AgileDev

v. 0.1.9-SNAPSHOT

Guidelines

2015-06-02

Table of Contents

1 Branching Models

Branching Models

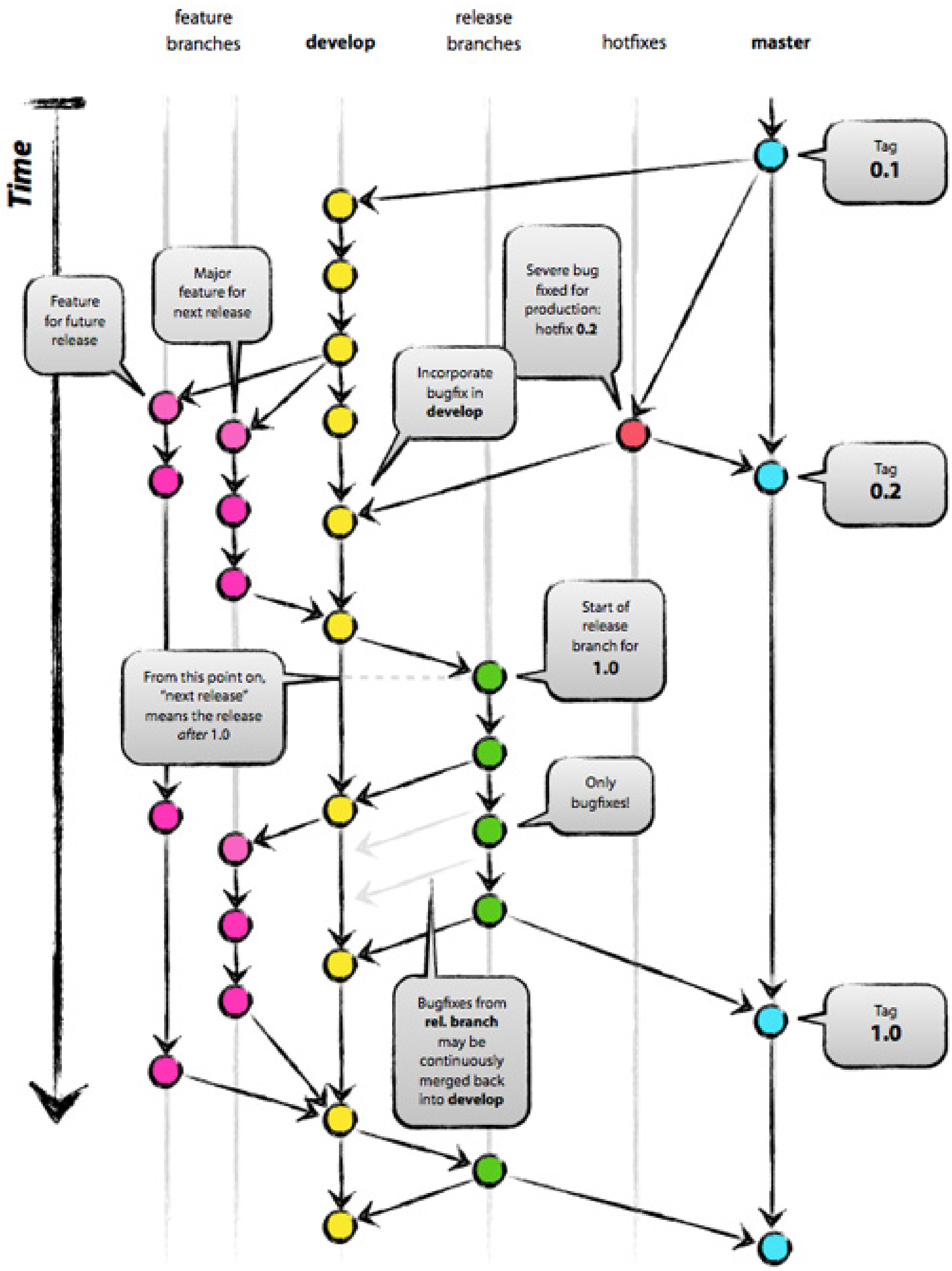
Within the following paragraphs will discuss some of the popular branching models which are around these days. After reading this you should be able to decide which one fits your project the most.

