Contents

[Functional requirements 2](#_Toc436218237)

[Possible approaches 2](#_Toc436218238)

[Rationale of the selected approach 4](#_Toc436218239)

[Technical design 5](#_Toc436218240)

[Missing requirements and assumptions 13](#_Toc436218241)

[Tools and frameworks 13](#_Toc436218242)

## Functional requirements

The client has 5 manufacturing facilities located in different sites. Each facility has multiple access points. Needs a system for management and monitoring these access points.

Create a system for management and monitoring:

1. A manager should be able to allow or deny access points to users or user groups
2. A manager should be able to create a duty roster of its employees.
3. A manager should be able to delegate management and monitoring rights based upon organizational structure from LDAP
4. A manager should be able to view log of accesses (bot successful and failures)
5. The access point should allow access to a user when he is allowed access and when he is on duty.
6. If an access violation occurs then the manager responsible for the department should be notified via email and sms.

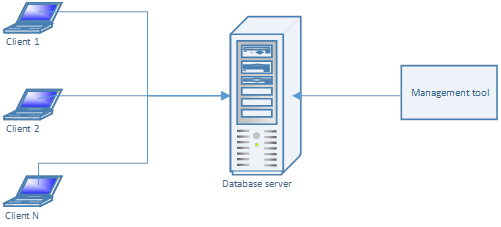
The organizational structure is stored in the company LDAP directory: sites, departments, managers, users and user groups. User name, email, login are specified for each user.

The system should be accessible from both LAN and the internet.

## Possible approaches

There are multiple approaches for this solution, each of them has its advantages and disadvantages.

#### Database-centric model

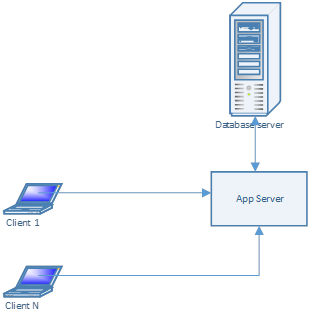


Advantages: fast development.

Disadvantages: hard to maintain and extend. Limited functionality

Such solution is possible only for prototyping. It is not recommended on production.

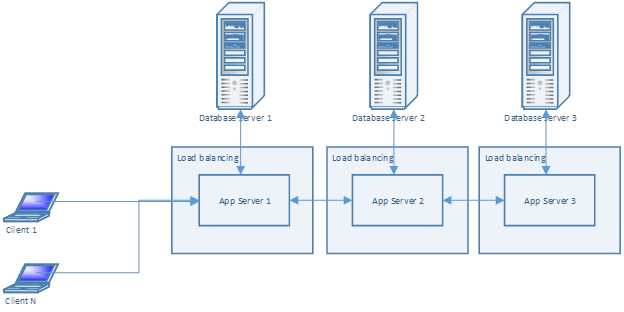
#### Client-server model



Advantages: App server contains a business logic necessary to clients. Database cannot be accessed directly.

Disadvantages: App server knows everything. Different systems cannot reuse app server, because it contains application specific logic that cannot fit different requirements in many cases. This approach also has poor scalability.

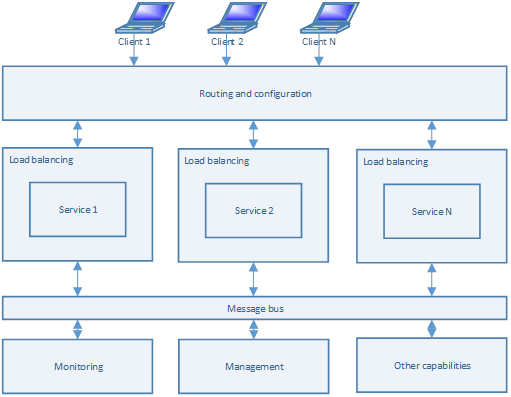
#### Multitier model



Advantages: This model is a bit better than client-server model. It allows putting different business logic in different app servers and distinguish it by requirements. It can be easier maintained and extended and client-server, because app servers represent an atomic applications. Good scalability.

