SOTI Data Analytics Service (DAS)

(SOTI Insights)

Technical Document - WORK IN PROGRESS

The Data Analytics Team

This document is intended for SOTI technical team members. This project is under development at a fast pace and information here is subject to change and not all features are %100 complete yet. Please, check the latest version. This is not a tutorial, we assume certain technical knowledge. For actual instructions on how to install the code read the README in GIT. For question related with this please contact pablo.elustondo@soti.net.

Note: a green background color is marking text recently updated or added.

What is this?

SOTI Insights, referred here as SOTI Data Analytics Service (DAS), will provide our customers with consolidated and customizable data sets, tools and processes to generate insight out of their contextualized historical or real time data using advanced metadata-driven tools and data science techniques.The application is intended to be used as a configurable, horizontally scalable and secure application that SOTI will primarily offer as a cloud multi-tenant but data-segregated service. It will leverage existing MobiControl data and collection processes but will not be restricted by those. The purpose of being fully metadata driven is to achieve fast run-around when implementing this solution to specific customers. This guide will review the business motivation, the conceptual design, the architecture, the current code and will suggest how to configure, run, deploy the application in the cloud.

High Level Business Goals and Use Cases 2

High Level Solution Concepts 4

High Level Architecture & Principles 10

Security & Privacy 14

Scalability & Performance Strategy 16

Cloud Compute & Deployment Strategy 18

Cloud Data Storage Strategy 20

Cloud Big Data Processing Strategy 20

Alerts, Notifications and Streaming Analytics 20

Cloud Continuous Deployment Strategy 22

Deeper into DAS Modules Functionality 22

Hands-on: Get, Understand and Run the Code 27

On Coding Standards and Quality Assurance 27

Annex - Technical Details - References 27

High Level Business Goals and Use Cases

The motivation for creating the SOTI Data Analytics Service (DAS) is based on the fact that our customers have expressed interest in an easy-to-consume solution that would help them consolidate and study the information generated by their mobile devices and other information sources and help them produce insights to make better business decision.

Besides the customer interest, we also know that this types of solutions are one of the areas of high growth and high profitability. SOTI Insights is both a tactical and strategical goal.

A good example of tactical business driver for this service would be a postal service, that expressed interest in battery performance facts and analytical insights to dispute mobile battery warranty claims or a chain restaurant that wants to have application popularity statistics for enhancing their in-store iPads user experience.

Most of these requirements imply collecting and keeping more rapidly moving and high volume information that normally exceeds the capacity of existing traditional relational database used by MobiControl. Also customers need very flexible and customizable ways to access that information and present it in a compelling way for easy monitor and analysis.

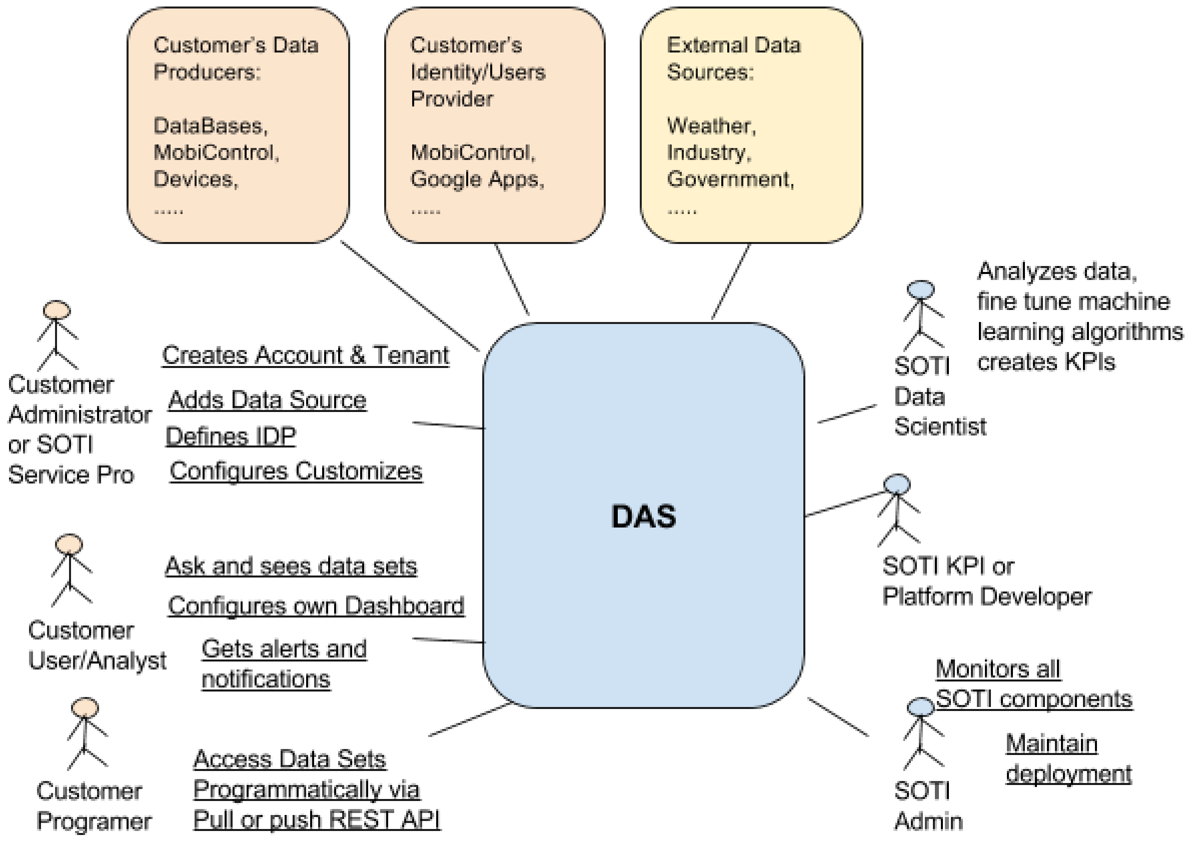
Fast time to production is important. If a customer wants to start analysing average device battery out of their mobile information, this process, which is somehow trivial a calculation,, should not need a special programming. Also, it should be fast. If the data collection agent is available this process should take virtually seconds. Of course, developing a smart KPI could take longer as somebody needs to analyze the problem and come up with the mathematical and practical approach to achieve this.

Providing a good customer solution involves both providing a secure and very scalable platform to get, store, transform and deliver information and providing specific pre-defined solutions. insights and data sets for specific business KPI. A big part of our project is related with developing those KPIs. However, this document only focuses on the platform.

Why would the customer chose us?. Because we believe that our expertise in device management related information combined with our skills in using and integrating disparate and complex innovative technology including cloud and databases put as in an unique position to provide an optimal and reliable solution.

Although we could offer this solution as a custom on-premise implementation, or as and hybrid on-premise-cloud, the intent is to deliver this solution as a multi-tenant data-segregated scalable cloud application that SOTI will offer as a service.

The following high level personas and use cases summarizes the basic functionality of DAS


Customer's Data Producers:
DataBases, MobiControl, Devices, ...
Customer's Identity/Users Provider
MobiControl, Google Apps, ...
External Data Sources:
Weather, Industry, Government, ...
Customer Administrator or SOTI Service Pro
Creates Account & Tenant
Adds Data Source
Defines IDP
Configures Customizes
Customer User/Analyst
Ask and sees data sets
Configures own Dashboard
Gets alerts and notifications
Customer Programer
Access Data Sets Programmatically via Pull or push REST API
DAS
SOTI Data Scientist
Analyzes data, fine tune machine learning algorithms creates KPIs
SOTI KPI or Platform Developer
SOTI Admin
Monitors all SOTI components
Maintain deployment

1. An Account Administrator or SOTI Service Pro can register its company with SOTI Services to get started. *(outside the scope of this document)*
2. An Administrator can create one or more "Tenants" within this account. A Tenant is essentially a segregated unit of data storage and computation, associated with an account, that ensures privacy and availability.
3. An Administrator adds an authentication/identity provider (IDP) to let users login into the application. *(only via Mobicontrol for now).*
4. An Administrator adds one or more 'data sources" that will feed its 'data lake'; for example a Mobicontrol instance, a data producing WEB API, etc.
5. An Administrator or Power User defines metadata that most users will use.
6. An Analytics User enters to the Dashboard for the first time and sees some initial configuration that gives him some basic information.
7. An Analytics User adds pages, charts, widgets, maps...etc.. to his/her dashboard.
8. An Analytics User defines dimensions, metrics, filters, reducers... (explained later)
9. An Analytics User sees its data sets both in snapshots or real time.
10. Any User setup alerts notifications to monitor specific conditions in prefered device.
11. Any User gets alerts . notifications to monitor specific conditions in prefered device.
12. A Programmer, use our Analytics API to feed its own dashboard or existing tools.
13. a SOTI Administrator can monitor the status of the system in terms of availability, performance, error handling, security, etc. Also he can increase or decrease resources, start and stop services, change security resources, maintain.
14. a SOTI Data Science Developer tries out arbitrary data-sets from our data-lake and analyses them using arbitrary statistical or machine learning packages and

languages. This can be Matlab, Octave, Zeppelin, etc. Algorithms created from this process, probably in python would be deployed to our operational cloud processes.

15. a SOTI KPI or Platform Developer uses development tools and platforms and access existing code and infrastructure to develop new KPIs to solve customer or industry specific business problems.

For more on business vision, drivers and future please see Tim Lee's Presentation

High Level Solution Concepts

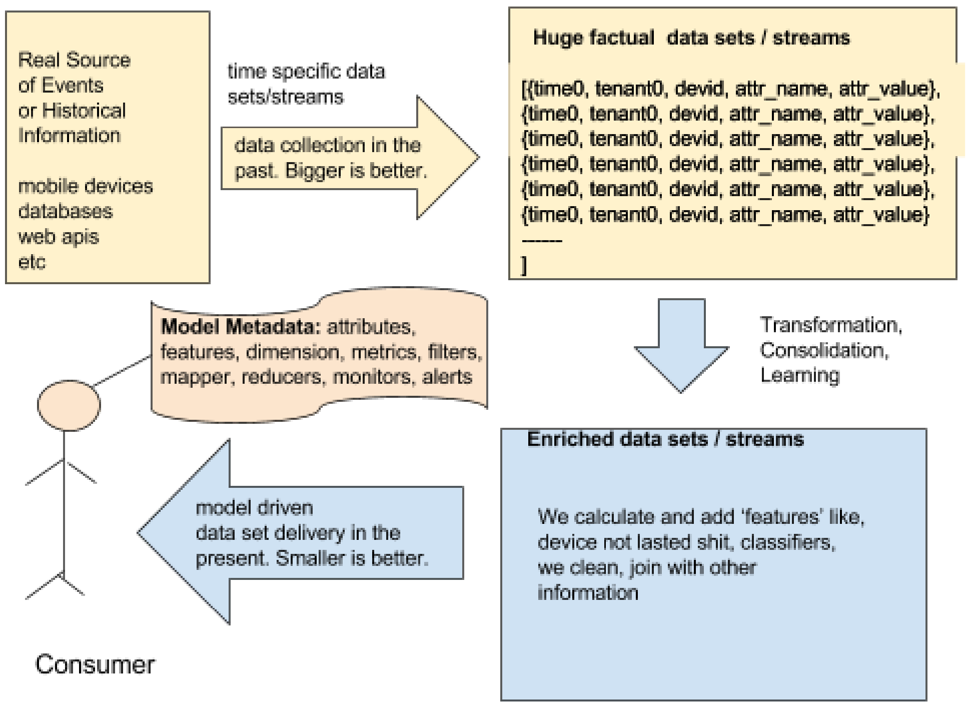
This section is a **review of very core concepts driving our solution**. Hopefully everything said in this section sounds obvious. The purpose is to clearly define our main concepts to fix a common domain language with precise terms and concepts; unambiguous definitions to define our requirements and solution. Our solution will target both historical and real time information. The concepts discussed in this section apply to both. A special section at the end will review specific concepts related with real time.

Besides typical business features such as account/tenant registration, user authentication, system configuration, etc, the core idea about our solution, from a mathematical points of view, boils down to one simple concept: **data sets.**

Everything we do is about getting, storing, transforming, interpreting, creating and enriching data sets. Dataset is essentially a list of objects where an object is a map of **attributes** into values. for example, a simple incoming data set may look like this example from mobicontrol device statistics table (dev_inst):

[{time0, tenantId, devid0, attr_name0, attr_value0, {some-complex-object}},
{time1, tenantId, devid1, attr_name1, attr_value1, {some-complex-object}},
.....]