1.1 Git-Flow

This workflow has been published by Vincent Driesen as a successful branching model for git and covers most of the standard needs for a ‘classical’ development project.

1.1.1 How it works

Git-Flow has the following branching model:


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

There are 2 primary branches:

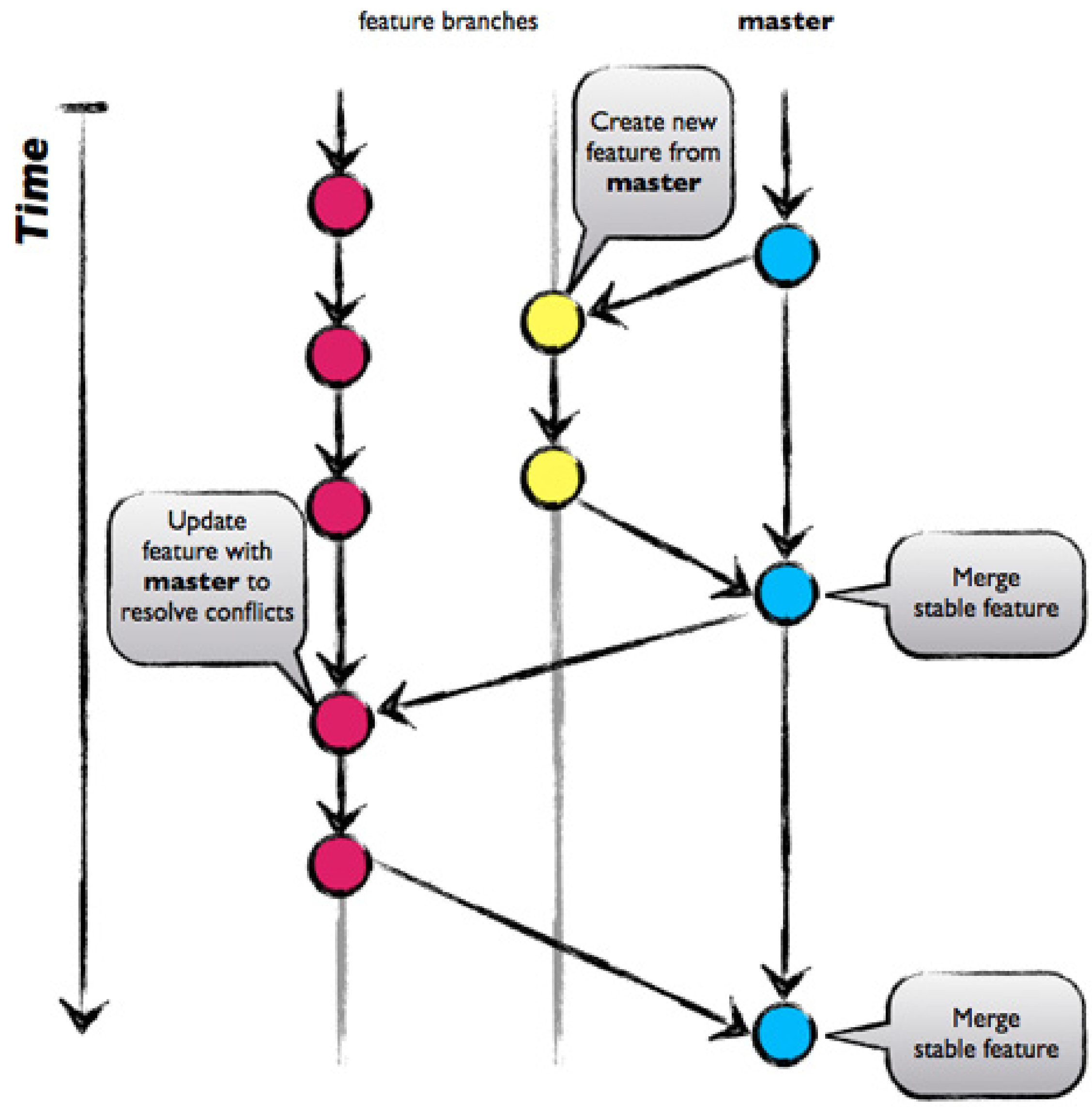
- `master` is the main branch where everything is stable. Each commit is a stable (fully tested) version of the project, (i.e. a release) which could be deployed to production and tagged accordingly.
- `develop` is the main branch where development is done. It will contain prepared changes for the next release in `master`.