Disadvantages: Because of app servers need to communicate to each other such architecture becomes a forest with configurations that is hard to maintain. Moreover, such configuration requires using server proxies between app servers that leads tight coupling between server interfaces. So it may lead problems with extensibility.

#### Micro-services model



Advantages: This model allows creating granular micro service, which do only small jobs. Each of those micro services can be placed in different server and load balanced so this model has perfect scalability and performance. Maintainability is also perfect because services do not know anything about each other. Each service can be developed and configured independently.

Disadvantages: learning curve.

## Rationale of the selected approach

According to specified requirements, micro-services architecture is a best choice. It allows creating small services, which can be load balanced independently and hosted on cloud. Each of such services can be reused by a different application. In our case, LDAP service is a good example of such shared service.

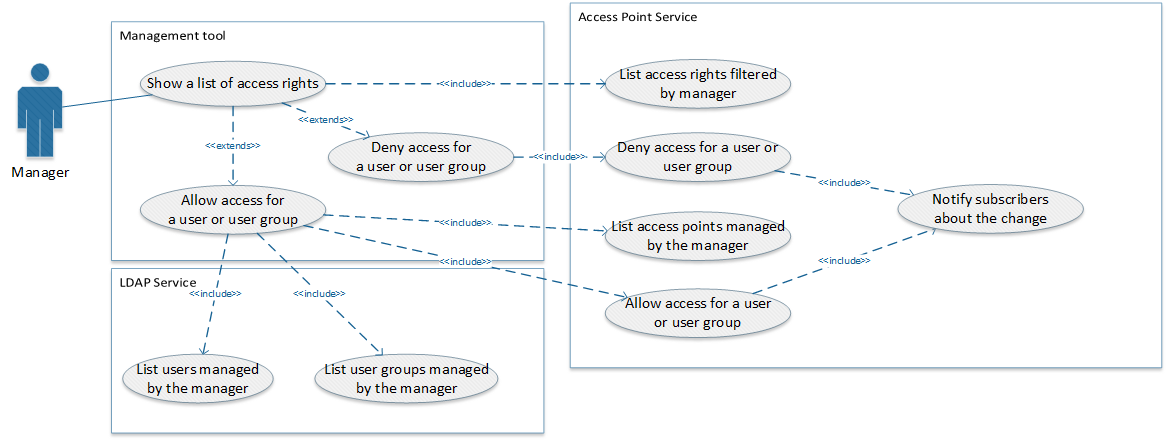
Message bus guarantees messages delivery, so we can be sure that any changes to data is processed properly.

There are few possible options to build the solution: MSMQ, Azure Service Bus, RabbitMQ. Because of RabbitMQ can be hosted in different environment, including Linux, Windows and clouds, it is the best choice for the system.

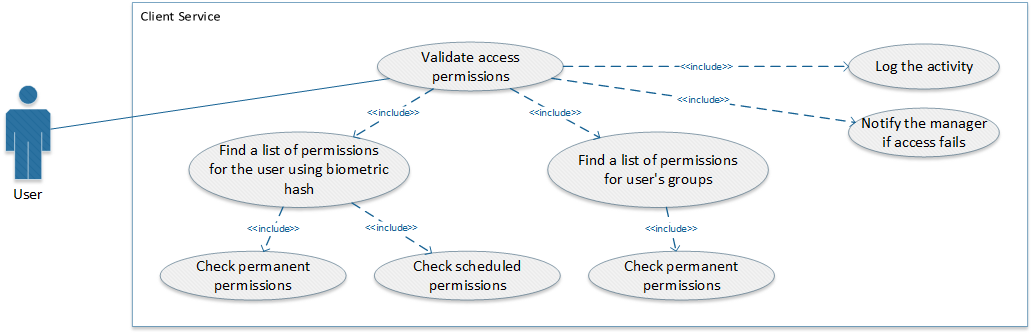
## Technical design

There are a list of use-case diagrams describing the common scenarios.