Datasets can be anything, but to drive value of of them we need a way to interpret and manipulate and there is where our other key concepts comes: **Metadata.**


Real Source of Events or Historical Information
mobile devices
databases
web apis
etc
time specific data sets/streams
data collection in the past. Bigger is better.
Huge factual data sets / streams
[{time0, tenant0, devid, attr_name, attr_value},
{time0, tenant0, devid, attr_name, attr_value},
{time0, tenant0, devid, attr_name, attr_value},
{time0, tenant0, devid, attr_name, attr_value},
{time0, tenant0, devid, attr_name, attr_value},
{time0, tenant0, devid, attr_name, attr_value}

]
Model Metadata: attributes, features, dimension, metrics, filters, mapper, reducers, monitors, alerts
Transformation, Consolidation, Learning
Enriched data sets / streams
We calculate and add 'features' like, device not lasted shit, classifiers, we clean, join with other information
model driven data set delivery in the present. Smaller is better.
Consumer

Metadata gives as a way to interpret data sets and to define operations and transformations on them. For example we can say that the previous data set has one **"dimension"** based on the attribute device id, that way we can build say a chart of some information 'per device'. We can say that we have defined a **"metric"** that averages up all the values of some attribute such as 'battery' and gives the average battery. In general a 'metric' is a function that can be applied to an atomic data point and that can be **'aggregated'** in a data set. Dimensions and Metrics do not exist in the data per see, their are a fabrications, a way to "inject semantics" into a data space. **Data only has 'attributes'.** We will explain this in a more concrete way to see this in action when we discuss our prototypical consuming tool, the dashboard.

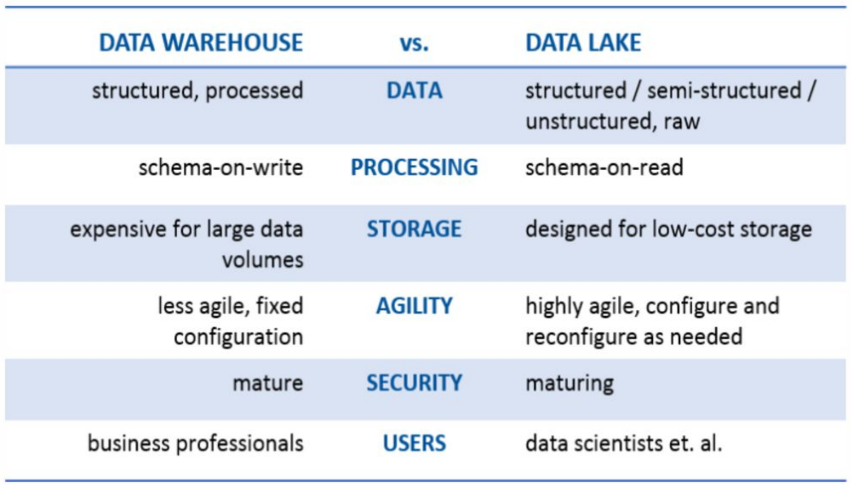
In machine learning parlance, we will call some of those attributes, dimensions, or metrics **"features".** This is pretty much just a different name, but it carries semantic meaning for a specific hypothetical model.

When a user analyze data sets and try to make sense of them, the user may **reduce** columns using "**dimensions**". He can also **"filter"** data using certain filters and enrich its data creating new "**metrics**" out of its data. Those 3 are some of our most common **data set operations** are **map**, **reduce** and **filter.** And although it can mathematically being defined in terms of the previous one, the **merge** operation is also a fundamental one. A merge operation will produce one consolidated dataset out of two input dataset by mapping data points from both sets into a common feature. attribute, dimension, or metric).

More formally,
Filter just transform a data set into a smaller sub data set by applying a predicate on its data points. Mathematically, a filter f for a data set universe D is a function f:D -> Bool. Since we

use high order programming languages like Javascript and Python, programmatically, ‘f’ will be a boolean expression that refers to attribute names in D.
Map, creates a new data sets with the same information as the original one but where all the data points are grouped by the data points of a given dimension.
Reduce, given a dataset and a ‘metric’ that uses an ‘aggregator’, it calculates the end result by aggregating all the values of the records using the aggregator. For example, if the aggregator is the addition, the reduction will be the sum of all the records
MapReduce, is just applying the previous two but we normally refer to it as a shorthand of mapping a data set to a dimension/s and then reducing the record set applying the aggregators defined in the metrics.
Merge, will ‘merge two’ data sets using a common attribute or dimension that allows to build a unified data point. Merge can de defined in terms of map but is handy to refer to it.

So, using metadata we can have many models to interprete our data. There is not “one data model’, like we use to do in classic relational data modeling. In our solution we have a **data lake**, which is a probably unstructured set of data sets and we use **multiple metadata models** to interpret this data in different ways. Every single user can decide which is the model he wants to see and manipulate. He uses metadata also to decide which information to collect from our sources, .which is this information coming from, etc. Just to summarize the idea of ‘data lake’ in contrast of ‘data warehouse’, the next figure may help.



DATA WAREHOUSE	vs.	DATA LAKE
structured, processed	DATA	structured / semi-structured / unstructured, raw
schema-on-write	PROCESSING	schema-on-read
expensive for large data volumes	STORAGE	designed for low-cost storage
less agile, fixed configuration	AGILITY	highly agile, configure and reconfigure as needed
mature	SECURITY	maturing
business professionals	USERS	data scientists et. al.

Key Data Set Types: Streams, Logs, Matrices

Two important subtypes of data sets are **streams, logs** and **matrices**.
Streams a time-based data sets with, normally, homogeneous data points. We can also call **streams ‘time series’ most of our datasets will be streams.** *In principle, all of our dataset can be interpreted as streams. Even the most static type data set like ‘list of provinces’ should have a timestamp and may need an update one day. Whether we keep the old version in the history or not is another subject, but yes, it can be seen as a stream.*

Matrices are a more specific type of data-sets that can be manipulated with matrix algebra which are normally used in various machine learning algorithms. Matrics deserve a specific treatment and storage strategy. Use of matrices is still not being considered at the implementation level in this project so we will not be covering this in htis article.

Logs are just streams of a disparate information. Streams tend to have homogenous attributes and therefore are easier to interpret with fixed metadata. Logs tend to be very eclectic and probably only text based queries make sense. We are also not covering logs processing for now.

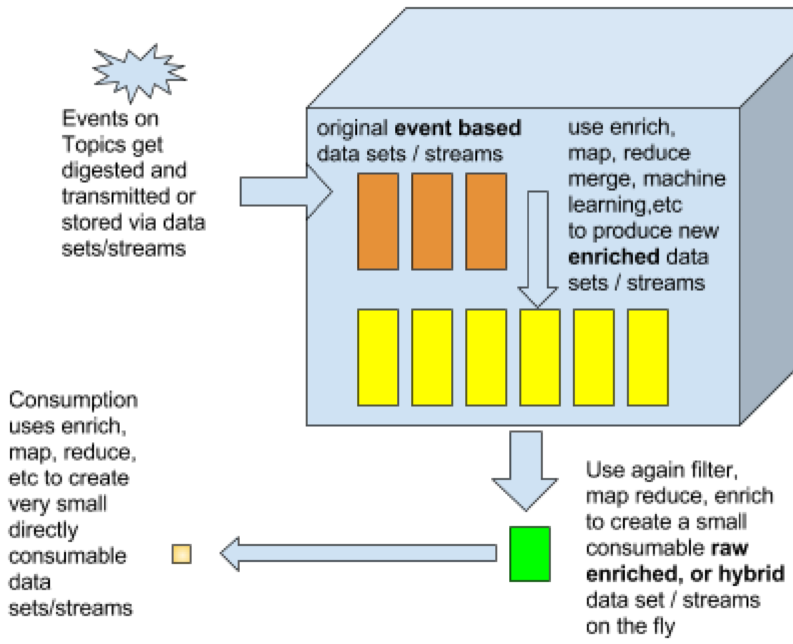
We will use data science technical including machine learning to transform certain data sets to create enriched data sets with new features like **extrapolations**, **classifiers,** etc.
Some data sets will be **matrices** to allow special computations such for example the normal equation for multivariate linear regression.

The Genesis Stream: Topics and Events.

Up to know we have said that SOTI Insights is about capturing, storing, processing and delivering streams (data sets) on behalf of our customer interests. But, **Where do streams come from?.**

Streams come from paying attention to "events" related to certain "topics".

Conceptually, our whole system is about letting tenants define subscriptions to certain events related with topics, let those events feed defined streams / data sets that will be probably enriched and stored, and then reduce those stream to delivered them to consumers in the forms of streams / data sets again.


Events on Topics get digested and transmitted or stored via data sets/streams
original **event based** data sets / streams
use enrich, map, reduce merge, machine learning,etc to produce new **enriched** data sets / streams
Consumption uses enrich, map, reduce, etc to create very small directly consumable data sets/streams
Use again filter, map reduce, enrich to create a small consumable **raw enriched, or hybrid** data set / streams on the fly

Putting it all together

Example

Suppose a new fresh tenant just wants to see information about devices managed by mobicontrol located in buses. He wants to know the battery level and the position of the device.

Currently this information will come from two different sources. The bus location from the bus service and the battery level from mobicontrol.So we can define that we wants to care about two topics, say, “buslocation” and “deviceinfo”. Let assume those events on those topics are coming to mobicontrol to our input data adapter IDA. (how this happens is another subject).

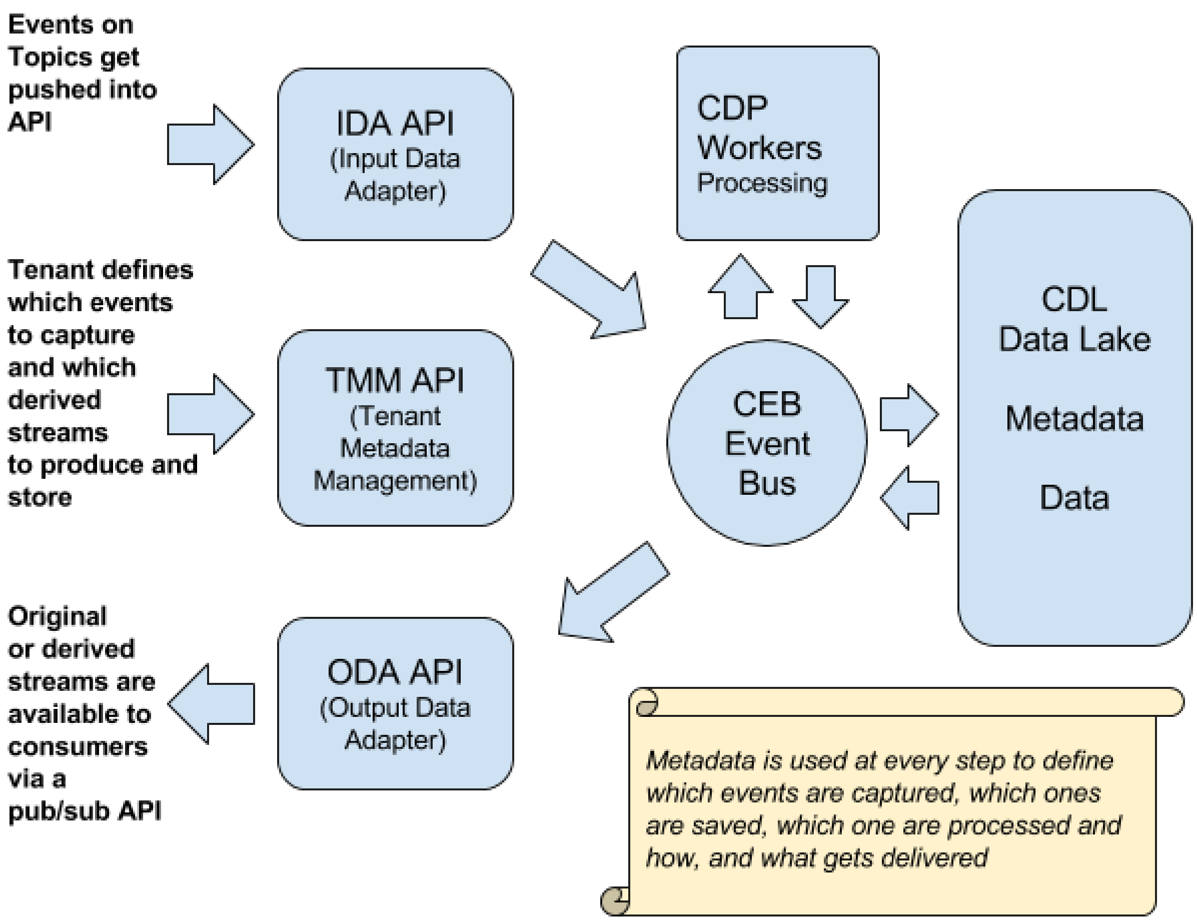
So, the tenant will got to the tenant metadata manager and specifies that he wants to have these two streams available in the data lake.The metadata for this tenant (that will be saved on the tenant database) will contain something like this

```
[{
        dataSetId:”buslocation”,
        dataSetName:”Bus Location”,
        dataSetType:”StoredStream”,   //stream history is stored in datalake
        attributes: [
                {attributeName: “busid“, attributeType:”int”},
                {attributeName: “deviceid“, attributeType:”int”},
                {attributeName: “lat“, attributeType:”float”},
                {attributeName: “lon“, attributeType:”float”}]
},{
        dataSetId:”deviceinfo”,
        dataSetName:”Device Information”,
        dataSetType:”StoredStream”,   //stream history is stored in datalake
        attributes: [
              {attributeName: “deviceid“, attributeType:”int”},
              {attributeName: “batteryLevel“, attributeType:”float”}]
}]
```