And secondary branches which are flexible over time:

- `feature` starts from `develop` and merge into `develop`. When you are working on a specific feature, you create a `feature\xxx` branch and once it's done, you merge it back into `develop` to add the stable feature to the scope for the next release. The essence of a `feature` branch is that it exists as long as the feature is in development, but will eventually be merged back into `develop` or discarded.
- `release` starts from `develop` and merge into `master` and/or `develop`. When `develop` is reflecting the desired state of the feature release, (all features for the release haven been merged), you create a 'release/xxx' branch. By doing this, you can prepare the next release, correct eventual bugs and continue development in parallel. All features targeted at future releases may not be merged into the `develop` branch until the `release` branch is branched off. It is exactly at the start of a release branch that the upcoming release gets a version number assigned.
- `hotfix` starts from `master` and merge into `master` and/or `develop`/ `release`. When you want quickly to resolve critical bugs in production you create a `hotfix/xxx` branch. When the hotfix is developed, you merge it back into `master` with the appropriate version number, and into `develop` and/or `release` branch to update it with the modifications you made.

1.2 GitHub

GitHub has the following branching model:


feature branches
master
Time
Create new feature from master
Update feature with master to resolve conflicts
Merge stable feature
Merge stable feature

As you can see in the picture above, there is only a single `master` branch and the following 6 basic rules should be followed:

1. **Everything that is in `master` could be deployed in production -** The `master` branch is the only meaningful branch of the project and it should stay stable in any circumstance so you can base your work upon and deploy it to production at anytime.
2. **Create descriptive feature branches from `master` -** When you want to develop a feature or a hotfix, you just create your branch from `master` with an explicit name that describe your work.
3. **Regularly push to `origin` -** In contrary of the Git-Flow where developers doesn't have to push its local `feature` branch to the main repo, you have to do that regularly within the GitHub model.
4. **Open a pull-request at anytime**
5. **Only merge after a pull request review -** This is more an advice than an absolute rule. It's a best practice that another developer should review the pull request and confirm that the branch is stable. From here you can, merge the branch back into `master` and delete the `merge` branch.

6. **Immediately deploy after you merge into `master`** - Once your branch is merged into `master` the whole thing is deployed to production. Doing so, you will stress on the necessite to keep `master` stable. Developers don't want to break everything because of its modifications were deployed, so they are more likely to pay attention to code stability before merge.

The GitHub model perfectly fits projects that don't have releases nor versions.

You do continuously integrate into `master` and you deploy the stable project to production regularly; sometimes several time a day.

Due to this it's very unlikely to add serries of big bugs. If problems appear, they are quickly fixed on the go. There is no difference between a big feature and a small hotfix in terms of process.

1.3 Final words

To choose the best model depends on the nature and needs of your project. It's up to you to consider which of these models fit more or less to your environment. If necessary adapt the framework to your needs.

2 Assertions

Assertions

TODO

3 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.

4 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.