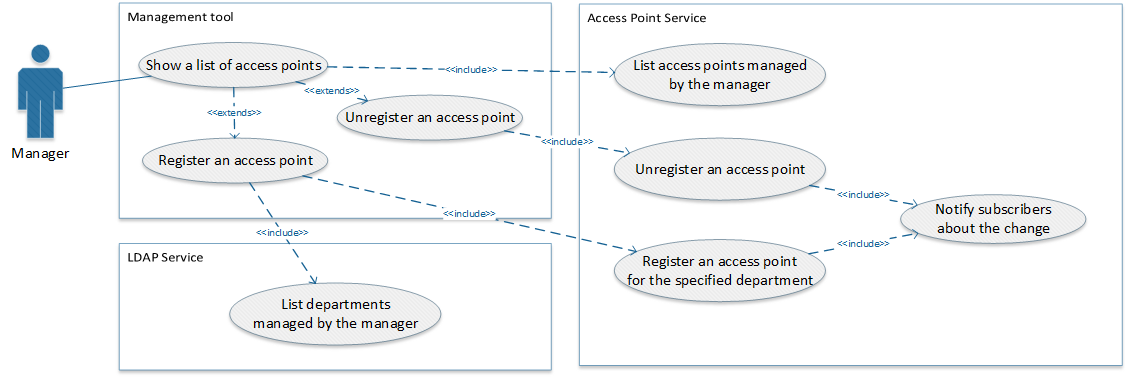
#### Use-case: Manage access rights



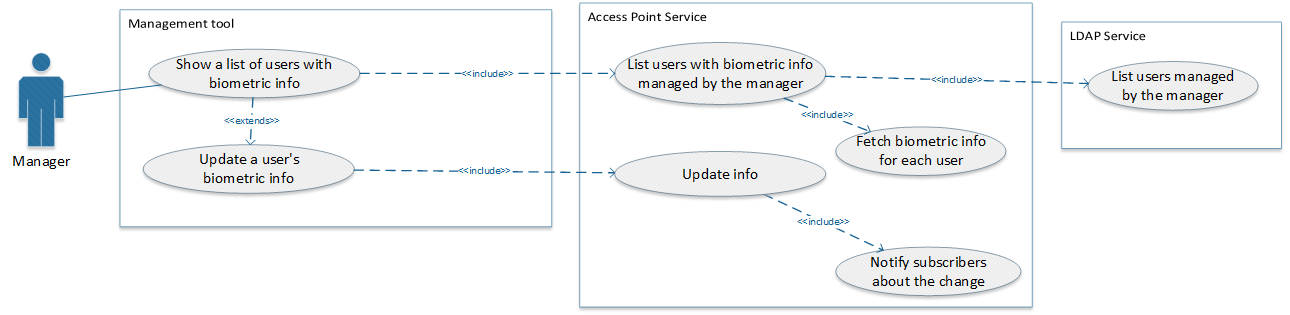
#### Use-case: Access an access point



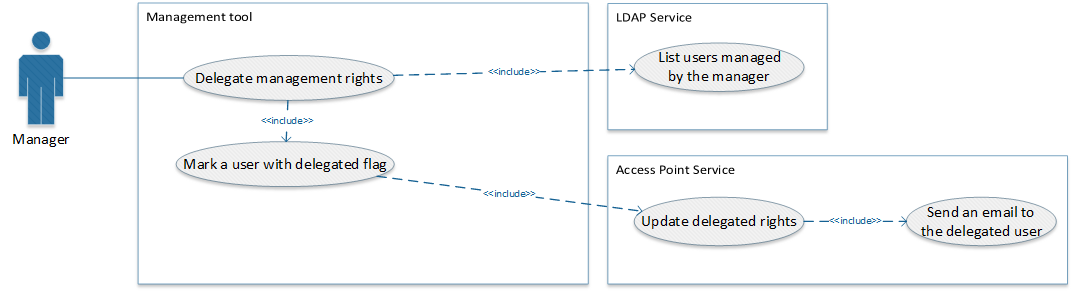
#### Use-case: Manage a list of access points



