDentifier) which is a string which is almost guaranteed to never conflict with another UUID generated on a different computer. Therefore each master database, or client connecting to any database, can generate a UUID and use it as the primary key on a table without much concern for replication conflicts.

Another advantage is that they can be freely exposed without disclosing sensitive information, and they are not predictable.

Downside to this choice is that every primary key is then a string which can be disadvantageous to performance on some systems and generating a massive amount of UUIDs can be expensive to process.

46.1 Related Patterns

- Entities - IDs of entities should be UUID.

47 3rd party registration

3rd Party Registration

Service instances must be registered with the service registry on startup so that they can be discovered and unregistered on shutdown.

Registration of the service should happen with the service registry on startup and unregister on shutdown. If a service instance crashes the service must be unregistered from the service registry. The same applies for service instances that are running but are incapable of handling requests.

Within the 3rd party registration a registrar is responsible for registering and un-registering a service instance with the service registry. When the service instance starts up, the registrar registers the service instance with the service registry and when the service shuts down the registrar un-registers the service from the service registry

The benefits of the 3rd party registration are:

- Service code is less complex than when using self registration since it is not responsible for registering itself.
- The registrar can perform health checks on a service instance and register/un-register the instance based on the health check.

Drawbacks of 3rd party registrations are:

- The registrar might only have a high level view of the service instance state (`UP`/ `DOWN`) and so might not know whether it can handle requests.
- Unless the registrar is part of the infrastructure, it’s another component that must be installed, configured and maintained. Also, since it is a critical system component it need to be highly available.

47.1 Related paterns

- Service Registry
- Client and server side discovery.
- Self registration is an alternative solution.

48 Abbreviations

Abbreviations

- CQRS - Command-Query Responsibility Segregation
- DAS - Deployable Artifact Set
- DBC - Design By Contract
- DIP - Dependency Inversion Principle
- DTO - Data Transfer Object
- LFU - Least Frequently Used
- LRU - Least Recently USed
- REST - REpresentational State Transfer
- SRP - Single Responsible Principle
- TCC - Try-Cancel/Confirm
- UUID - Unique Universal IDentifier
- VV - Version Vector