5 Patterns

Patterns

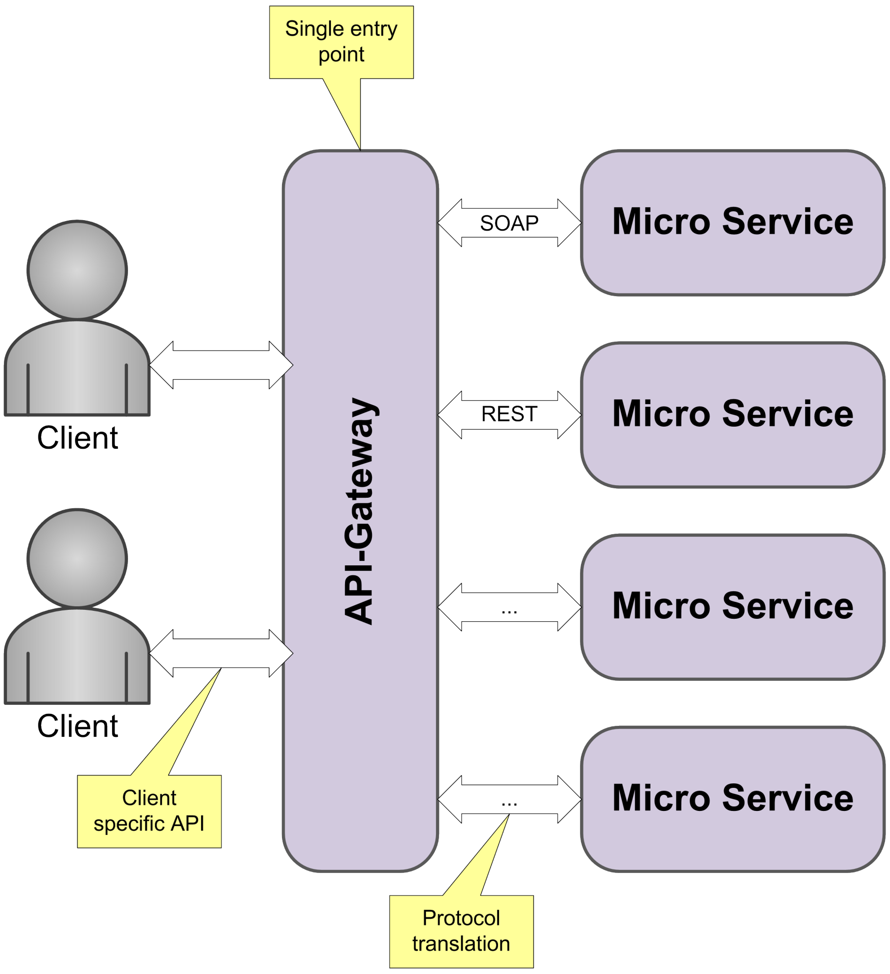
TODO: Include context diagram for patterns

6 API gateway

API Gateway

Your system must expose to clients functionality that they can use. However the granularity of the APIs provided by the micro services is often different from what the client needs. Micro service 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 micro service easier.

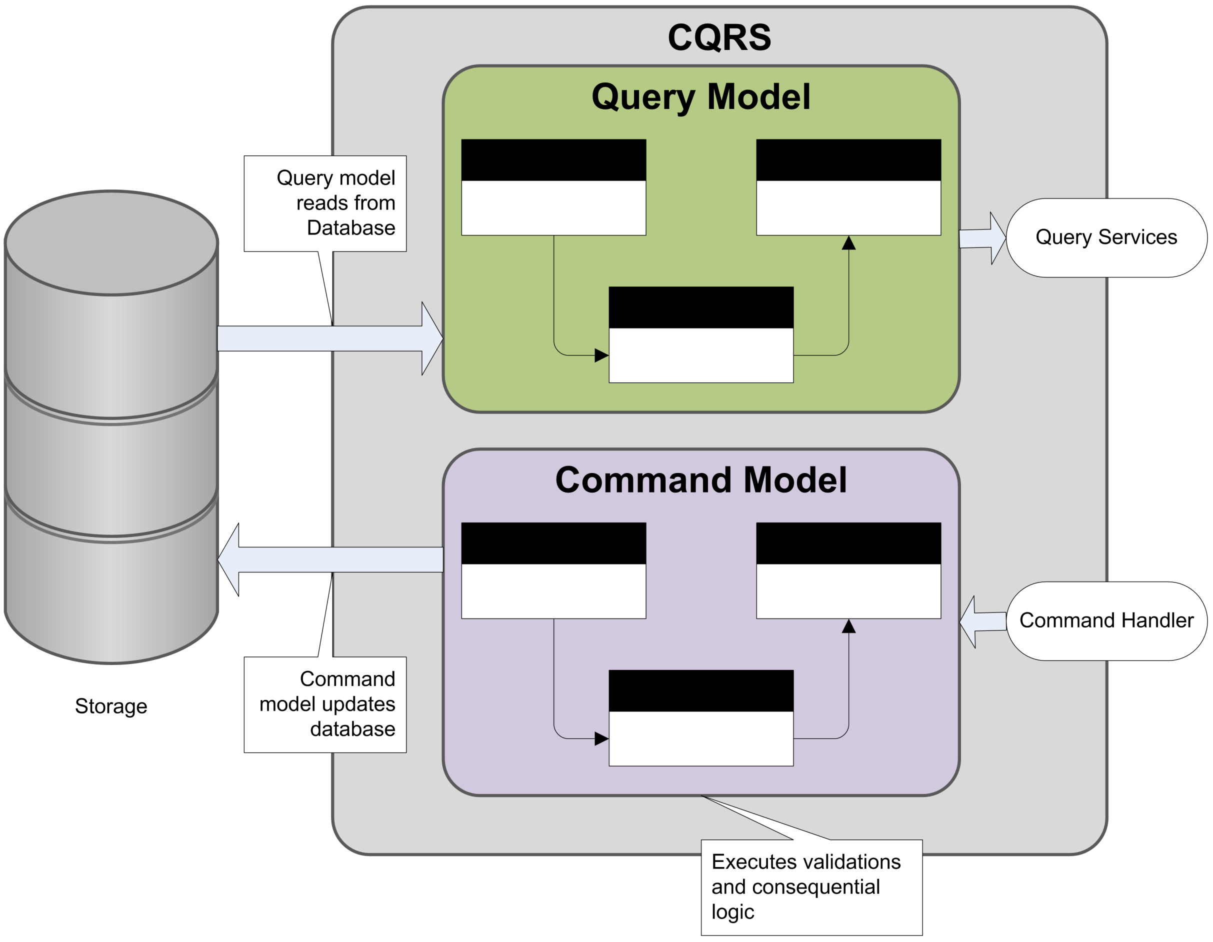
- Try-Cancel/Commit - By this pattern we can realize distributed transactions for micro services.

