APPROACH FOR CHOOSING OF ARCHITECTURE MODELS FOR DISTRIBUTED APPLICATIONS DESIGNING

Elena Chebanyuk, Mykhailo Kostiuk

Abstract: *Distributed system - a number of independent computers linked by a network [Oxford, 2016]. Distributed applications allow different devices to collaborate and simultaneously proceed data. Centrally, designing of such application require some skills and additional knowledge about all levels of architecture patterns.*

In this article terminology used for distributed application designing is systematized. Then approach for matching different types of architectures according to customer requirements is proposed. Schemas and stacks of technologies for different architectural solutions are represented. Also advantages and disadvantages for different architectural solution are represented. Analyzing them software architect can refine architectural solution and compose recommendations for further development.

Comparative cost analysis for implementing different architectural styles is represented. This analysis allows to estimate stakeholders efforts and cost of distribution software on client side.

In conclusion recommendations for implementing proposed approach are represented.

Keywords: *software architecture, distributed application, Web-Service, component, event processor, entity framework, Managed Extensibility Framework (MEF), PushSharp API Development, Amazon, Amazon DynamoDB, ElastiCache Redis, Amazon Simple Queue Service(SQS), Amazon Web Services, cloud computing.*

ITHEA Keywords*: D2 Software Engineering, D 2.0 Tools, D.2.10 – Design Topic, D.2.11 - Software Architectures.*

Introduction

Increasing role of information exchange today is a precondition of appearing many applications designed of client-server and n-tier architectural styles Application that are based on client-server architectural style are widely meet today. Growing amount of different types of devices are precondition for appearing application that need to execute part of task in one device and continue execution by means of another devices.

Cloud computing, necessity to create software, simultaneously proceeding data from different resources, is a precondition for creating distributed applications. From the other side growing amount of mobile devices and tablet PCs causes to designing of applications, aimed to information exchange between different data storages.

For example you create request on mobile phone application and perform it on server. Such companies as IBM and Microsoft [Microsoft, 2003] propose stack of technologies for designing and development of distributed applications. But peculiarities of customer tasks need to precise architectural solutions.

Main definitions and standards for distributed technologies designing

There are several OMG standards that define procedures and algorithms, performed in distributed applications. They are Distributed Simulation Systems Specification, Data Distribution Service (DDS). But definition of several terms is not presented in OMG standards and is systematized from practical usage of researches and stakeholders.

Table 1. Controlled vocabulary of problem domain "Definitions and standards for distributed technologies designing"

Term	Definition
Distributed system	A number of independent computers linked by a network [Oxford, 2016]. Collection of independent computers that appears to its users as a single coherent system (or) as a single system [Kangasharju, 2008].
Distributed applications	A **distributed application** utilizes the resources of multiple machines or at least multiple process spaces, by separating the application functionality into more manageable groups of tasks that can be deployed in a wide variety of configurations [Microsoft, 2012].
Components	1. an entity with discrete structure, such as an assembly or software module, within a system considered at a particular level of analysis. ISO/IEC 15026:1998, Information technology — System and software integrity levels.

Term	Definition
	3.1. 2. one of the parts that make up a system. IEEE Std 829-2008 IEEE Standard for Software and System Test Documentation. 3.1.6. 3. set of functional services in the software, which, when implemented, represents a well-defined set of functions and is distinguishable by a unique name. ISO/IEC 29881:2008, Information technology — Software and systems engineering — FiSMA Note 1 to entry: A component may be hardware or software and may be subdivided into other components. The terms "module," "component," and "unit" are often used interchangeably or defined to be subelements of one another in different ways depending upon the context. The relationship of these terms is not yet standardized. A component may or may not be independently managed from the end-user or administrator's point of view. [ISO/IEC/IEEE 24765:2010, 2010] A specific, named collection of features that can be described by component definition. [Unified Component Model for Distributed, RealTime and Embedded Systems, 2013]
Component standard	A standard that describes the characteristics of data or program components subdivided into other components. [Unified Component Model for Distributed, Real-Time And Embedded System, 2013].
Data Distribution Service (DDS) standard	Standard describes internal structures of classes and modules, that provide interaction between components. Also there are recommendations of organizing and performing such processes as exchange information between components and collaboration of objects by means of subscribing and publishing. Data, prepared for exchanging, are published. Other services that need such data subscribed to procedures of common data updating. After subscribing component or service listens port of other connection to obtain data for further processing. [Data Distribution Service (DDS), 2015]