As long as he does this, and after some minimum delay, he will be able to call the output api ODA and start getting information about this two streams. If he called the API as a subscriber he will be getting those two streams separated. We could use this info to plot buses on a map or to show battery level by device id.If we wants to see battery level by location he will need to define a third data set that links this data together. Now, here we have a few alternatives. We can link this data together at the UI side, a the API side, at the database side. To simplify let say we will calculate this in database and store the result. So, our third data set will be something like this:

```
{
        dataSetId:”deviceinfoandlocation”,
        dataSetName:”Device Info and  Location”,
        dataSetType:”StoredStream”,   //stream history is stored in datalake
        attributes: [{attributeName: “busid“, attributeType:”int”},
                {attributeName: “deviceid“, attributeType:”int”},
                {attributeName: “lat“, attributeType:”float”},
                {attributeName: “lon“, attributeType:”float”},
                {attributeName: “battery:evel“, attributeType:”float”}]
}
```

All these steps, capturing events, storing and or transmitting events and delivering will be govern my metadata. The next diagrams show the first components in our solution that will be in charge of this processed without going into details on aspects like security, user interface or internal details.



Events on Topics get pushed into API
IDA API (Input Data Adapter)
CDP Workers Processing
Tenant defines which events to capture and which derived streams to produce and store
TMM API (Tenant Metadata Management)
CEB Event Bus
CDL Data Lake
Metadata
Data
Original or derived streams are available to consumers via a pub/sub API
ODA API (Output Data Adapter)
Metadata is used at every step to define which events are captured, which ones are saved, which one are processed and how, and what gets delivered

Streams Velocity, Frequency and Delay

Interests streams will move at certain **velocity (bits/sec) and** maybe **predictable frequency** and carry them from end to end will imply certain **delay.**

Velocity talks about how many data points we send per second and that can always been measure at least on average. **Delay** is about a shift between the actual point in time of the **sensor actual data collection and the actual information delivery.** A typical example of delay is the few seconds delay in satellite TV, which still support a great velocity and frequency. Most of the time delays are acceptable and even desirable for quality control.

Real frequency at the sensor level only applies to cyclical signals, but normally what we are only interested in is **‘sampling rate’**, which is the **‘desired frequency’.**
However, Some industries, such as intraday stock trading, will be interested in shortest possible delays,

So, to set requirements for our data collection here we need to specify:

- desired **Velocity Capacity** (Bits/sec)
- possible **Desired Sampling Frequency** (1/sec)
- desired **Delay** (sec)

We will go back to this deeper in a our section on **Streaming Analytics Implementation Strategy.**

Alerts and Notifications

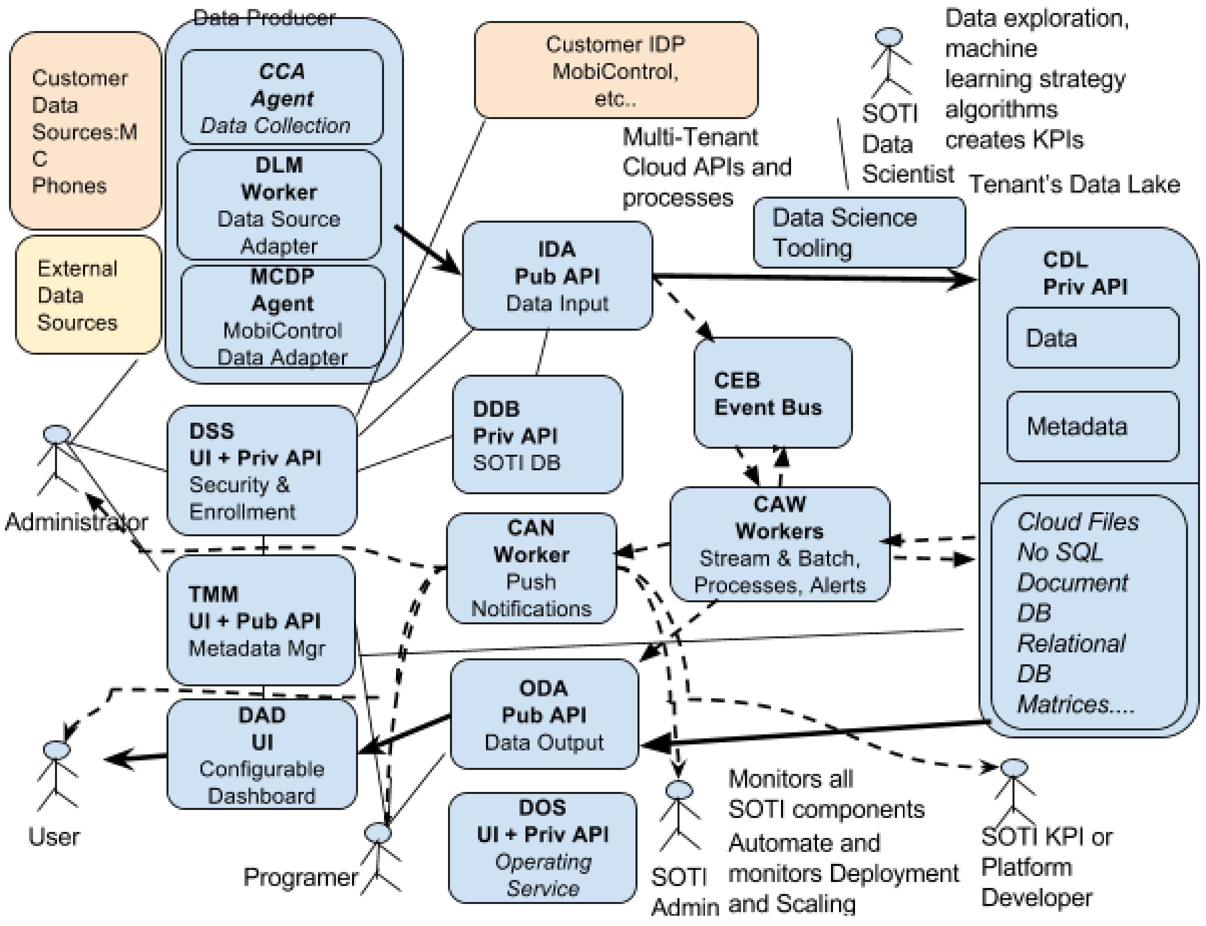
From a data set perspective, alerts and notifications are just another derived type of data sets that are created after a condition is detected. That condition is detected by a boolean function that defines when a data set has reached certain situation. However, the management and operation of alerts and notifications presents a special problem domain as alerts should be calculated as soon as possible and being delivered to the consumer in a convenient and reliable way. We will go back to this deeper in a our section on **Alerts and notifications.**

High Level Architecture & Principles

This section is an overview of the current logical decomposition of DAS into specific components following the single responsibility principle. Logical decomposition must not be confused with deployment architecture; a box in this diagram does not mean a server or container in production environment. DAS is an microservices-oriented application that may be deployed as FaaS, or PaaS, or IaaS. We discuss this later in cloud deployment strategy.

Also, logical decomposition should not be confused with complete custom implementation. Some modules, like the event bus, for example, will be implemented with some standard existing tools. However, we always will create a facade to hide that dependencies and follow the **principle of programming to an interface**. Tools and implementations may change in the future, our segregated interfaces should. hopefully, stay. Also, adapters create a shield against vendor lock-in as they encapsulate certain functions that may be rewritten.Components can evolve independently and updated without affecting others.

All modules are multi-platform multi-tenant with minimalistic code using modern open languages such as node.js and others. Components communicate via http or event queues. **Bold arrows show natural synchronous data flow. Dotted lines show events flow.**


Customer Data Sources:MC Phones
External Data Sources
~~Data Producer~~
CCA Agent Data Collection
DLM Worker Data Source Adapter
MCDP Agent MobiControl Data Adapter
Customer IDP MobiControl, etc..
Multi-Tenant Cloud APIs and processes
SOTI Data Scientist
Data exploration, machine learning strategy algorithms creates KPIs
Tenant's Data Lake
Data Science Tooling
IDA Pub API Data Input
CDL Priv API
Data
Metadata
CEB Event Bus
DDB Priv API SOTI DB
DSS UI + Priv API Security & Enrollment
Administrator
CAW Workers Stream & Batch, Processes, Alerts
CAN Worker Push Notifications
Cloud Files No SQL Document DB Relational DB Matrices....
TMM UI + Pub API Metadata Mgr
ODA Pub API Data Output
DAD UI Configurable Dashboard
User
Programer
DOS UI + Priv API Operating Service
SOTI Admin
Monitors all SOTI components Automate and monitors Deployment and Scaling
SOTI KPI or Platform Developer

The previous diagram shows the most relevant modules in our system and the most relevant communication needs. For simplicity not all possible communication lines has been drawn. We will go into details of each component and its dependencies in later sections. This sections just gives a very high level overview.

DSS satisfy the uses cases for **Administrators** mentioned before; modules to create and administer one or more “Tenants” within this account. A Tenant is essentially a segregated unit of data storage and computation, associated with an account, that ensures privacy and availability. The **DSS** modules provides both UI and API to enroll a tenant. A Administrator uses DSS to add an authentication/identity provider (IDP) so user can login into the application. (for now only via Mobicontrol). DSS has been architected into a UI DSS(UI) and a private backend DSS(BACK) part for security reasons.

TMM lets administrators to define data sources and data sets. An Administrator uses TMM to add one or more ‘data sources” that will feed its ‘data lake’. DSS has been architected into a UI TMM(UI) and a public backend TMM(API) to let customers to define and manage their metadata both in UI and programmatically.

DAD is a UI module that an analytics user uses to see its configured Dashboard for first time and sees some very basic initial metadata configuration that gives him basic information. Then, An Analytics User uses DAD to add pages, charts, widgets, maps...etc.. to his dashboard. An Analytics User uses DAD set up alerts notifications to monitor specific conditions.

DSS, TMM and DAD are the user interfaces to our customers.

PAN is the extra way to reach our consumers. it is in charge of pushing alerts and notifications using a push mechanism. Although some ‘alerts’ may be seeing by consumers when using the UIs, PAN will make sure they get a notification even if they are not using the UI. Notifications may be send in the form of emails, mobile notifications,text messages,etc.

ODA is a data adapter / facade that provides all the information to DAD but also to external API users. .ODA is just a secure data adapter that accepts data requests from DAD or http requests and responds from information coming from the data lake. ODA communicates with DSS to get a token that will be sent to ODA to get information. A Programmer, use our ODA to access our Analytics API to develop its own dashboard or to connect to their existing data analytics tools. **IDA** is another adapter, this time to let data in our system.

DAD, TMM(UI), DSS(UI), TMM(API), ODA, IDA and are our internet exposed modules. Also, these components are 100% vendor neutral and support both historical and real time metadata driven data sets.

CDL is our APIs to our “Data Lake”. It resolves data extraction and update translating data definitions written in our metadata language into the corresponding data lake implementations. ODA and IDA will communicate with CDL to get and put information.
CDL encapsulates the specific data stores Currently, the idea is to use a document database (MongoDB) and also to S3 cloud files for logging. When considered convenient, information may also be consolidated into a SQL relational data, RedShift, store for more complex analytics. However, this tools can change.