7 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

7.1 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.

7.2 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.

7.3 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.

7.4 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.

8 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 micro service 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.

8.1 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.

8.2 See also

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

9 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.

9.1 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.

9.2 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.

9.3 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.

9.4 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.

9.5 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.

9.5.1 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.

9.5.2 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.

9.5.3 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.

9.5.4 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.

9.5.5 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.

10 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.

11 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.

12 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.

13 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.

14 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.

14.1 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.

14.2 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).

14.3 CRUD operations

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

15 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.

16 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`.

17 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.

17.1 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.

18 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`.

19 Context mapping

Context Mapping

TODO:

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

20 Enterprise Integration Pattern

Enterprise Integration Pattern

TODO

21 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.

21.1 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.

21.2 Resources

- Exclusive Consumer within ApacheMQ
- Parallel exclusive consumers within ApacheMQ

22 Micro Service Architecture

Micro service architecture

Within a micro service 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.

Services are developed independently from another.

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

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 micro service 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 micro services. One approach is to partition services by use case. We could for example have a micro service shipping within a partitioned ecommerce 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 micro service design as well.

22.1 TODO

- Check whether OSGI could help with decreasing memory consumption.

22.2 Related Patterns

- API Gateway - defines how clients access the services in a micro service architecture.
- Client side discovery and Service side discovery - Patterns used to route requests for a client to an available micro service.
- 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 micro service easier.
- Service Statelessness - To have services scalable we should attempt to make them statelessness.

22.3 See also

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

23 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.

23.1 Related Patterns

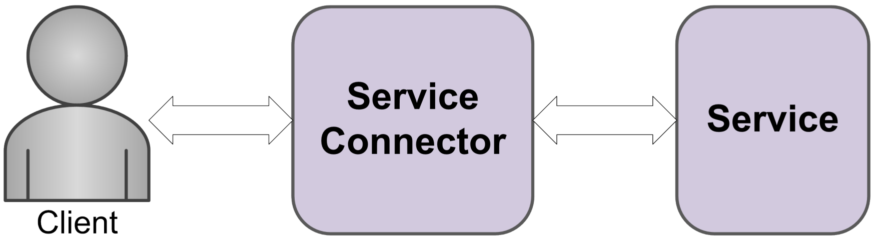
- Micro Service Architecture is an alternative to the monolithic architecture.

24 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

25 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.

26 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.

27 Abbreviations

Abbreviations

- CQRS - Command-Query Responsibility Segregation
- DBC - Design By Contract
- DIP - Dependency Inversion Principle
- DTO - Data Transfer Object
- REST - REpresentational State Transfer
- SRP - Single Responsible Principle
- TCC - Try-Cancel/Confirm