Term	Definition
Service	Service is an application function packaged as a reusable component for use in a business process. It either provides information or facilitates a change to business data from one valid and consistent state to another. The process used to implement a particular service does not matter, as long as it responds to your commands and offers the quality of service you require. Through defined communication protocols, services can be invoked that stress interoperability and location transparency. A service has the appearance of a software component, in that it looks like a self-contained function from the service requester's perspective. However, the service implementation may actually involve many steps executed on different computers within one enterprise or on computers owned by a number of business partners. A service might or might not be a component in the sense of encapsulated software. Like a class object, the requester application is capable of treating the service as one. Web services are based on invocation using SOAP messages which are described using WSDL over a standard protocol such as HTTP. Use of web services is a best practice when communicating with external business partners [IBM, 2017].
Data-centric publish subscribe (DCPS) mechanism	Data-centric publish subscribe (DCPS) mechanism defines the functionality used by an application to publish and subscribe to the values of data objects. It allows: • Publishing applications to identify the data objects they intend to publish, and then provide values for these objects. • Subscribing applications to identify which data objects they are interested in, and then access their data values. • Applications to define topics, to attach type information to the topics, to create publisher and subscriber entities, to attach QoS policies to all these entities and, in summary, to make all these entities operate. Listeners and conditions (in conjunction with wait-sets) are two alternative mechanisms that allow the application to be made aware of changes in the DCPS communication status [IBM c), 2015].

Term	Definition
Architectural style	A family of systems in terms of a pattern of structural organization. Definition two: a characterization of a family of systems that are related by sharing structural and semantic properties [ISO/IEC/IEEE 24765:2010, 2010].
Architecture	Fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution. According to standard ISO/IEC 15288:2008 (IEEE Std 15288- 2008), Systems and software engineering architecture is the organizational structure of a system or component. Syn: architectural structure [ISO/IEC/IEEE 24765:2010, 2010]
Unified Component Model for Distributed, Real-Time And Embedded Systems	Standard Unified Component Model for Distributed, Real-Time And Embedded Systems defines necessary procedures for communication between components. Structure of distributed software system is represented by means of packages. Data exchanging are performed by special contracts. [Data Distribution Service (DDS), 2015] The contract package holds the definitions of contracts for UCM applications. Contracts mainly cover the definitions of interfaces and data types and package that defines a meta-model for standard data types. The contract package gathers several classes. A set of standard data types is defined; it is also possible to create metamodel extensions in order to define additional data types. Constants define specific values for a declared data type. Interfaces define consistent sets of methods related to a given service. The contract package also provides mechanisms to support the characterization of business and platform elements, using annotations and configuration parameters. Among those declarations, only data types and interfaces are considered as types and can be used to specify interactions between components. Constants and

Term	Definition
	exceptions are used to enrich the domain application specifications but do not directly contribute as the definition of contracts of interaction between components. Annotations and configuration parameters are used to decorate declarations. [Data Distribution Service (DDS), 2015]
Client	the code or process that invokes an operation on an object. ISO/IEC 19500-2:2003, Information technology — Open Distributed Processing — Part 2: General Inter-ORB Protocol (GIOP)/Internet Inter-ORB Protocol (IIOP). a node, cluster or capsule, which: a) contains a basic engineering object corresponding to a computational client object; and b) contains, or is potentially capable of containing, stub, binder and protocol objects in a channel supporting operations involving the client object. (ISO/IEC 14752:2000, Information technology) [Data Distribution Service (DDS), 2015]
Event	Occurrence of a particular set of circumstances. According to ISO/IEC 16085:2006 (IEEE Std 16085-2006), Systems and software engineering event is an external or internal stimulus used for synchronization purposes [ISO/IEC/IEEE 24765:2010, 2010].
Event synchronization	Control of task activation by means of signals. NOTE Three types of event synchronization are possible: external interrupts, timer expiration, and internal signals from other tasks. [ISO/IEC/IEEE 24765:2010, 2010]
Event-sequencing logic	a description of how a task responds to each of its message or event inputs. A communication sent from one object to another. IEEE Std 1320.2-1998 (R2004) IEEE Standard for Conceptual Modeling Language Syntax and Semantics for IDEF1X97 (IDEFobject) [ISO/IEC/IEEE 24765:2010, 2010].
Message	a communication sent from one object to another. IEEE Std 1320.2-1998 (R2004)

Term	Definition
	IEEE Standard for Conceptual Modeling Language Syntax and Semantics for IDEF1X97 (IDEFobject). [ISO/IEC/IEEE 24765:2010, 2010]
Metamodel	a logical information model that specifies the modeling elements used within another (or the same) modeling notation. IEEE Std 1175.1-2002 (R2007) IEEE Guide for CASE Tool Interconnections — Classification and Description. a metamodel Vm for a subset of IDEFobject is a view of the constructs in the subset that is expressed using those constructs such that there exists a valid instance of Vm that is a description of Vm itself. (IEEE Std 1320.2-1998 (R2004) IEEE Standard for Conceptual Modeling Language Syntax and Semantics for IDEF1X97 (IDEFobject)). a model containing detailed definitions of the meta-entities, meta-relationships and meta-attributes whose instances appear in the model section of a CDIF transfer. ISO/IEC 15474-1:2002, Information technology specification of the concepts, relationships and rules that are used to define a methodology. ISO/IEC 24744:2007, Software Engineering — Metamodel for Development Methodologies.3.4. Syn: meta-model [ISO/IEC/IEEE 24765:2010, 2010]

Related papers

Paper [Sharmaa, 2015] describes main architectural styles, that are used for designing of distributed applications. It but lacks from recommendations in which situations which architectural style will be applied.