Although directly synchronous communication is ok when parties know each other in advance and call can be resolved quickly, for the general case of longer processes or communication cases when parties do not know each other, the best communication method is an event bus. **CEB is the module fulfilling that role**. CEB allows an application to publish a stream of records to one or more ‘topics’ and allows an application to subscribe to one or more topics and process the stream of records produced to them.

CAW represents/manages a family of asynchronous workers process / processes that takes care of longer and asynchronous tasks. CAW may process data based on events or based on schedule. CAW can subscribe to events from a topic and act as a *stream processor*, consuming an input stream from one or more topics and producing an output stream to one or more output topics. It can also act from events performing long running jobs.

DDB is an API wrapper for our SOTI central document database to store tenant configuration and other critical information to run DAS as a multitenant application. CDB document database is supposed to be small and probably in memory.

In production deployments, all SOTI components will be monitored and managed by our monitoring service **DOS.** This is just a service that can talk to all the other modules, ask for status, and probably act on them on behalf of administrators.

MCDP is a specific data collection agent done for mobicontrol to collect data from mobicontrol.

Modules Roles

The previous diagrams shows a module ‘role’ under the name of the module. We have identified the following roles:

1. **Web App**
   This is the typical web application that in our case is mostly client side angular. They normally have a simple server that serves the page and act as a dependencies proxy.
2. **Web API Public**
   This is the typical web API that you can call using http or connect using sockets to get updated. Describes its interface with swagger. They normally have a simple server that serves the page and act as a dependencies proxy.
3. **Web API Private**
   Same as before but not exposed to internet.
4. **Worker**
   This is a server that will act normally on a schedule or an event to perform some tasks probably calling other modules, creating information in date lake and rising events.
5. **Event Bus**
   This is a server process that accepts and emits events creating a way to asynchronously communicate a set of event producer and consumers.
6. **Agent**
   This is a process that also acts on schedule or events to collect data from the host machine and send information to DAS.
   Besides having different functional goals, this different types of modules will be deployed in a different way. We will discuss this in the deployment section.

Opinionated Architecture / Design Principles

Although we have suggested some of our core principles before, we will use this subsection to summary key architecture / design principles that guide our solution. The order in which the principles appear is random. All of them are important. Although we try to follow common ‘best practices’, all architecture are somehow opinionated in nature and this one is not an exception.The principles expressed in this section help the reader to understand some motivation for some decisions. We list them first and then we go into more details. Fell free to skip this section if you are not too interested in our opinions.

Conceptual Integrity
Component Based
Metadata Driven
Collect First, Interpret Later
Low Vendor Lock-In (AWS first, but exit strategy)
Program to Interface
Wrap Dependencies
Microservices Oriented
Horizontal Scalability
Isolated Development
    Everything runs on small local environment
Right Layer, Right Paradigm
KISS SOLID Patterns
Pragmatic BDD
As Open as Possible, or Adapters

As Standard as Possible

A good example of standard is our node.js application. To get and run it is always, git clone, npm install, npm start. Period. Our code should run 'out of the box', by default mocking or simulating its dependencies You can work and test components independently. You can develop code independently. All our components can run in 'test' mode with sample data and components in a very simple and natural way.

As Open as Possible

The dichotomy between proprietary and open systems has always been there for a reason. In the long run is normally better to use open system that make us independent from a vendor. On the other side, innovation normally moves proprietary first. Only using standards may result in delayed innovation or use unreliable layers that impact performance and maintainability. The risks and problems of going too proprietary are obvious.
So, in this context, our strategy is. "as open as possible". , THat means that we will privilege a standard way of doing something. When that is not convenient we will minimize the impact by creating adapter or layers to encapsulate the dependency. And interesting example is now the notion of serverless computing. AWS moved first, Google and Microsoft are following. Fortunately, a framework is starting to appear to deploy node.js in those 3 platforms form a common ground.

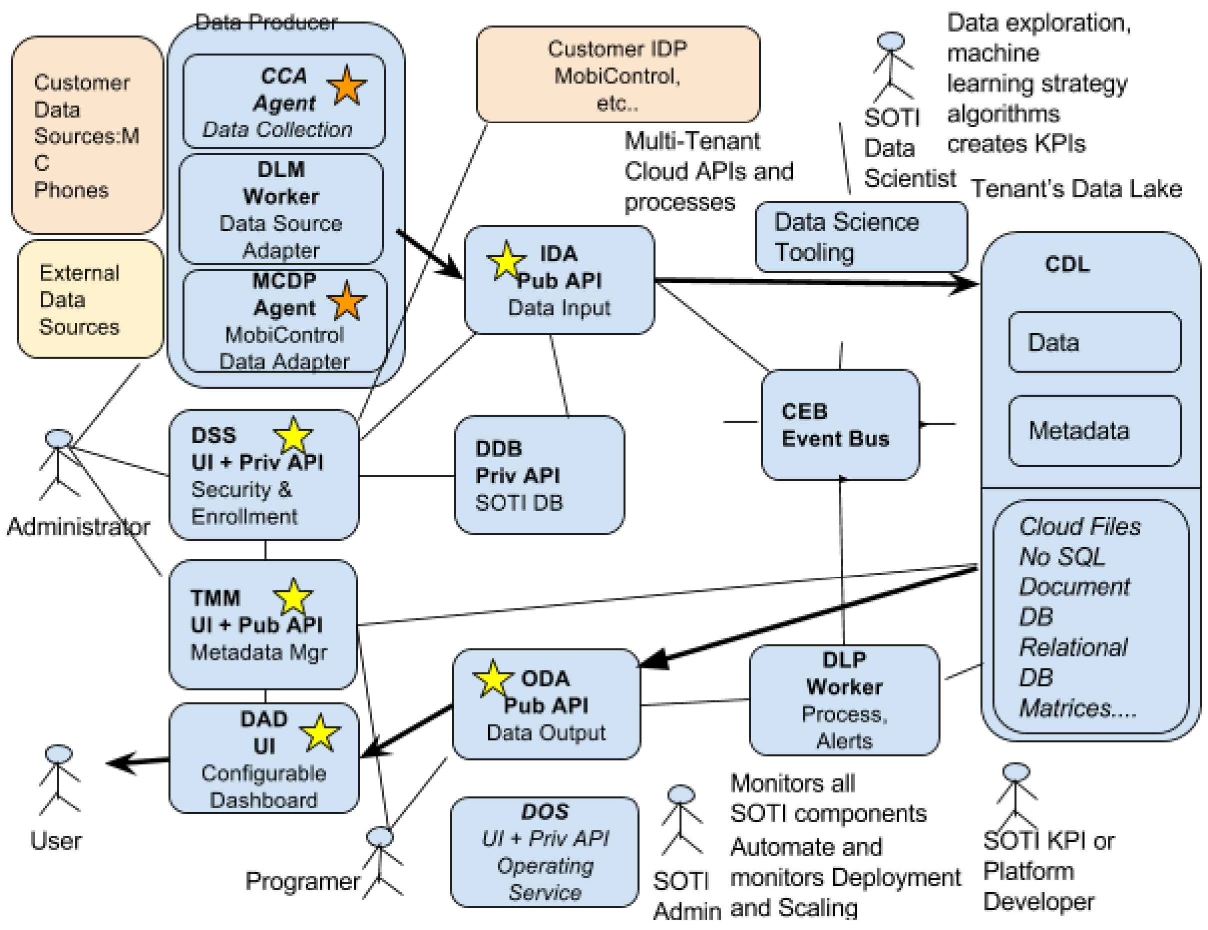
Security & Privacy

As we shown in our wiring chart before, DAS system is based on a set of multi tenant communicating components that access a tenant-based physically segregated data lake. This section will review our strategy for solving different aspects related with security and privacy. I will use italics when the aspect is already being discussed but no implementation still available.The next diagram shows the the vulnerability risk of our different modules if they are deployed optimally in a cloud environment.

Note: what we mention here are general concepts and should not be interpreted in terms of implementation concepts. Certificates, for example, are going to be used. But the way this is going to happen will be different if we use One Premise, IaaS, PaaS or FaaS deployment strategies.

Security Exposures

The next diagram show the previously presented modules marking the exposures. Modules with a yellow star are exposed to internet but in our control, modules with red start and exposed to internet and not in our control.



Customer Data Sources:MC Phones
External Data Sources
Data Producer
CCA Agent Data Collection
DLM Worker Data Source Adapter
MCDP Agent MobiControl Data Adapter
Customer IDP MobiControl, etc..
Multi-Tenant Cloud APIs and processes
SOTI Data Scientist
Data exploration, machine learning strategy algorithms creates KPIs
Tenant's Data Lake
Data Science Tooling
IDA Pub API Data Input
CDL
Data
Metadata
Cloud Files No SQL Document DB Relational DB Matrices....
Administrator
DSS UI + Priv API Security & Enrollment
DDB Priv API SOTI DB
CEB Event Bus
TMM UI + Pub API Metadata Mgr
ODA Pub API Data Output
DLP Worker Process, Alerts
DAD UI Configurable Dashboard
User
Programer
DOS UI + Priv API Operating Service
SOTI Admin
Monitors all SOTI components Automate and monitors Deployment and Scaling
SOTI KPI or Platform Developer

SSL and Certificates

All our components use https to communicate between then and the the outside. Our server will present a certificate based on *.das.soti.net.
In case you want to install and test the system in a SOTI machine without having this certificate, we show how to do this in a SOTI machine in an Annex.

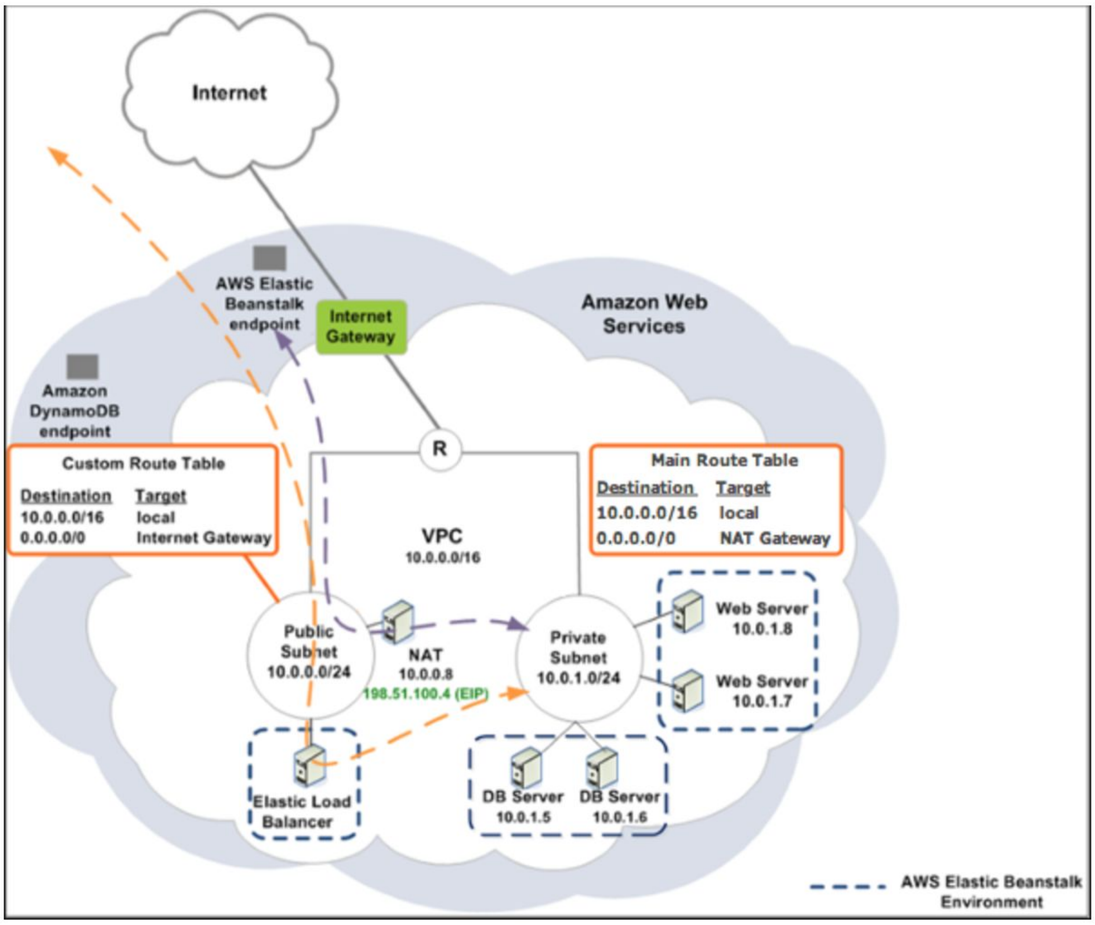
When possible, certificates client certificates will also be validated. This is possible to do in our SOTI to SOTI communications. We have propose also that, when possible, maybe validating our customer client certificate would be used.