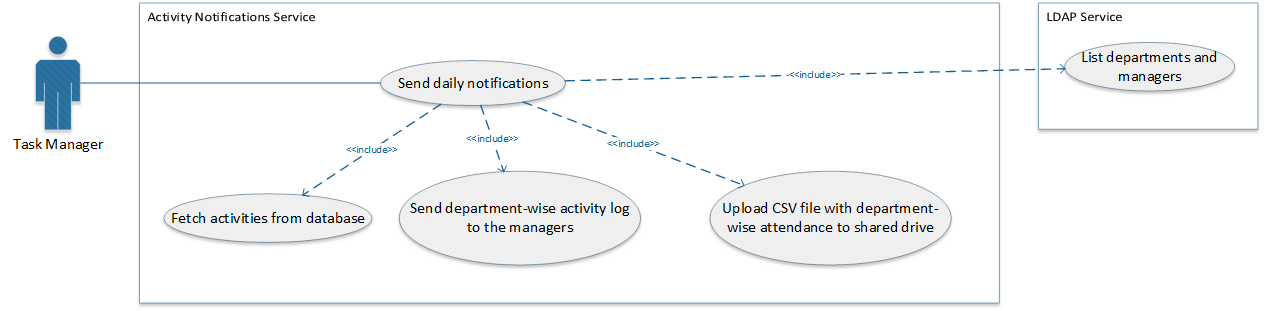
#### Use-case: Manage user biometric information



