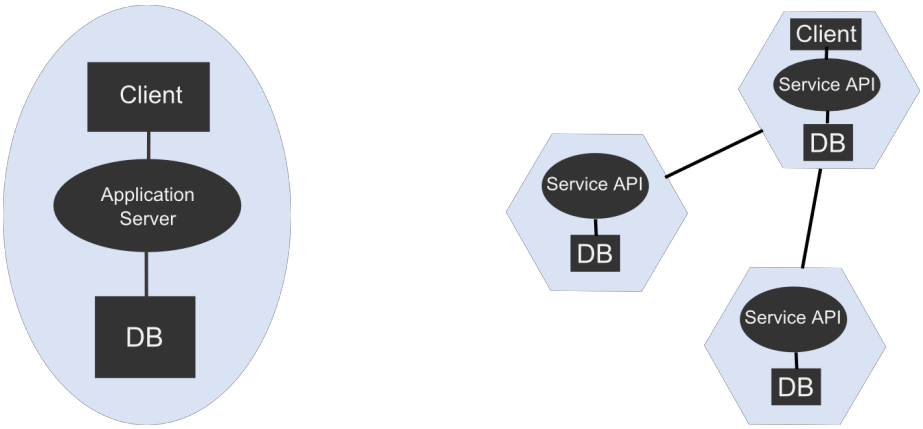
Microservice Workshop Position

Jørn Ølmheim, Einar Landre - Statoil ASA

Over the years as software developers we have seen a lot of programming trends and architectural styles come and go. Microservices is the latest of these that we think shows a lot of promise for certain use cases. While we have not been part of any projects that have actually made what would be considered a microservice, we have been involved in many explorations into using web technology.

Microservices

The Microservices architectural style is best described in contrast to the monolithic architecture that has historically been the way we build applications in our company.



The figure above shows the difference between a typical monolithic app on the left and a set of microservices on the right. The microservices are symbolized by hexagons to show that this is an example of a Hexagonal architecture as originally described by Alistair Cockburn. A typical monolithic app will contain all the logic and the data storage required to implement the functionality of the application, albeit modularized internally into components with explicit or implicit public api's depending on the developers competence and discipline. Microservices on the other hand are modularized in nature, having an inherent boundary around the component in form of an explicit public service api. This native modularization is the key to both the pros and cons of the microservice architecture, in that it makes it easier to maintain the functionality, easier to deploy and scale the services individually. At the same time it makes it harder to monitor and harder to ensure orchestration and communication between the components of the system.

A key question for me is why we seem to gravitate towards the monolithic architecture. Historically, the way to build an app was centered around a monolithic database. The earliest efforts used so called 4GL tools to create a relatively light presentation layer on top of the database which contained most of the business logic. This evolved into the three tiered application shown above, built in Java or .Net, with more of the application logic on the server, but still a great deal of emphasis on the database. Still to this day many developers are more knowledgeable in database design the in object oriented design, resulting in more database centric solutions. Even when we build web services, which are

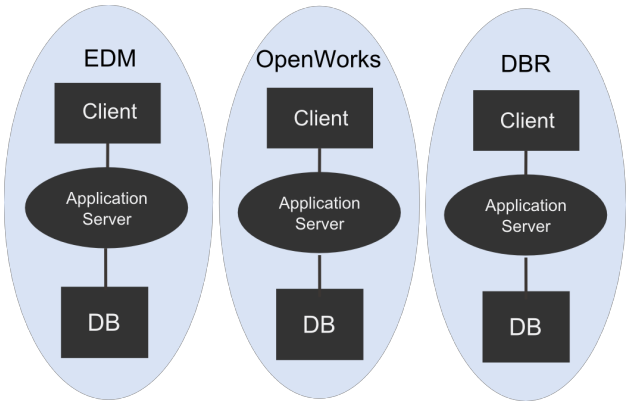
inherently suited to modularization, the end result is most often that we put multiple services or service endpoints in a single deployment unit.

We can think of many reasons why we seem to gravitate towards a monolithic design, chief among which is in our opinion the tendency to focus the design effort on the domain data and data storage over the domain dynamics and domain model. This is the effect of the general lack of knowledge about object oriented design and domain driven design in the average developer. This is slowly changing, and developers are getting more knowledgeable. That is especially the case with developers who are experimenting with web technologies, so why do we end up with monolithic design here as well?

The area in which we have done the most work with web technology and web services is in Enterprise Integration. The motivation has been to make all the important data that is currently locked in to monolithic enterprise applications more easily accessible for applications, business analysis and research projects.