Note: Until we implement client certificate validation we are using shared secrets and key to validate authenticity of SOTI to SOTI communication.

Virtual Private Networks

As we suggested in the diagram, only certain components should be exposed to the internet, the rest should only be accessible from our trusted services inside a virtual private network.

For example, here is a suggested deployment strategy using Amazon Elastic Beanstalk layer that we will introduce in the deployment section;



User Authentication

DAS has been designed from scratch from the idea that user authentication should be provided by an external independent entity or ‘identity provider”. Although we are going to store user-based information and configurations, we are not going to manage user profiles or credentials. We are assuming customers will use their existing way to authenticate users. For the moment we have only implemented this using MobiControl as a identity provider. Customers that already own MobiControl can register their instance to authenticate theirs users. (MobiControl maybe also configured to authenticate user to an external identity provider as well causing a further delegation to say, OneLogin, Google, etc.

Sessionless JWT tokens and shared secret

When an external user or system wants to get access to our dashboard or api, they will end up redirected to our DSS modules that is in charge of handling security. We will go more in details about DSS later but for now let’s mention that DSS will issue an expiring signed token with basic information like “tenant id”, ‘“user”, etc. This token will be exchanged for information from data services like ODA. Our applications share a general secret to sign and verify this token. The advantage of using stateless signed tokens is that this allow us for massive scalability and multi tenancy.

Data Segregation

Although our application is multi tenant, it could also run on a customer bases, (we will discuss this also when showing strategies to deploy. However, only one thing is not multi tenant and that is the customer data.

Customers data reside in account

Access Control

This feature is still under development. User authentication will ensure that user belongs to a specific tenant and by doing so we know they can only access data for that tenant something enforced at the lower level by data segregation. Now, to specifically access a portion of the tenants data according to some access control rule, our initial idea is to use the notion of data access configuration that we will present for the dashboard. A data access configuration file clearly defined which data sets a user can access in detail. Depending on the user access control level, he will be able to create more or less powerful data access configurations.

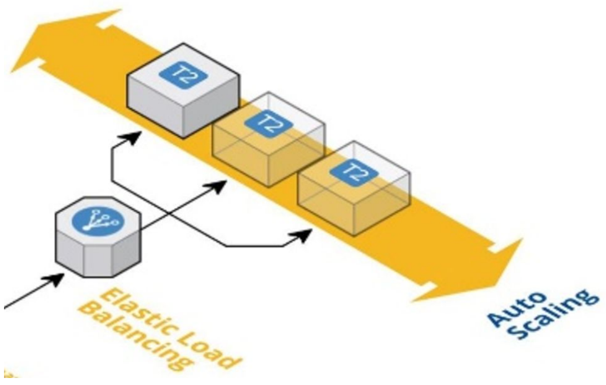
Scalability & Performance Strategy

This section is a very high level view on our basic strategies to deal with scalability and performance of our application while keeping a low cost and administration burden. The key scalability / performance problem is normally related with these aspects:

1. **Web/API frontends:** How to handle large number of users and request in our internet facing and supporting modules
2. **Data Stores**: How to grow our data lakes
3. **Large Batch Processing:** How to batch process large data sets using probably complex algorithms.(Scale Batch Data Processing)
4. **Real Time Processing:** How to deal with real time stream
5. **Push Alert and notifications:** how to maintain a large number of actives connections to push notifications and alerts.

Web/API frontends

For scaling our web apps our strategy is relying on horizontal scalability based on the use of very light multi platform node.js servers that can run in small linux machines. Using AWS elastic platform, probably Elastic beanstalk, we will deploy our node.js servers with load balancer and parallel small servers.



Data Stores
For scaling our data lake, we will use various scalable methods. Two of most important are (S3) Cloud File System , which are totally managed by AWS and will scale well and a the document database MongoDb.

For MongoDb finding the right strategy is much more delicate and we can barely cover here the basics. The idea is that MongoDB provides native replication to ensure availability;

auto-sharding to uniformly distribute data across servers;and in-memory computing to provide high performance.

However, first of all we should consider **vertical scalability** to a certain extent. Sharding, the method for **horizontal scalability** is a more complicated and costly strategy for db is should be minimized. When vertical scalability has reached a top, sharding is involved.

Vertical Scaling for for MongoDB

So, first of all we need **good servers**. **RAM size** is the most important factor for hardware; Next important aspect will use **SSDs** for write-heavy applications.Most MongoDB deployments should use **RAID-10.** MongoDB will deliver better performance on **faster CPUs.**Configure **compression for storage** and I/O-intensive workloads. MongoDB natively supports compression when using the WiredTiger storage engine. Compression reduces storage footprint by as much as 80%.
For best performance, users should run one mongod process per host.
More information on this subject please refer to MongoDb Performance Best Practices.

Horizontal Scaling for for MongoDB

Cloud Compute & Deployment Strategy

Although DAD can be deployed on premises and we actually do this for development and testing purposes the target production environment is a virtualized scalable cloud environment. . Architecting and deploying an application like DAS on a cloud environment involve thinking on various subjects that can be implemented in a fews different ways. We will focus here on the compute strategy for our web and basic ‘traditional’ application which we have written in pure node.js. Based on a our principle mentioned before keep a low vendor lock in while keeping it simple, reliable and with good performance.

Most services offered by cloud providers are similar en nature and most of them are based on open source libraries that could be deployed. Hadoop and spark for example are used by most cloud provider in their ‘data analytics’ solutions.

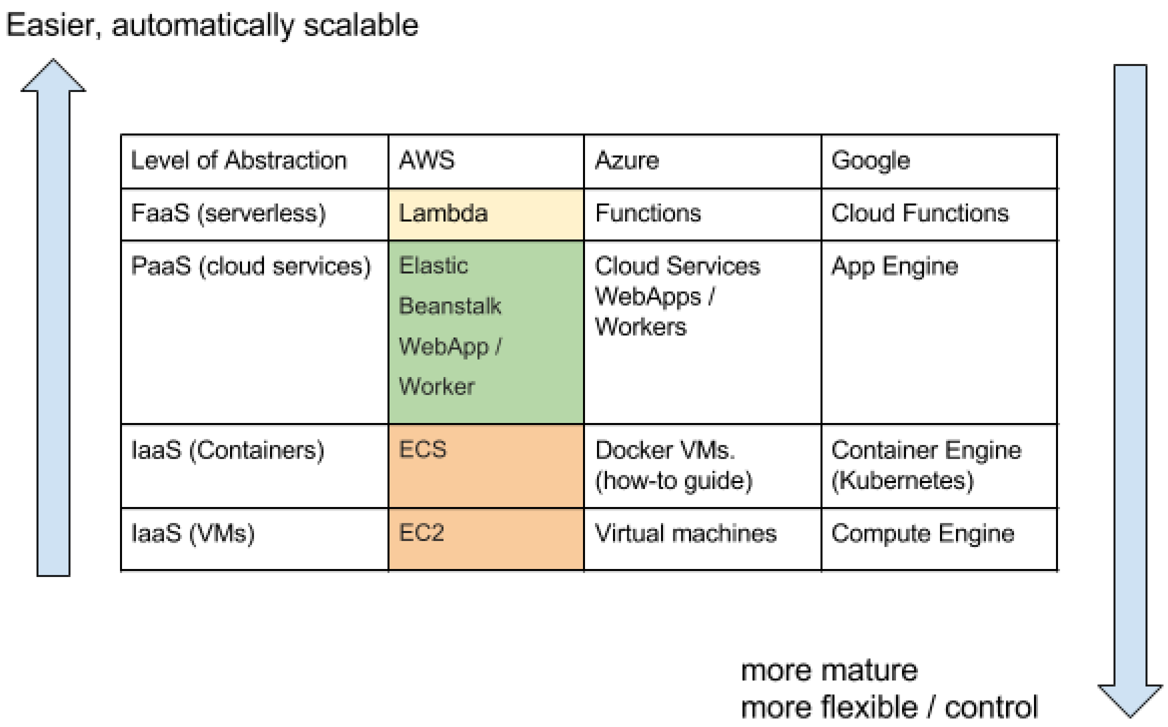
Our principle here would be the try to use a platform that delivers the most reasonable modern infrastructure at a low vendor lock in and low administrative cost.

In practice there are 4 main cloud vendors today, being amazon the distant leader computing the same market share as the other 3, Azure, Google and IBM together.

Although AWS is our primary target we do not want to get completely lock . So, we analyze here how the compute options appear in the main providers and how we plan to choose between them.

In a nutshell, we have this set of comparable options today between the 3 bigger / more popular ones. Let's review those options in the area of computation. Another section deals with the area of data storage.

There are two main are of computation. The ones oriented to the networking applications, web applications, workers, etc which is part of our solution and the area of big data analytics with more specific requirements. This section focuses on web applications / api which are part of our multi platform solution and ‘the face to our users’. We will discuss big data analytics platform in the next section.


Easier, automatically scalable
more mature
more flexible / control

Level of Abstraction	AWS	Azure	Google
FaaS (serverless)	Lambda	Functions	Cloud Functions
PaaS (cloud services)	Elastic Beanstalk WebApp / Worker	Cloud Services WebApps / Workers	App Engine
IaaS (Containers)	ECS	Docker VMs. (how-to guide)	Container Engine (Kubernetes)
IaaS (VMs)	EC2	Virtual machines	Compute Engine

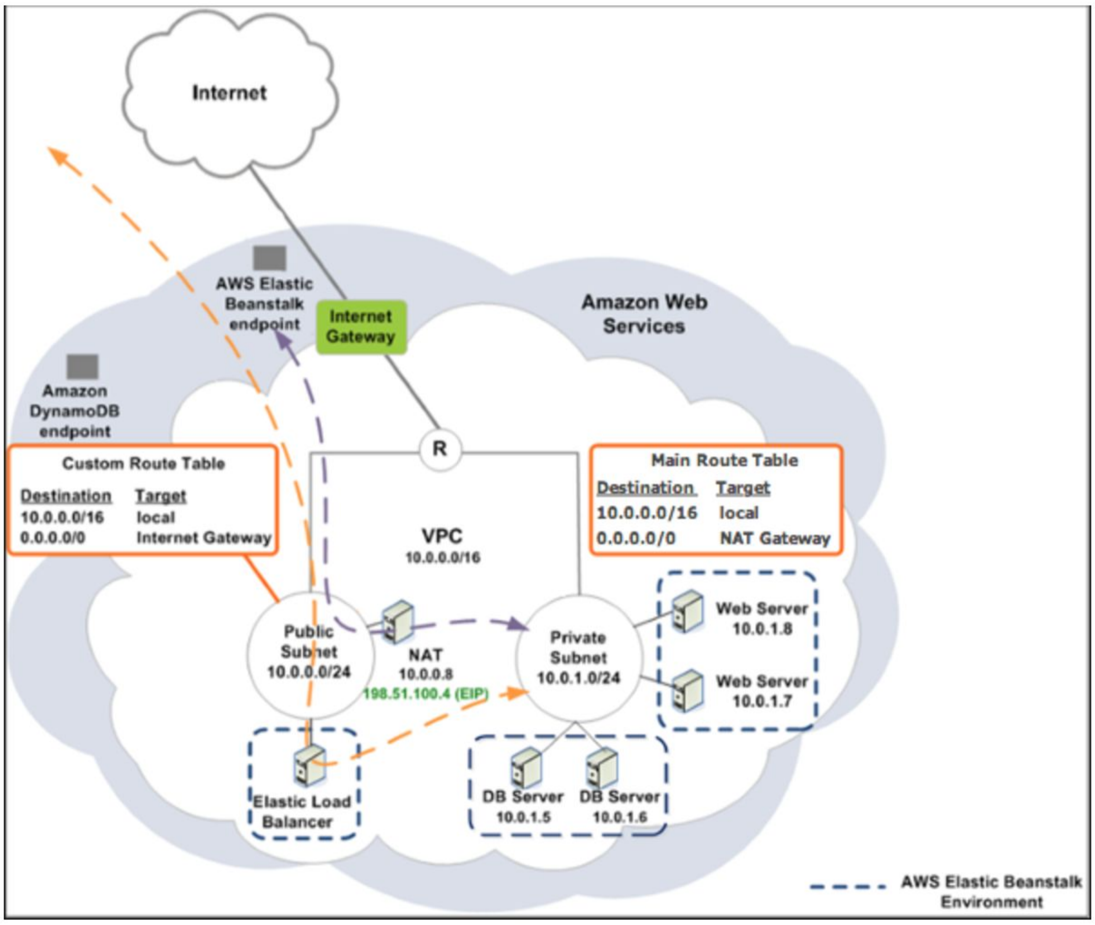
Essentially, and without adding layers of other middleware, you can deploy applications to AWS (and other providers) in 3 mains ways, order by level of flexibility.
Creating a virtual machine is the most flexible but also the most demanding from administration point of view. Using virtual machines is what is normally called IaaS infrastructure as a Service. Then, for web applications you can use web app which are, essentially, simplified containers for simple web applications, such as node,js. Using web applications is much more simple to accomplish scalability, easy deployment, load balancing, security etc. And the simplest of all, but also extremely easy to scale is the Lambda microservices approach. These last two are more what is called PaaS or Platform as a Service. We are planning to use each of these 3 mechanisms depending on the case. For deploying our node.js services it seems that the most logical approach is using web apps / workers. It seems to be the right balance for us.