#### Use-case: Delegate management rights

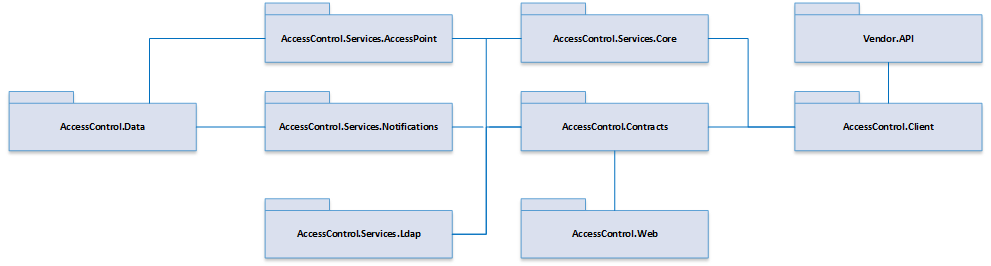


#### Use-case: Daily notifications

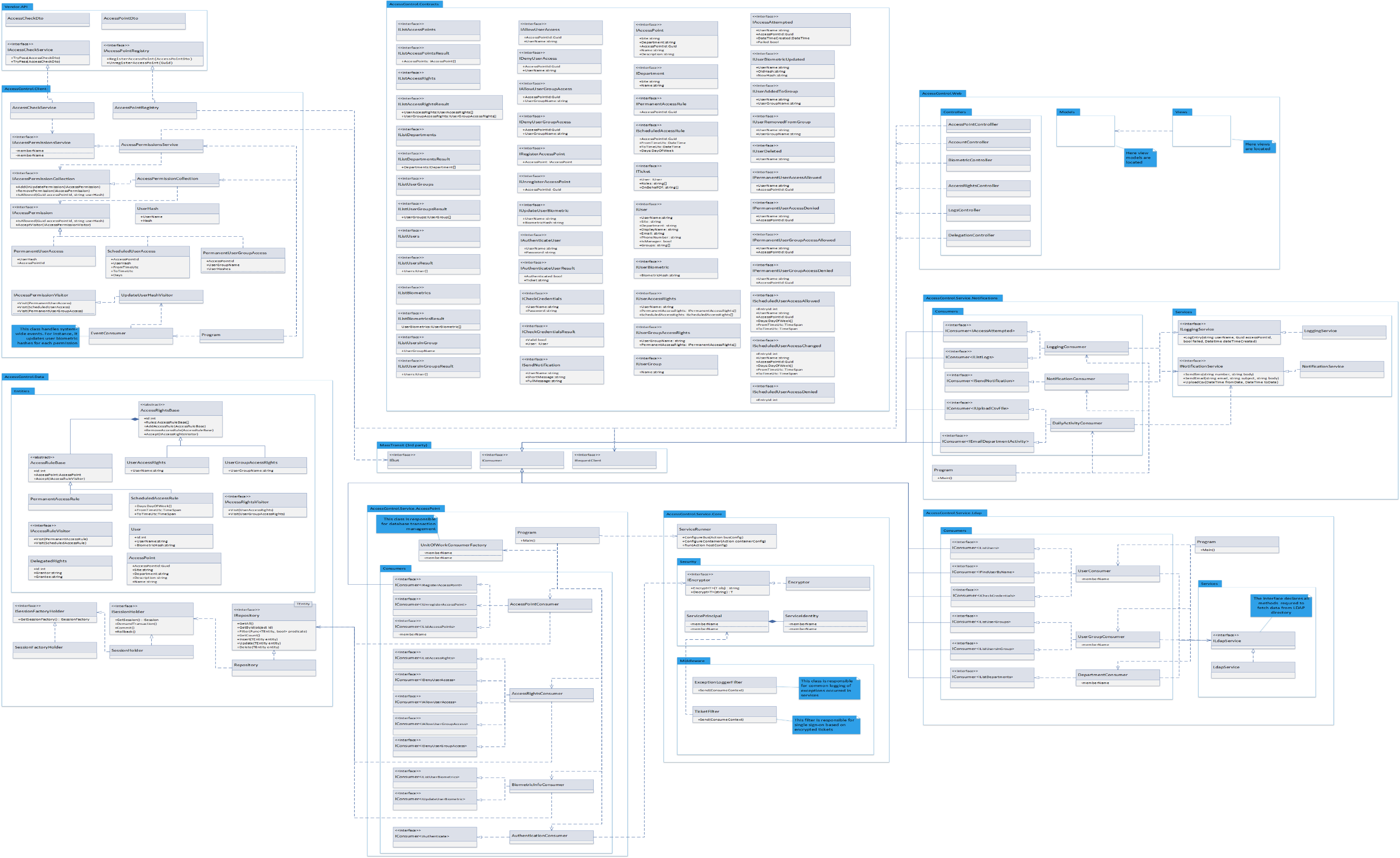


According to requirements, the following packages are required.