Paper [Ciuffoletti, 2015] proposes a foundation for choosing of architectural styles for distributed systems.

Usually for designing distributed applications combination of architectural styles is used.

Layered architectural style should be used in application which can be divided into several layers on the basis of services or functionalities in such a way that lower layer forms the basis for upper layer.

Division of system into layers leads to easy debugging and flexible to update and maintain in future. Cloud computing architecture is for cloud based applications like Gmail drive, Google App Engine. It has limited functionality since till now it has not covered all the requirements of information technology. Higher scalability is the main feature of this architecture [Sharmaa, 2015].

Use of Internet is increasing day by day results in heavily used of client server architectural styles in the web based applications. Some examples of web based software are: AceProject, Ganttic, Celoxis etc. For a server based application supporting many clients which is going to be used by a web browser uses client server architecture model. It is also applicable in application area for centralized resource system (like management functions, storage) which is going to be used by many clients. Heavily used and famous example for client server architecture application area is email, World Wide Web, FTP software, e-commerce applications. The complexity of such applications is not as high as blackboard based applications. Client server based applications keeps high functionality and efficiency. 3-tier client server applications have good reliability because of middle tier which perform all security related task [Sharmaa, 2015].

Multi-tier architectural style – is used to distribute application functions on different physical components.

In more simple variant client-server architecture is used. It is used when it is enough to collaborate data exchanging mechanism. Often client-server architecture is a part of n-tire architectural style.

Service oriented architecture – is used to simplify code reuse procedure and when it is necessary to exchange data thought central component, namely service bus.

Paper [Ciuffoletti, 2015] proposes two different approaches to virtualization:

• a container-based approach evolves is grounded on using of separate resources for performing some operation. Also during execution services use isolated memory and computation resources. Other approach, namely hypervisor technique provides performing software functions by means of memory and resources of operating system. Authors make a conclusion that the realization of complex but agile distributed architectures, composed of small and specialized services: the microservice approach is a promising design paradigm that is tightly bound to (or merging with) the container technology.

• another uses the lower layers of the running operating system to implement one or more containers, that are isolated environments.

Task

To design an approach for distributed application architecture choosing. In order to do this it is necessary to do the following:

- systematize both functional and non-functional requirements for designing distributed application;.
- propose architectural solutions of notification engine considering different user requests;
- define a stack of technologies for supporting every architectural solution;
- estimate cost of various architectural solutions implementing;
- propose an approach for architecture of distributed applications choosing.

Summary of functional and non-functional features of designed system

Functional features

The system should provide high level stability during high load on the servers, provide scalability for the solution (based on the number of records in the consumer table of the GMB market DB, number of active markets, and number of active events per market), message durability and ability to re-execute the job even if the job failed.

The system should provide the following main functionality:

- ability to add new triggers for events;
- ability to start/stop triggers for events;
- ability to remove triggers;
- ability to update trigger's configuration;
- ability to generate messages for job execution;
- ability to write messages to queue (send request);
- ability read messages from queue (receive request);
- ability to execute job according to the message data;
- ability to send e-mail notifications;
- ability to send push notifications;
- ability to generate DB notifications.

The main notification engine execution flow should consist of next steps:

- scheduler should initiate trigger work;
- trigger process its internal logic;
- after processing, trigger should generate necessary message for handler;
- generated messages should be pushed to the dedicated queue (name of the dedicated queue should be configured);
- the queue stores the message until it will be read and processed by the handler;
- handler each time monitors the queue for new available messages and reads them;
- after message was read, handler process message according to the message data;
- after successful processing, handler should remove the message.

Non-functional features

In addition to the main functional requirements, the system should provide additional non-functional requirements that contains:

- as the whole GMB system deployed on the Amazon Web Services (AWS), NE should be also deployed on this environment;
- the system should be possible to be moved to another hosting environment, without providing any changes for the existing functionality;
- NE should be independent part of the GMB;
- NE should not influence business logic of the main part of the GMB (backend services).

Distributed system design

At first need to be noted, that NE will be designed as a distributed application. So the system will consist of two parts. The first part will be called Event Processor (EP), and the second one will be called Job Processor (JP). According to the provided functional and non-functional requirements, four different application architectures were designed.