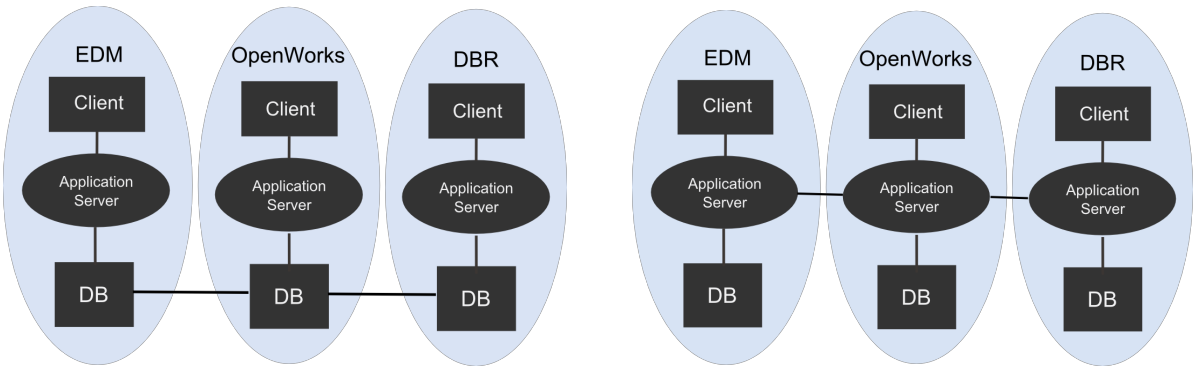
Enterprise Integration

Like any large enterprise we are saddled with a lot of monolithic applications covering varying parts of our value chain, some bespoke and some commercial, some built by large industry vendors and some built internally. The all share what we call a “stovepipe” architecture, with a huge database storing the data in a oftentimes convoluted model. Getting the data out of these systems is a challenge, and efforts often have to be repeated for each client.



In the best of cases, integration between these systems often follow one of two patterns:

1. Database to database communications using views or database links
2. Application server to application server communication using web services

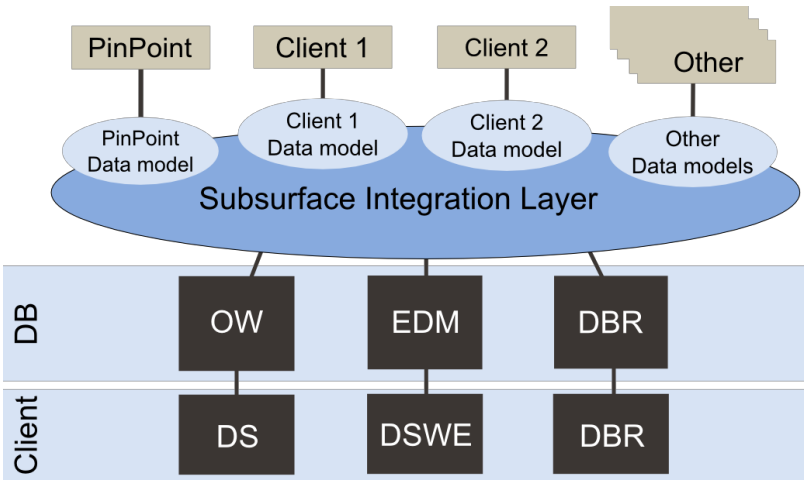


In reality, as I’m sure you must realise, like most enterprises we also have client to database and app server to database connections creating a veritable spaghetti architecture.

To alleviate this situation we have been experimenting with web services as a layer on top of the monolith databases as a new way of making the data within these systems available to other applications. We especially like the principle of keeping the logic in the end-points and the pipes dumb, in contrast with recent, and failed, brush-ups with Enterprise Service Bus architectures.

Subsurface Integration Layer

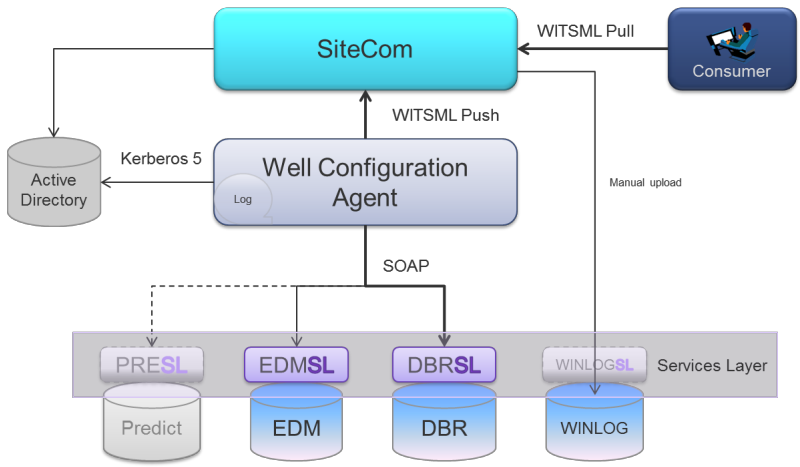
The subsurface integration layer in the figure below is our first attempt at this. The service consists of a series of connector and translation services, one for each of the monolith databases, and a central Statoil specific data model. On top of this we have implemented a set of client specific services that fetches data and presents it in a model that is adapted to the client.



There are a lot of considerations that has to be made when creating these services, most importantly authentication. We are trying to achieve single sign-on across all our apps, which in itself is a difficult undertaking. In addition it is important that authentication and access control is maintained from the client through the integration services and into the backend databases. This is not made any easier by the fact that most of the enterprise applications are monoliths not created to be accessed from other than their native clients.