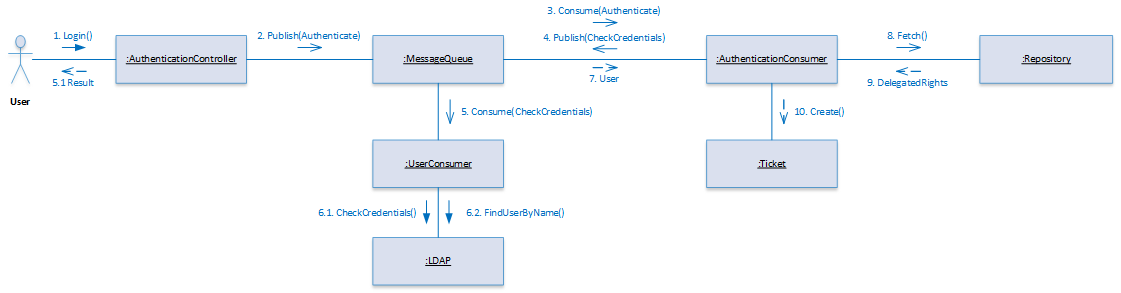
#### Package diagram



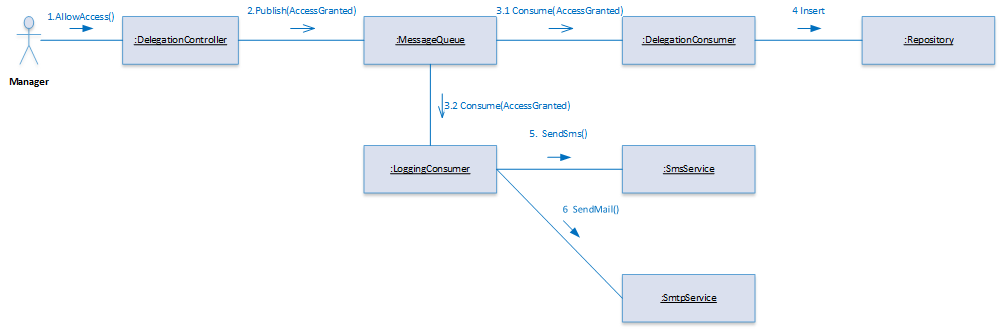
#### Class diagram



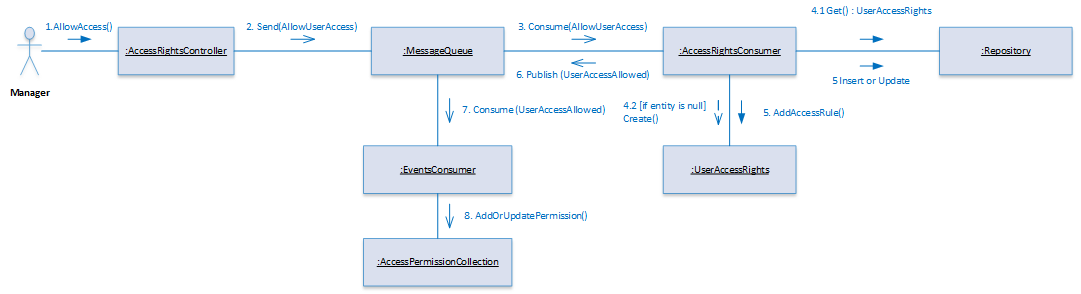
#### Communication: Authentication



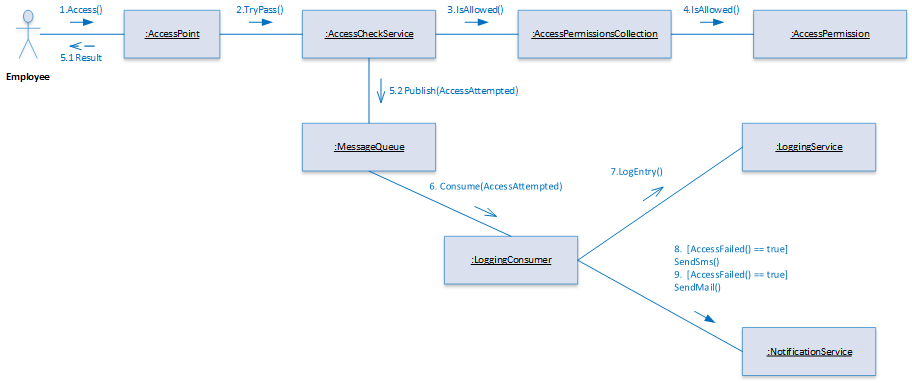
#### Communication: Management permissions delegation