The first architectural solution

Based on the data about the system in general and architectural patterns NE will consist of two separated parts: Event Processor (EP) and Job Processor (JP).

Main definition of Events, Event processing architecture, Event source and event comsumers are represented at [IBM c), 2015].

Principles of organizing job processor are represented on [IBM d) 2015]. It is proposed to design EP is designed as a simple service. JP is designed as a web service, and will have specific URL. This URL will be the same for all markets.

For each market and for each application in the market, which will use NE, event processor will load application plugin. Plugins will be connected to the EP using Managed Extensibility Framework. Each plugin consists of a number of triggers, which will execute specific events (jobs) every 15 seconds (time will be configured in the plugin application configuration file and could be changed for a specific request from the client side).

Each trigger could access to the DB through a separate library called Framework Module. Framework Module – is a custom library, designed especially for code reusing in different modules of the GMB system. It encapsulates DB models (for both market and administrative DBs), generated by Entity Framework (EF), specific constants, general helper methods, methods to send HTTP requests etc. All the data will be stored in the MS SQL Server DBMS (because GMB already has all data in this DBMS).

After successful execution of the trigger, if it is necessary, trigger will send result to EP in order to initiate EP to send HTTP request with necessary information to JP, for further data processing (send push or e-mail notification, or generate DB notification). JP processes request from the EP and sends push notifications through Push Sharp API, sends e-mail notifications through simple mail client, and stores DB notifications in the

The design of the notification engine architecture is shown on the Figure 1.Architectural features

In order to provide updates of the trigger configuration, NE should be restarted at all. Such realization does not support “on flight” reconfiguration of the triggers. All of them are independent from each other, but if the EP fails, all triggers will not work at all. In order to start them working again, the restart of the service is needed.

Figure 2. shows EP functional requirements, and Figure 3 shows JP functional requirements shows covering of the functional requirements.