How to deploy DAS modules to Cloud Provider (AWS)

As stated in our previous chapters, our intention is to run on the cloud as a multitenant application and our natural target is AWS. However, we want to be as standard and open as possible with hurting innovative solutions, performance or administration cost.
Our code is mostly 100% pure node.js and we always encapsulate dependencies with adapters that may be easy to rewrite if moving to another tool or platform.
Having say that, we will show here our strategy to deploy DAS on on AWS using Elastic Beanstalk that we think is the right balance of performance, easy administration but at the same time easy to move to another provider like Azure.

Elastic Beanstalk

Every node.js server would be deployed as a web app or worker to be AWS beanstalk layer. (Deployment of MongoDb and Apache Kafka will follow another procedure)
Fir of all, we need to differentiate internet-facing and internal only services.



Cloud Data Storage Strategy

Our solution is based on the notion of “Data Lake”. That strategy is based on fast and very scalable data plain data capture, then, later processing and then consumption over different options.
So, pour strategy may use a combination of data storage options. 3 of the most importants are: Simple files systems (S3) , Document Databases (MongoDB) and highly structured databases like redshift.
Since, in the short term, we do not need to worry about deploying or configuring S3 or redshift, we will concentrate in this section to talk about deployment options for MongoDB.

Cloud Big Data Processing Strategy

Although big data processing is also a compute platform, big data analytics has specific needs like batch processing of very big large data sets. Streaming analytics also impose a special challenge but is address in the next section.

....WORK IN PROGRESS

Big Data Subject	AWS	Azure	Google
Heavy Processing	AWS Elastic MapReduce (EMR), a managed Hadoop, Spark and Presto solution.		
Data Pipeline	AWS Data Pipeline		

Events, Pub/Sub, Alerts and Streaming Analytics

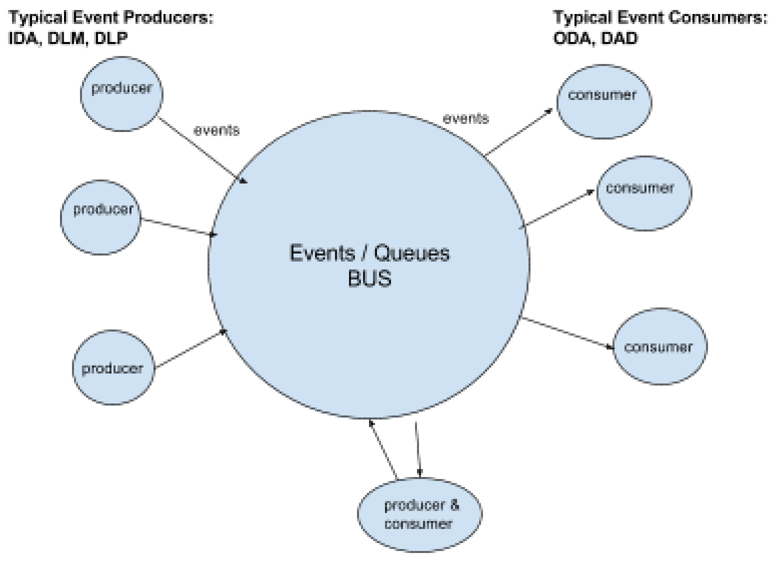
This section is about rapid processing of information with the goal of giving consumer a very fresh piece of information, alter or notification. The key specific need here is to be able to capture, distribute and process and “push” or “send” information very fast to the consuming side.

When dealing with slow moving analytical information that is being updated periodically, is acceptable and convenient that information goes from the source to the data store, then gets processed and flows to the consumer ‘by demand’. Server side waits for request and process those request using computing power or reading precomputed information.

When dealing with fast moving information that is need as soon as possible the logical way to for information to flow from the server to the client. Also, during information capture, events need to be distributed into different internal consumers like processors, databases, etc in parallel. This is need to be able to process and send conclusions as fast as possible.

For example, when receiving device information with battery status, we may want to save this information and then process it historical to calculate a moving average. But at the same time, the consumer may be interested in having the actual battery status as soon as possible without having to wait until the unwanted average is calculated. Further, our internal system may also generate events after some process discovers that some alert need to be triggered a bit after the battery status was received.

This basically creates the need for a very fast, reliable and scalable platform to producers - consumers platform



There are various ways a module A can 'push' events or notifications to another module B in a fast way, the most common ways are:

1. Module A knows module B in advance and just calls an API in module B.
2. Module B "subscribes to events from module" module A at some point and after that Module A just calls module B.
3. Modules B just 'emits' a message to a common "bus" and Module A subscribe to that "bus" and gets the message.

We are going to use all these methods depending on the situation. Method 1 is the best if both modules know each other in advance. Is clearly the most efficient. However, this creates a hard dependency that can be a problem when change is needed.

The second approach is extremely popular but also needs module B to know about module A. Almost every javascript asynchronous program uses this approach.

The third method is the most flexible. The performance impact, if well implemented, is minimal. This is the role of tools like Kafka that we will probably adopt.
Anyway, following our 'low dependence lock-in' we will always create and adapter that maybe in later time we will rewrite to use another framework or tool to implement the bus.

For the specific problem of pushing information from servers to browsers, the main idea is to use a dynamic subscribe approach based on sockets.

Data Formats - Serialization - Array Programming

Conceptually, it is always better to have a common way to represent, store and transmit information. For us this is JSON. Like XML, this is text based format with a very powerful expressing power and very easy to manipulate.

However, of course, this is not the best way to store and transmit information from a practical point of view. So, even when systems may accept and deliver information using json they may us other formats to internally manipulate and store that information.

For example, MongoDB uses BSON and is normally very transparent for a JSON user.
In the case of Matrices or array-oriented scientific data to represent scientific data sets format like NetCDF are orders of magnitude more efficient to represent, store and manipulate matrices.

Array programming will become fundamental for scientific data manipulation and machine learning. Very complex and time consuming process can be express concisely and efficiently using matrix computation. Many languages focus on array programming to achieve this such as MATLAB, Octave, the NumPy extension to Python., etc.

So in many context we may decide to have JSON to Other Format converters to convert information back and forth. This is normally called data serialization and done by serialization tools using metadata. A good example is Apache Avro

This topic is still in the backlog for future consideration.

Cloud Continuous Deployment Strategy

Version Control Integration
Ideally, our code will be stored in a single version control locations based on GIT and deployment is about running a few scripts that will push the latest version of the code to the corresponding instances.
In the case of Elastic Beanstalk we can use this EB command interface to push changes after unit and integration testing was completed.

For the moment we manage our own repository, but probably it would be better to use a hosted environment. Here are some options: GitHub, BitBucket, Aws CodeCommit.

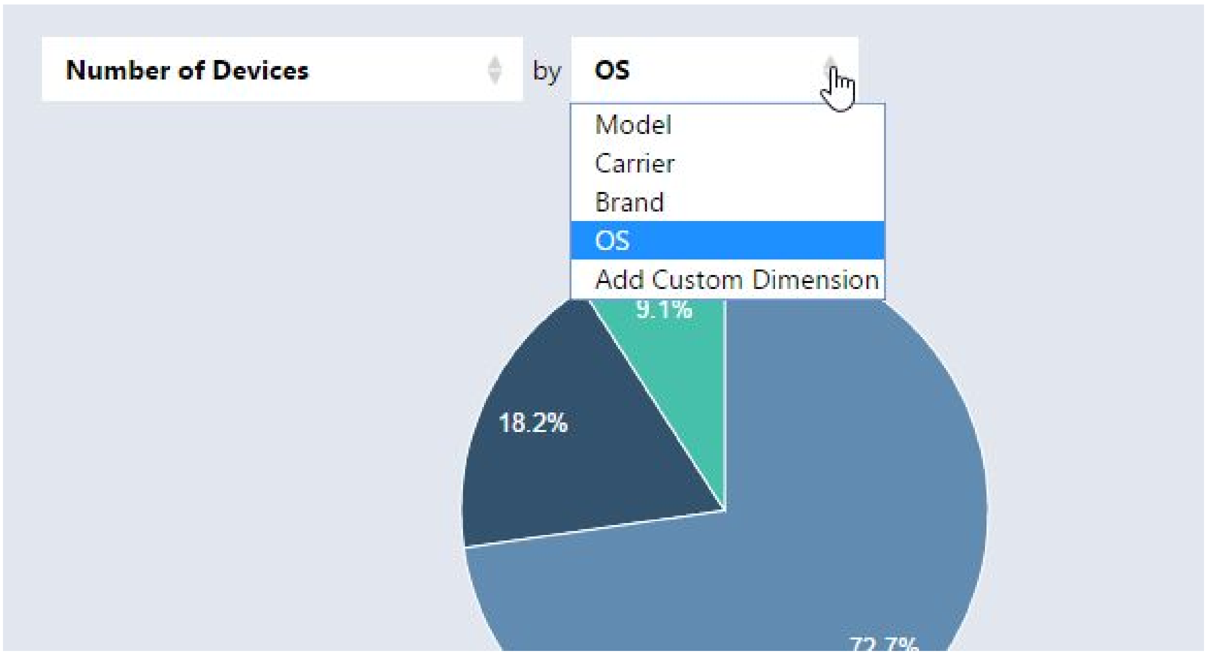
Deeper into DAS Modules Functionality

DAD: The Data Analytics Dashboard

DAD is configurable, dashboard where pages filled with charts, ‘widgets’ with key information, alerts and notifications are available to the users. This is the most visible part of our system from a user perspective. Customer can still access the information from our APIs and look at it with their own tool but this will be our “reference implementation”. The highly configurable, secure, easy to consume, pre-facilitated way to look at information.

The main idea of the dashboard is pretty simple but powerful. Every user has its own configuration that will specify how many pages they have, what components (charts, widgets, maps, tables, etc) those pages show, how these components navigate and how those components get data.

To visualize this in a concrete case let’s imagine a user is only interested in one data set adn he decides to look at a pie chart. So, maybe he configures some a chart that looks like this:



This chart, offers the user the possibility of looking at his dataset using 4 predefined dimensions. Those dimension may come from a pre-stored configuration or done by somebody by himself in the past. He can dynamically change some of the information that define this charts, such as parameters, dimensions, metrics..etc

He can also add a totally custom dimension on the fly using attributes that he knows may exist in the data set (or gets this from a dictionary).

Now, let’s look a bit more into the who we do this.

That user configuration is stored in a big json file, that is kept in the database but can be changed dynamically, that will define this chart using a json configuration.