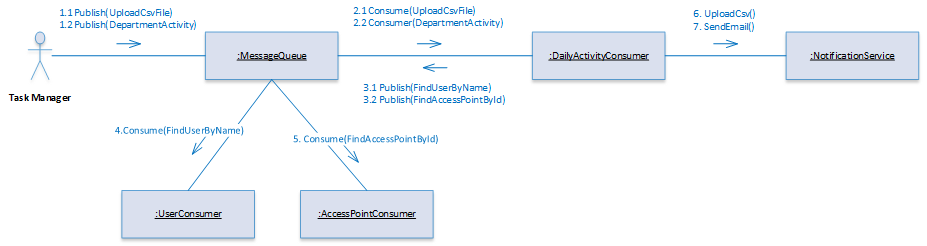
#### Communication: Allow access point



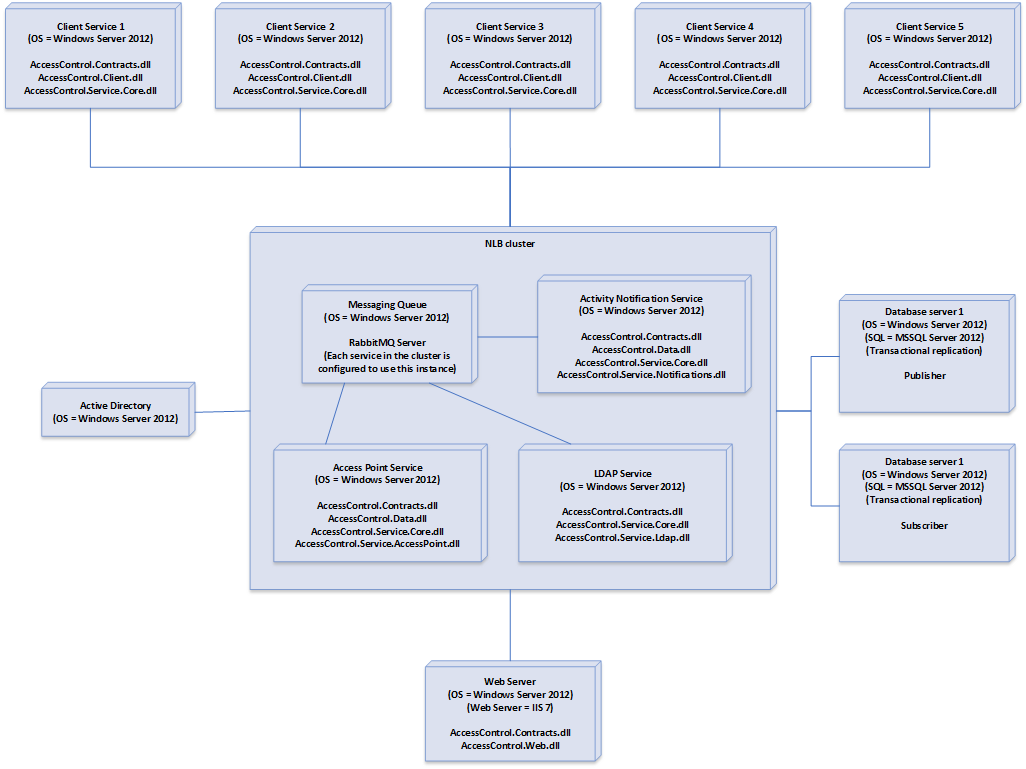
#### Communication: Access an access point



#### Communication: Upload daily logs and send emails



#### Deployment



## Missing requirements and assumptions

1. User names remains read-only, because LDAP directory does allow changing it.
2. There is not specification of 3rd party API used by vendor software. So it is assumed that vendor API defines WCF contracts (see Vendor.API)
3. There are no specifications for management system of access points. It is assumed that manager is responsible for management of access points located in the department.
4. Because of Internet can be switched off unexpectedly, we should guarantee that employees allow access points. But it is assumed that the system does not deliver notifications about access violation. If it is required, the client service should store such notifications in internal database. At this point this implementation is out of the scope.
5. There are no security requirements. We assumed that the system must support Single Sign-On. To fit this requirement the system creates an encrypted ticket and passes it to remote services.
6. To protect data from unauthorized access all communications should be made through SSL.
7. There are not requirements that describe how system should work in case of electricity problems. So it is assumed that 3rd party vendor hardware is responsible for it.

## Tools and frameworks

1. Frameworks: .Net 4.5, ASP.MVC 5
2. RabbitMQ and MassTransit to establish communications between components.
3. ORM: Fluent NHibernate or Entity Framework
4. IoC tool: Microsoft Unity, Castle Windsor or Autofac
5. Quartz.Net for task scheduling
6. Testing Frameworks: nUnit for unit tests. SpecFlow for functional tests