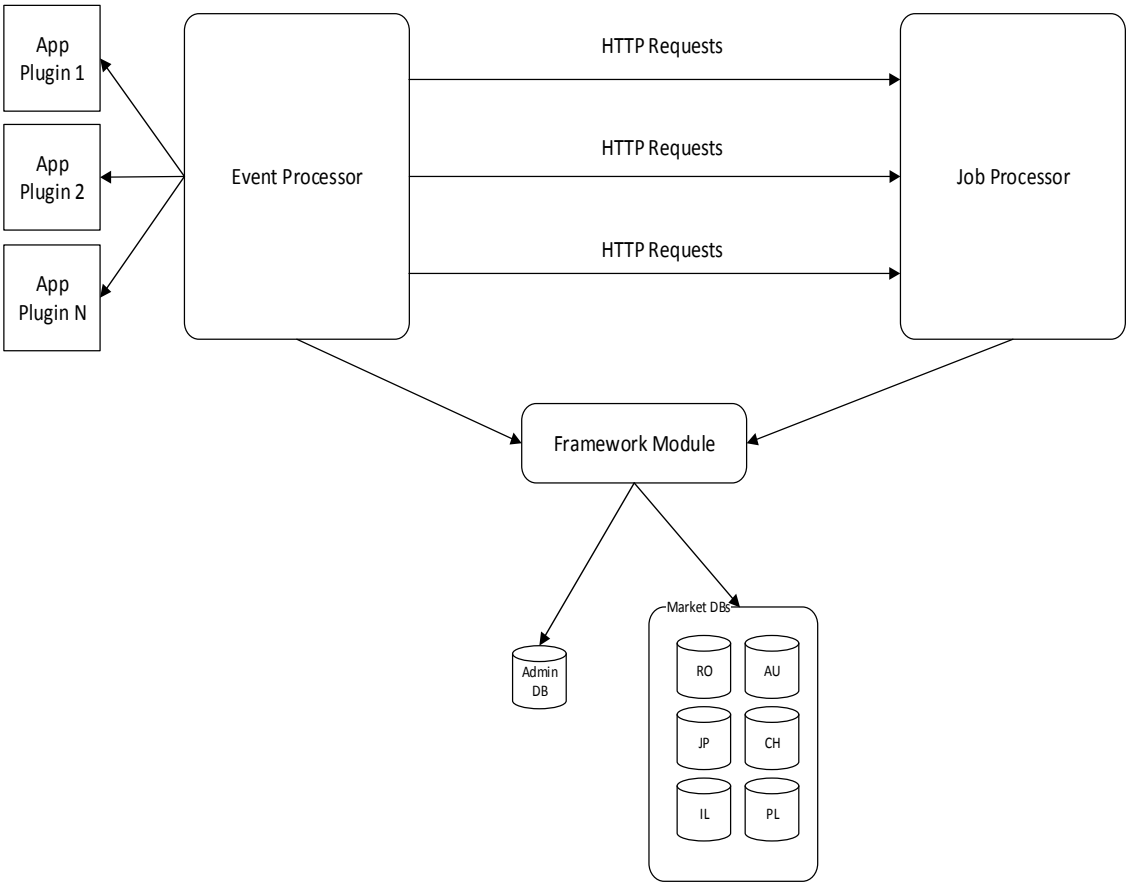


Fig. 1. The first architectural solution of notification engine

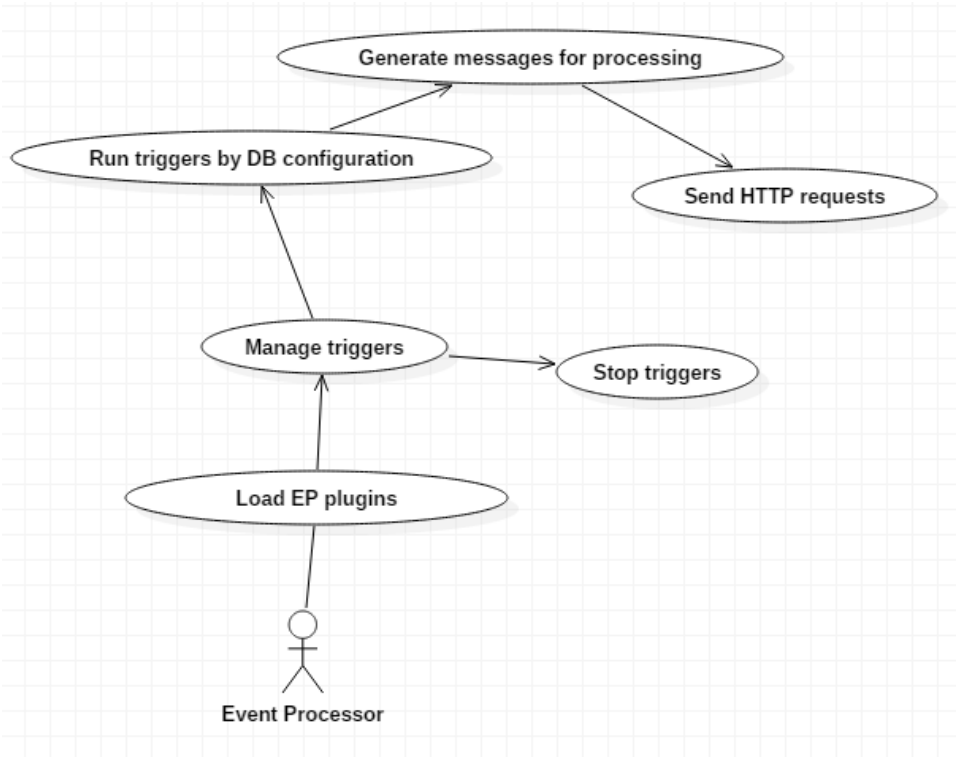


Fig. 2. Use case diagram of event processor

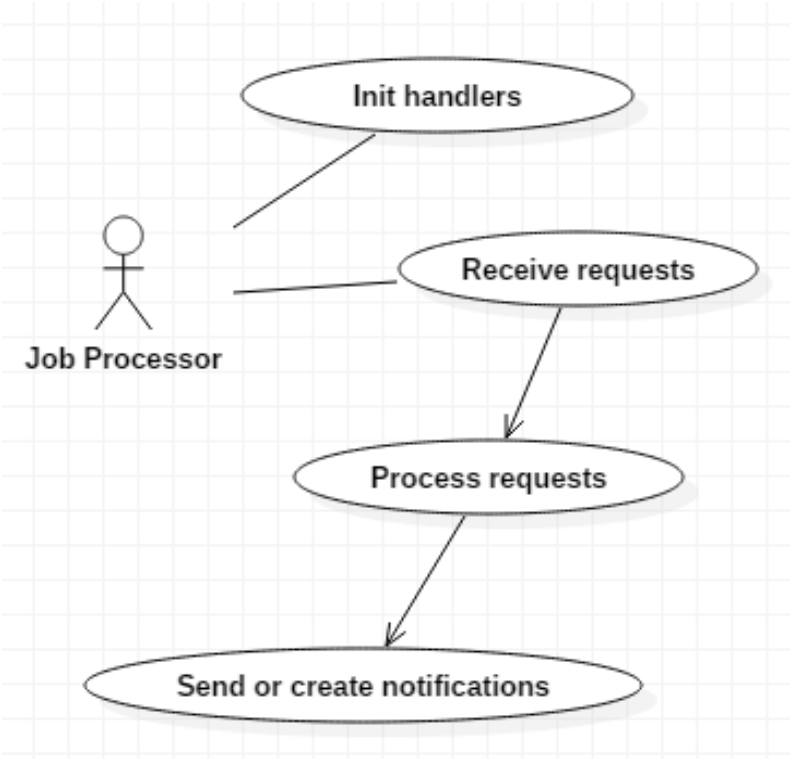


Fig. 3. Use case diagram of job processor

Stack of technologies for the first architectural solution

1. Windows service, used as a core principle of the EP creation.
2. Web service, used as a core principle of the JP creation.
3. Managed Extensibility Framework (MEF), used by EP to load plugins.
4. Entity Framework (EF), used by framework module, to provide access to the market and administrative DBs.
5. Hypertext Transfer Protocol (HTTP), used to send requests from EP to JP, for further processing.
6. PushSharp API used to send push notifications.