Well Configuration Agent

Automated drilling is a relatively new application area that requires a lot of data from different sources. We have built a set of web services that gathers these data and pushes them to our central real-time datastore, SiteCom.

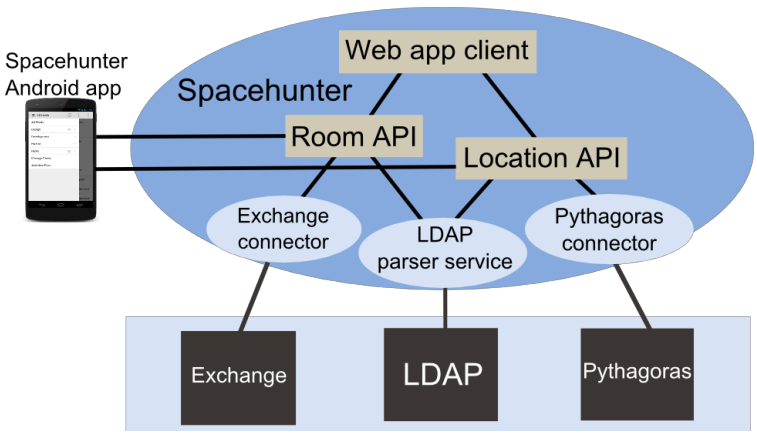


Here we are fetching data from some of the same monolithic databases that we are dealing with in the Subsurface Integration Layer. However, no reuse of the connector services have been achieved, and once again we have ended

up with a more or less monolithic web service that contains several components written specially for a specific client. This is partly due to the fact that the two products were made by different teams, but this does not explain the fact that both have created single deployment units containing services that are related but clearly have different reasons to change.

Spacehunter

Another small training exercise we have done is with booking of meeting rooms. Statoil is a heavy user of Microsoft Office and Exchange for email. Thus booking of meeting rooms for our meetings is integrated with the Exchange server and the Outlook calendar client. If you know about the meeting rooms and their cryptic names this can be a good and efficient solution, but especially for searching for rooms at a new or unfamiliar location this is not ideal.



There is a lot of available information about the meeting rooms in different sources. Exchange handles the booking information, that is when a room is free or booked, and details about the meetings. Our LDAP catalogue server contains an exhaustive list of all the meeting rooms with much of the information we need, for example whether there is video conferencing equipment and smartboards in the room. The Pythagoras system is the office space information system that has additional information like floor plans.

We have built two services, one that handles the room information including room metadata and availability, and a location service that handles information about the different office locations around the world.

Information from these two services are combined in two clients, one web based client and an android app. Both of these lets the users browse rooms by location, add favorites, search for available rooms at a specific location at a given time and find rooms close to the users current location.

Monolithic web service vs microservice

In all of the cases above we have ended up implementing what would potentially be multiple microservices as a single deployment unit, albeit for slightly different reasons. Sometimes it has been to make deployment easier, and sometimes there are technical complications like authorization that motivates the decision. Or is this simply a case of us not completely understanding our domain and our needs yet.

Should these have been made into separate microservice at once, or does it make more sense to keep them together as a larger module until a concrete need is there? Is there some clear advice on what to make into a separate microservice and what to bundle together into a bigger services? Is this just a case of us not knowing enough about the web technology and the supporting tools, or perhaps not enough about applying the principles about object oriented design and domain driven design to distributed systems?

A word about technology

We have been experimenting with Docker to run these services, with good results. Docker makes it easy to deploy the application and make sure that the variability between development, test and production server environments are as minimal as possible. Docker is in my mind one technology that tips the balance more in favor of separating out microservices, since it takes away most of the deployment and versioning burden. It does not, however, help with the orchestration, communication and monitoring that is required in a more distributed system.

Logging tools like Logstash, and monitoring tools like New Relic potentially make this easier. However, there are, legitimate or illegitimate, security concerns about putting system and application log data into the cloud.

Summary

In my view Microservices promises to be “web technology” done right. The architectural style uses the fundamental principles of the web to its advantage, and has the potential to achieve a lot of the capabilities that we lack especially in the area of enterprise integration. Organizing the software in small components with focused capabilities has the effect of avoiding the “kitchen sink” problem that many monolithic applications suffer from. That is: “We already have this system that does almost the same thing. Lets put the new functionality in there.” Having more fine grained components has both positive and negative effects, and it is important to consider all of these before choosing this particular style.

Having said that, we seem to gravitate naturally towards the monolithic style of developing apps and even web services. Partly this is lack of understanding of the benefits of the Microservice architecture, partly it is fear of the increased costs in terms of orchestration and monitoring that is required.

Still we think that web services and microservices are especially suited in the enterprise integration space, but we struggle with determining the granularity and figuring out which services to separate out and which to keep together. There is a clear tradeoff here between the benefits and the (at least perceived) increased cost of the microservice architecture.

Are we running the risk of the microservice architectural style becoming yet another way of incurring increased costs by prematurely structuring our architecture for future benefits that may not occur?