The following (shortened for clarity) json object defines the chart show before.
We have, for now, 4 information visual elements: chart, table, widget and map.
Please see comments inside to understand the role of each configuration attribute.

```
{id: "chartbardrill", //UNIQUE ID NORMALLY USED FOR INTERNAL PURPOSES
   name: 'Drill Test', //NAME THAT APPEARS ON THE PAGE
 element: “chart”,  //THIS ELEMENT IS A CHART
   type: "bar", //THIS IS A BAR CHART
   endpoint:'ListOfDevicesNotSurvivedShift', //SERVER API THAT RETURNS DATA SET
   data:[  //THIS IS TEST DATA FOR UNIT TESTING AND HELPS TO UNDERSTAND EXAMPLE
       {DevId:'vzfsvzfsvzfsvz0', LastBatteryStatus:10,
BatteryChargeHistory:JSON.stringify([5,6,5,7,8,9]),os:'iOS', brand:'Apple', model:
"iPhone 6" , carrier: "Fido", explodes:"no"}
       ............. more ...........
   ],
   reduction:{ //AT THIS MOMENT, THE CHART REDUCES THE DATA SET USING THE DIMENSION
‘OS” AND A METRIC “COUNT OF DEVICES”
       dimension: {attribute: 'os', name:'OS'},
       metric: {attribute:'DevId', op:'count', name:'Number of Devices'}},
   reductions:[//THOSE ARE OTHER PREDEFINED REDUCTIONS YOU CAN “DRILL” TO
       {dimension: {attribute: 'model', name:'Model'},
           metric: {attribute:'DevId', op:'count', name:'Number of Devices'}},
       {dimension: {attribute: 'carrier', name:'Carrier'},
           metric: {attribute:'DevId', op:'count', name:'Number of Devices'}},
             ............. more ...........
       ],
   dimensions:[ //PREDEFINED DIMENSIONS YOU CAN USE TO CREATE OTHER REDUCTIONS
       {attribute: 'model', name:'Model'},
       {attribute: 'carrier', name:'Carrier'},
             ............. more ...........
   ],
   metrics:[//PREDEFINED METRICS YOU CAN USE TO CREATE OTHER REDUCTIONS
       {attribute:'DevId', op:'count', name:'Number of Devices'},
       {attribute:'Last Battery Status', op:'avg', name:'Average Battery Status'}
   ],
   tableId: 'table1', //ID OF THE TABLE ELEMENT THAT CAN SHOW THE RAW DATA SET
   action: 'drillFromElement',//THE ACTION WHEN USER CLICKS ON A CHART’S BAR
   parameters: [//PARAMETERS TO SPECIFY THE DESIRED DATA SET FROM ENDPOINT
       {
           shiftDuration:8,
           shiftStartDateTime:"2016-08-25"
             ............. more ...........
       }],
   uiparameters: [//PARAMETERS USER CAN CHANGE ON THE FLY FROM THE UI
       {
           Type: DadParameterType.DateTime,
           Name: "Shift Start Date & Time",
           DataSource: "shiftStartDateTime"
       }
             ............. more ...........
   ]
}
```

Summary: the dashboard is very customizable tools that can be used to create and configure elements to show data in various way. We have only shown the surface, other things are coming like notifications, alerts, etc. This configuration can be different for every user and will be stored in the database. What a user can do will also be restricted to a dictionary according to the access control level.

ODA: The Output Data Adapter

The purpose of ODA is simple: to provide an internet exposed swagger-compliant cloud-provider-independent API for DAS to publish its public data endpoints to be consumed by the dashboard and by client applications.

It also handles the verification of the access token and will be responsible for first layer of configuration based access control.

Also, it may be the place to do any remaining transformation, filtering or aggregations.

Yes, in some sense there is a bit of overhead as data or data our is going through an extra ‘hop’. Welcome to modularization. The impact is minimal especially in a scalable cloud environment and the benefits are various from security to loosening the coupling of consumers and producer of information that will evolve independently at different speeds.

As we said before ODA, DAD and it twin IDA will be exposed to internet.

Most likely the load balancer will also add overhead and an extra hop, but it pays off at the end.

IDA: The Input Data Adapter

The purpose of IDA, the twin brother of ODA, is simple as well: to provide a cloud-provider-independent API for DAS to accept data from customer or external data sources.

It also handles the verification of the access token and or certificate presented by the different data sources and will be responsible for shipping this information to the cloud infrastructure.

Also, it may be the place to do any remaining transformation, filtering or aggregations that may be needed when capturing data.

Same comment about minimal redundancy as before.

DSS: The Security Service

This module is responsible for:

- Let account administrators to define ‘tenants’ for its account.

    - this is pretty similar to what you do when opening an account in a cloud provider and creating a cluster o similar container concept.
- Define how user will be authenticated.
    -
- Authenticate user, on behalf of the provided IDP, returning a JWT token.
- Let account administrators to define ‘data sources’ for this account.
- Authenticate the data sources when they try to be able to put data.

For security reasons, as this module is exposed to the internet, it has been separated into a front-end and back-end independent. The front end is responsible for UI and redirections and backend is responsible for communicating with DB and signing tokens.

put this back?:
About DSS? this For now he can

add a Mobicontrol instance to act as data provider using the MCDP C# data collection agent

and a basic generic API data provider using DLM

both MCDP and DLM are basically data readers that pull information from the data source and send the information to the Input Data Adapter IDA.

IDA is just a secure data adapter that accepts data from data sources and sends the data to the data lake.

DSS is also an authentication authority in our system. When data sources need to send data to IDA, they need to get an authentication token from DSS.

Hands-on: Get, Understand and Run the Code

This section is targeted to developers and we expect you to use a command line.

On Coding Standards and Quality Assurance

(DRAFT)

Functional Testing
Performance Testing
Security Testing

Unit Level Testing
Component Level Testing
Partial Integration Testing
Full integration Testing

Annex - Technical Details - References

THIS SECTION WILL BE REWRITTEN. PLEASE READ THE README IN GIT FOR NOW.

Installing DAS on Premise / IaaS / Containers Services

The purpose of this section is to show how to install the node.js part of the code in a single or multiple servers without a managed cloud platform. Although this is not our main intended escenario, we want to show here that this is possible to be done with security in mind. In this case, we will have to manage certificates and reinforce the server to server communication security.

We will show how to do this on a single server but clearly the instructions can be follow in various server in the same fashion.

The ultimate goal, however, is to deploy this application to a cloud environment assigning different servers or containers to each module.. We will address that in the next section.

The code is based on Node.js. So, is multiplatform and you can use any operating system, a cloud server or an in-site machine.

In general terms, installing node.js part of DAS is a simple as

get certificates
git clone
npm install
npm start

In other sections we will discuss non core components that run specifically on windows (MCDP) or code that only runs on AWS. Also, we will later discuss aspects related automation and virtualized cloud environment in a cloud environment.

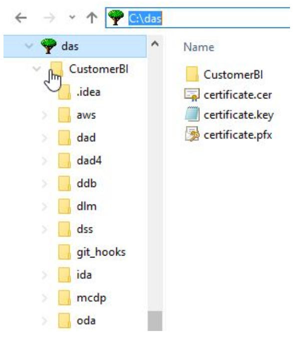
Preparing environment (SSL, Git, Node, MongoDB)

SOTI DAS is a secure web/cloud application and all its components only run on SSL.

*Although we can run them with self signed certificate we recommend using the SOTI.NET signed certificate that you can get from your machine to create a certificate with exportable key to certify your machine. In another environment you should get a proper server certificate. For the SOTI final product the intent is to use a SOTI.NET signed certificate for domain "***.das.soti.net**". DAS modules that we are going to install need GIT, node.js, npm and some of them mongodb.*

So, please, follow the next steps to prepare your environment.
Although all paths and filenames to shared resources like certificates, keys and connection strings are configurable, our system accepts the root folder of the installation as the default location. For simplicity sake our instructions will assume you use the default location.

1. *Create a folder to contain the project, for example C:\DAS*
2. *Put there both the certificate and the public key in two separated files*
   a. *call them "certificate.cer" and "certificate.key".*
   b. *If you do not know how to do this, see the Annex section for hints.*
3. *Install GIT and Node.js (> v7.5.0) and MongoDb (> v3.4.3)*
   a. *Make sure npm is new (> v4.1.2)*
   b. *cd to your folder, for example C:\DAS*
   c. *git clone http://taipan:8080/tfs/SOTITFS/_git/CustomerBI*
4. *After all this process, you should have a folder structure similar to this:*



About the code and the default configurations
All the core modules you are going to install in your machine are node.js applications that can be found in the corresponding folder in GIT: "dad", "oda", "ida", "dss", "ddb", "dlm" and other modules that my come in the future such as "dos". In every folder you may see various files but the most important ones from the configuration point of view are:

package.json
appconfig.json
readme.md

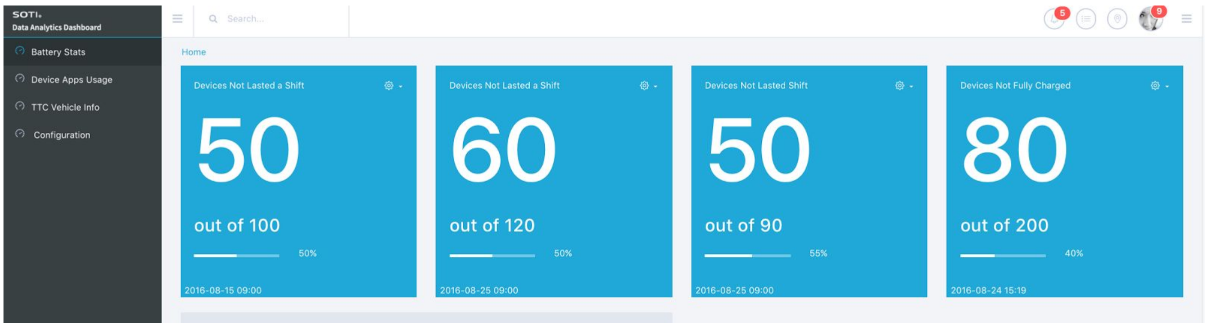
*All modules come with a default configuration that you can use out of the box so you do not need to change too much for a single server installation. Only thing is that appconfig.json will normally contain a variable **'testingmode" that comes in true by default.** You will have to set this variable to false to run in production mode.*

Installing DAD (Data Analytics Dashboard)

To see DAD in action go to C:\DAS\CustomerBI and do

1. *cd dad*
2. *npm install*
3. *npm start*
5. *hit the url https://youserver.corp.soti.net:4200*
   a. *or just https://localhost:4200 if you are inside the machine*

and if nothing went wrong, you should see a page sort of like this:



Good. If you see that the browser tells you that page is insecure that may mean that certificates could not be found or are not trusted.

For now, you are looking at DAD in test mode. Data that you see is coming from a predefined test data. To run for real you need to update the configuration in appconfig.json that comes like this by default

```
{
 "testingmode":true,
 "dss":"https://10.0.91.25:3013",
 "oda":"https://dev2012r2-sk.sotidev.com:3002"
}
```

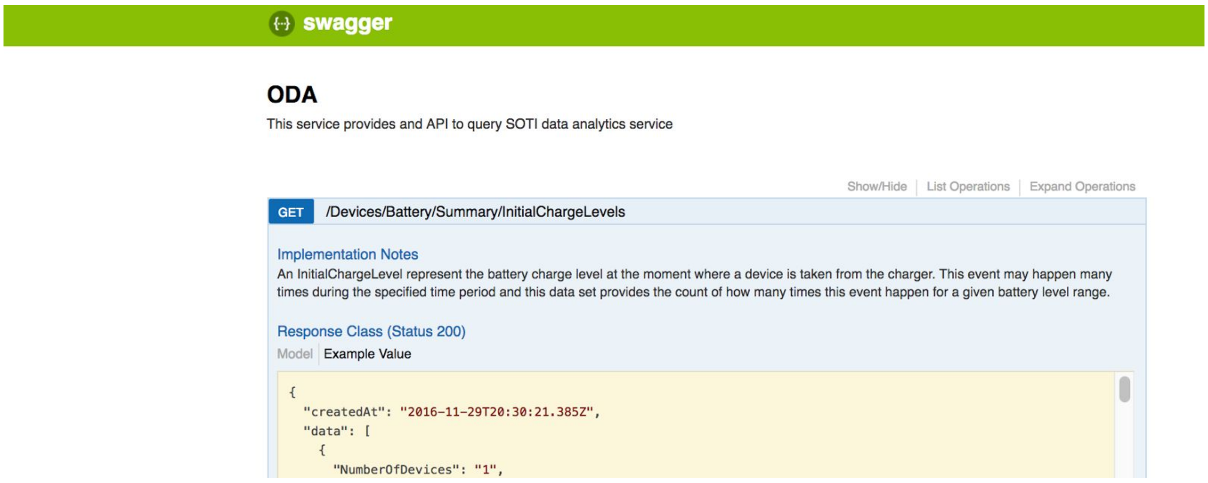
You will need to change 'testingmode' to false.