Advantages and disadvantages of the first architecture

Advantages of the designed architecture:

– The system could be migrated from one hosting environment to another without any changes in the source code of the NE.

– As JP is a Web service, it could be deployed using auto scaling technologies, just routing service need to be added (it also does not need any changes in the source code).
– The system could be deployed in different hosting environments (for example: EP on the Microsoft Azure, and JP on the Amazon AWS).

Disadvantages of the designed architecture:

– JP is a Web service, so it needs additional payment for domain address and external IP address.
– HTTP requests is not good solution for communication between EP and JP because in some moments of high load EP could generate so many requests, so that JP (IIS service on the Windows OS) will break down due to overload (the number of maximum parallel threads and connections is limited by the IIS). This case does not provide message durability.
– As this architecture supposes main data processing on the EP side, and JP just executes sending of the notifications leads to that point that all main functionality locates on the EP and its plugins. As all triggers process data in asynchronous way, during the high load on the system, triggers will have long timeouts for connections to the DB or external services. It caused because the number of threads is limited in the machine, on which the service is deployed.
– In order to update configuration of the triggers, NE must be restarted. Reload is a very bad case for NE, because EP need to reload all plugins for all markets and this process is rather long. The restart time for a Windows service is approximately from 3 to 10 minutes (based on the number of plugins).

The second architectural solution

The second version of the architecture of the NE is more complex, than the first one, and it has its own peculiarities.

EP is designed as a number of independent event generators (lambda functions). They are designed using AWS Lambda. Each trigger in the EP will be presented as a Lambda function. JP is designed as a windows service with its plugins.

Each AWS Lambda function starts by scheduler. Each scheduler has its own configuration (configuration stored in the AWS profile for each lambda function), which could be changed from the AWS management console.

The communication between AWS Lambda and JP will be provided through the message bus. The message bus will be durable – if any issues arise, messages will be retained and processed when services are back online. The AWS Simple Queue Service (SQS) will be used as message-bus service. Each Lambda function will send messages to the dedicated queue (names of queues can be configured from AWS management console).

JP will listen all the available queues in the SQS and read messages from them. For each market and for each application in the market, which will use NE, job processor will load application plugin. Plugins will be connected to the JP using MEF. Each plugin consists of a number of handlers, which will execute specific jobs according to the data, provided in the message.

The JP has one job plugin for each application and one default plugin. The design pattern command will be used to execute dedicated functionality for each job. In order to provide command pattern functionality a special Dispatcher class was designed in the JP. Dispatcher task is to subscribe to a dedicated queue, read message from the queue, and pass this messages to a plugin for further execution. Per one queue will exist only one dispatcher instance. Each dispatcher works in its own thread.

DB storage will be moved from MS SQL Server DBMS to the AWS Dynamo DB. Dynamo DB will significantly increase the speed for the read and write requests, because it is NoSQL, object oriented database. Administrative DB will be removed from the system at all, instead of it, a new table with available markets will be created independently in the Dynamo DB storage.

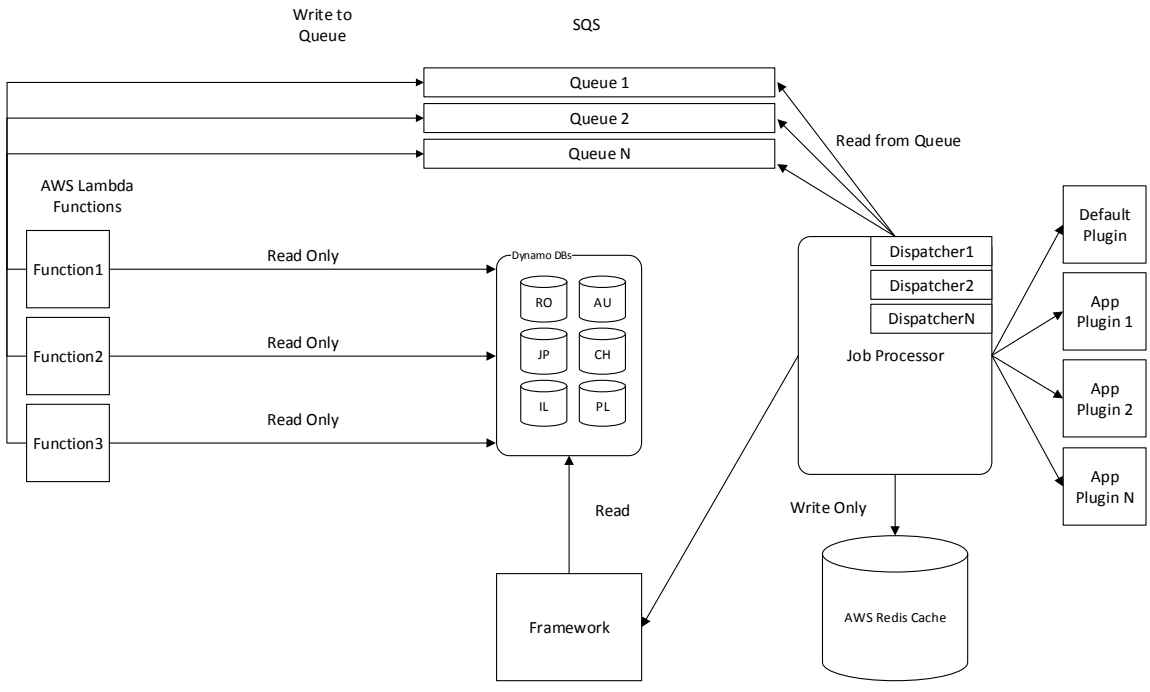


Fig. 4. The second architectural solution of notification engine

AWS Lambda supports direct access from the function code to the AWS SQS and Dynamo DB, so any new framework does not need, as a result, the creation of the new lambda function will not be a problem.

Each handler in JP could access to the DB through a separate library called Framework Module, but only for the read operation. Framework module need to be changed in order to support access to a new DB instance. In order to provide access to the AWS DB and SQS services, AWS Toolkit need to be installed. In addition, direct access to the Dynamo DB from the framework module will be provided through Linq2DynamoDB framework (analog of EF). After successful execution of the handler in JP plugin, if it is necessary, handler sends push or e-mail notification, or generate DB notification, or just stops. Push notifications will be sent through Push Sharp API, e-mail notifications will be sent through simple mail client, and DB notifications will be saved in the ElastiCache Redis, which provided by AWS.

As DB notifications need to be accessible very fast (request timeout no more than 100ms for a good 100 Mbit Internet connection), was suggested to use the fastest object oriented data storage – Redis Cache.

The design of the notification engine architecture is shown on the Figure 4.

In order to provide updates of the trigger configuration, administrator of the system should just change event data for each Lambda function. All necessary things for updating this data are available in the AWS Management Console. Such realization supports “on flight” reconfiguration of the triggers. All of them are independent from each other.

JP handlers are always loaded into the system, and they don’t need to be reloaded. Handlers task is to execute job according to the message data, they don’t depend on any configuration. If JP service fails, all the data which was not processed will stay in queues, and in order to restore the work of the service, it should be restarted at all.

The next use-case diagrams. Figure 5 shows EP functional requirements, and Figure 6 shows JP functional requirements) show covering of the functional requirements.