Installing ODA (Data Output Adapter)

To see DAD in action go to C:\DAS\CustomerBI and do

```
cd oda
npm install
npm start
```
with the browser hit https://localhost:3002/docs

if all went well you should see something like this:



ODA

This service provides and API to query SOTI data analytics service

/Devices/Battery/Summary/InitialChargeLevels

Implementation Notes

An InitialChargeLevel represent the battery charge level at the moment where a device is taken from the charger. This event may happen many times during the specified time period and this data set provides the count of how many times this event happen for a given battery level range.

Response Class (Status 200)

```
{
  "createdAt": "2016-11-29T20:30:21.385Z",
  "data": [
    {
      "NumberOfDevices": "1",
```

if you did not get the page, check the console for possible errors or warnings, make sure you have the certificates explained before.

For now, you are looking at ODA in 'dev' mode running in port 3002 and communicating to DDB on localhost:8000. You can update this in appconfig.

```
{
"deployment_type":"dev",
"port": 3002,
"db-address" : "http://localhost:8000",
```

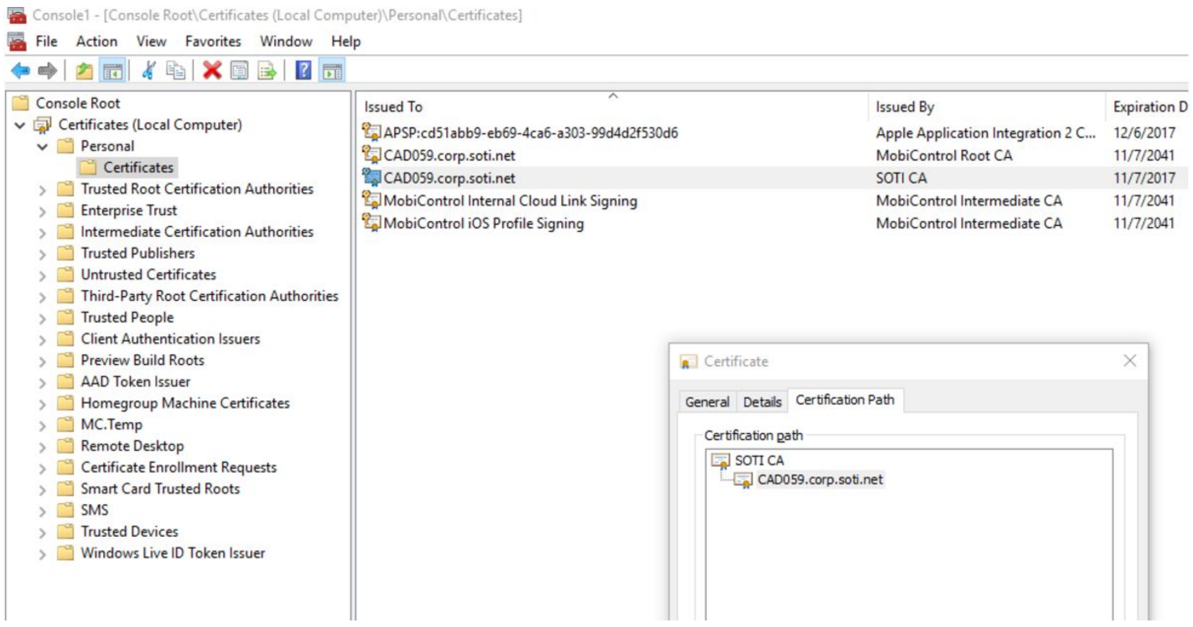
}

Installing DSS (Data Security Service)

if you did not get the page, check the console for possible errors or warnings, make sure you have the certificates explained before.

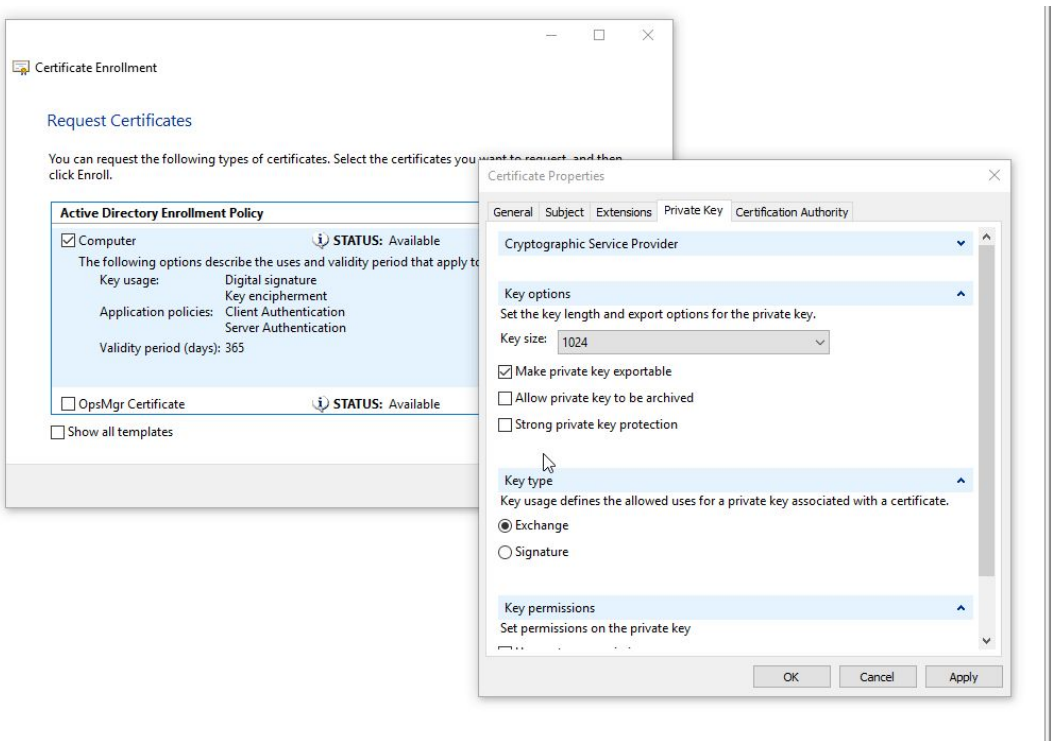
How to use my SOTI machine certificate

Skip this section if you know how to do this.
Using MMC (Microsoft Management Console), find SOTI certificate and create a new certificate for your machine making your private key exportable


Console1 - [Console Root\Certificates (Local Computer)\Personal\Certificates]
File Action View Favorites Window Help
Console Root
Certificates (Local Computer)
Personal
Certificates
Trusted Root Certification Authorities
Enterprise Trust
Intermediate Certification Authorities
Trusted Publishers
Untrusted Certificates
Third-Party Root Certification Authorities
Trusted People
Client Authentication Issuers
Preview Build Roots
AAD Token Issuer
Homegroup Machine Certificates
MC.Temp
Remote Desktop
Certificate Enrollment Requests
Smart Card Trusted Roots
SMS
Trusted Devices
Windows Live ID Token Issuer
Certificate
General Details Certification Path
Certification path

Issued To	Issued By	Expiration D
APSP:cd51abb9-eb69-4ca6-a303-99d4d2f530d6	Apple Application Integration 2 C...	12/6/2017
CAD059.corp.soti.net	MobiControl Root CA	11/7/2041
CAD059.corp.soti.net	SOTI CA	11/7/2017
MobiControl Internal Cloud Link Signing	MobiControl Intermediate CA	11/7/2041
MobiControl iOS Profile Signing	MobiControl Intermediate CA	11/7/2041

SOTI CA
CAD059.corp.soti.net



You can request the following types of certificates. Select the certificates you want to request, and then click Enroll.

Active Directory Enrollment Policy

☑ Computer STATUS: Available
The following options describe the uses and validity period that apply t
Key usage: Digital signature
Key encipherment
Application policies: Client Authentication
Server Authentication
Validity period (days): 365

☐ OpsMgr Certificate STATUS: Available

☐ Show all templates

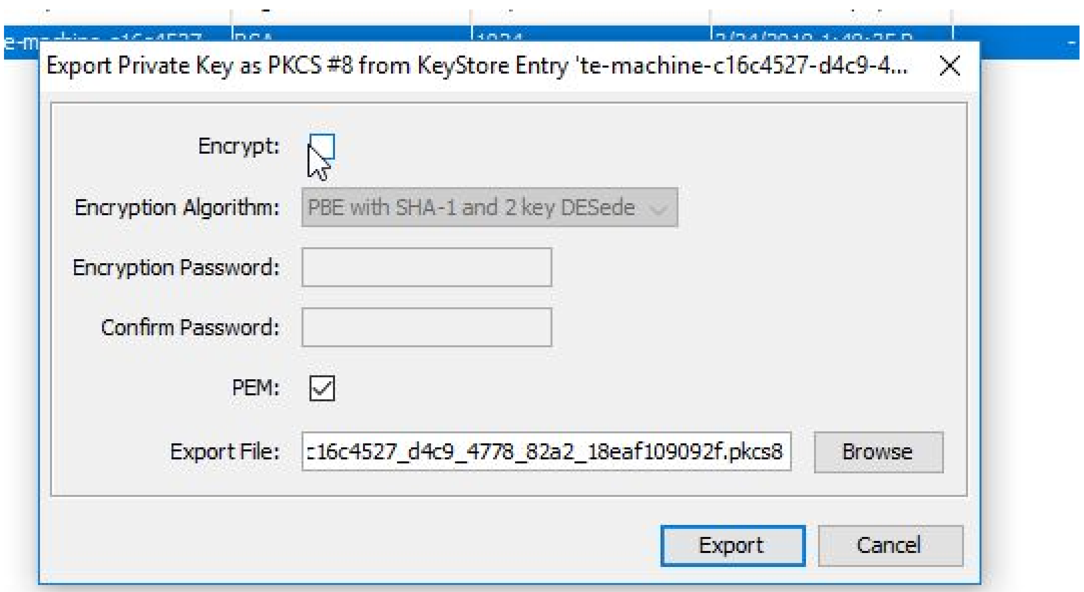
Cryptographic Service Provider

Key options
Set the key length and export options for the private key.
Key size: 1024
☑ Make private key exportable
☐ Allow private key to be archived
☐ Strong private key protection

Key type
Key usage defines the allowed uses for a private key associated with a certificate.
● Exchange
○ Signature

Key permissions
Set permissions on the private key

Then, export the private key using PKCS without encryption


Export Private Key as PKCS #8 from KeyStore Entry 'te-machine-c16c4527-d4c9-4...

Encrypt: ☐

Encryption Algorithm: PBE with SHA-1 and 2 key DESede

Encryption Password:

Confirm Password:

PEM: ☑

Export File: :16c4527_d4c9_4778_82a2_18eaf109092f.pkcs8

How to set up DDB on a remote instance.

Pre-requisites:

1. *IT/Cloud team created an EC2 with Ubuntu on it.*
2. *Certificates are ready for each components*
3. *Certificates are placed in root folder of user.*

Steps:

1. *Using an SSH client, connect to the machine with username and password*
2. *Run the following commands to install node (v6) and required files*
    1. *curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -*
    2. *sudo apt-get install -y nodejs*

c. sudo apt-get install -y build-essential

1. *Once node is installed change to /etc folder and set up hostname*
    1. *Cd /etc/*
    2. *(change hosts file to include hostname) Sudo vim etc/hosts*
        1. *Replace 127.0.0.1 localhost è [service].das.soti.net (in my example I use ida.das.soti.net)*
    3. *(change hostname) Sudo vim etc/hostname*
        1. *Replace ip with [service].das.soti.net*
    4. *Sudo reboot to restart*
2. *After computer is rebooted go to home folder and create a folder for the component*
3. *Place the source code of the component*
4. *NPM install*
5. *Update Appconfig /config.json of the component to provide location of certificates / access keys / secretes as required*
6. *If Mongodb is required for the component, install mongodb and start a mongo server. Otherwise skip this step. (Only dlm and ddb require this)*
7. *NPM start*

Every Api call to it must present this header

x-access-token with the following password.
'x-access-token' :
A3J9SyhuZ9TTyNdzCq5LPmNlHb32KasnRothKwnGyCegLvp9bCEMkhYbl51xPzUFK1jHUYh9EeQzqSu0464ZFCpaZ6zmiyXo3p98EbHSzPuFX1zTmX5c26vfpnl1G5khykf8dnloubXcul3T93M4jjLj1UJgnU0OwmjEH5ZA7GmHC0kKmO8gK6KCi9eDDDW6OaTOkoOm