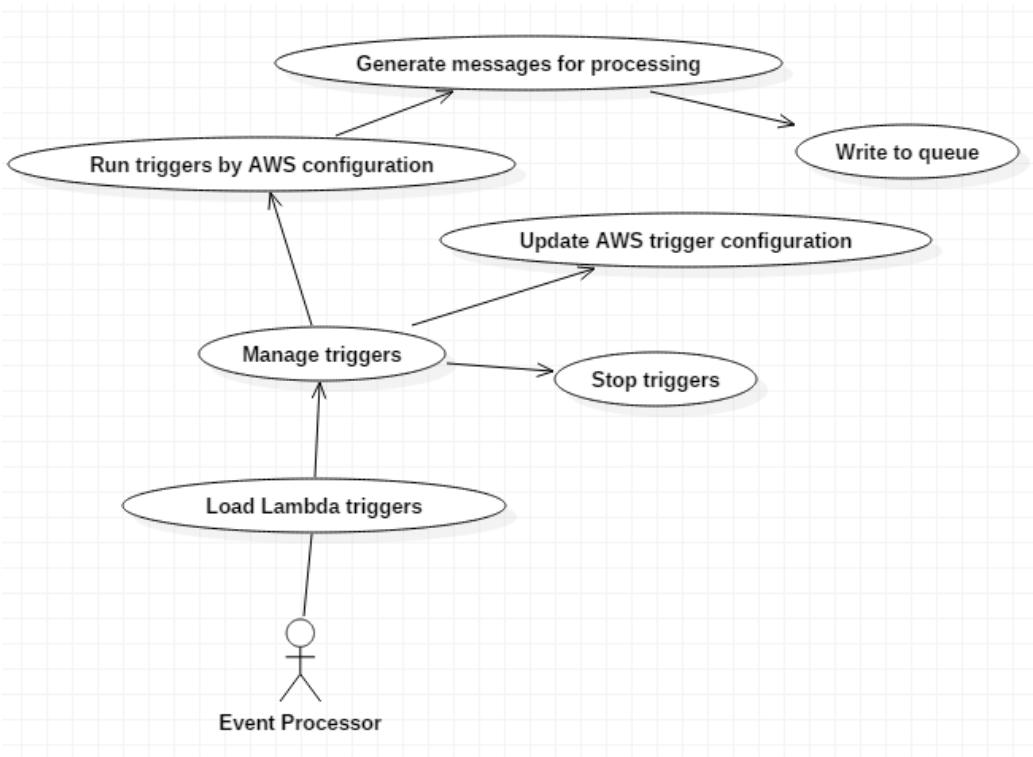


Fig. 5. Event Processor use case diagram

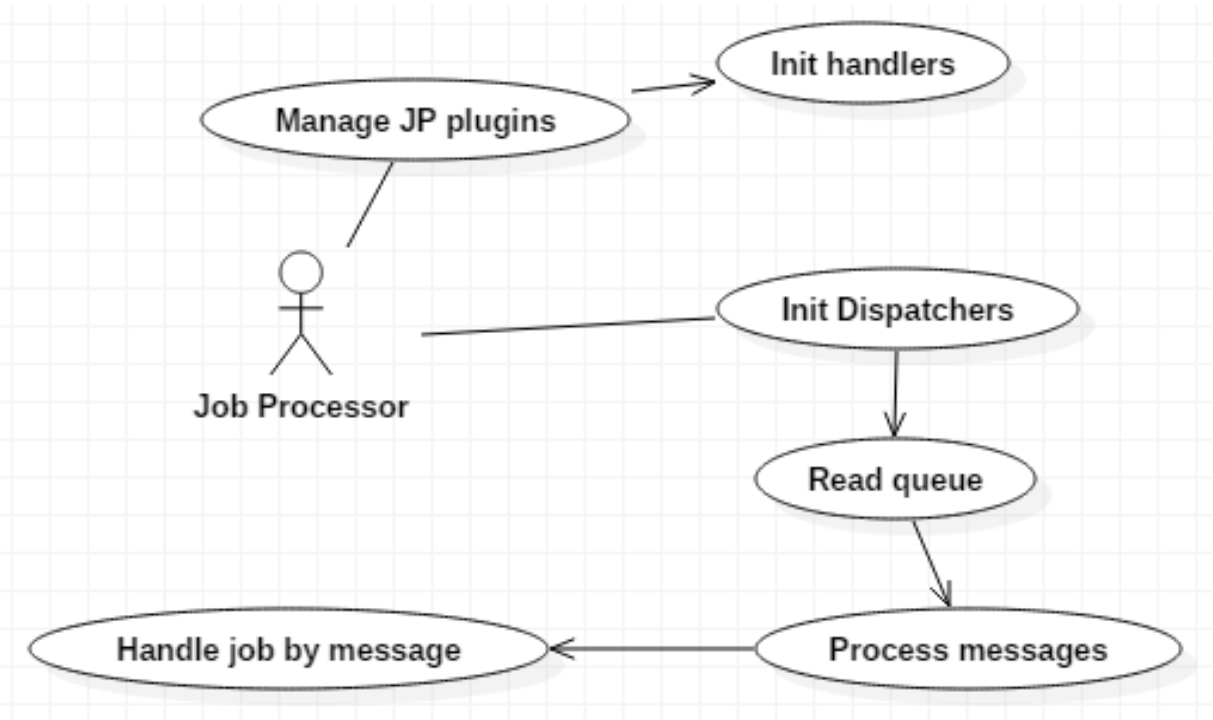


Fig. 6. Job Processor use case diagram

Stack of technologies for realization of the second architectural solution

Stack of technologies, which will be used for this architecture:

- AWS Lambda is used as EP triggers.
- Windows service is used as a core principle of the JP creation.
- Managed Extensibility Framework (MEF), is used by EP to load plugins.
- AWS SQS is used as a message bus between EP and JP.
- AWS Dynamo DB is used as main data storage.
- Linq2DynamoDB (analog of EF), is used by framework module, to provide access to the market DBs.
- Command pattern is used for JP to provide queue reading.
- PushSharp API is used to send push notifications.
- AWS ElastiCache Redis is used to store DB notifications.

Advantages and disadvantages of the second architectural solution

Advantages of the designed architecture:

- EP part is designed as independent functions on the AWS side. It means that AWS will scale up or scale down these functions according to the load, provided on them. It is fully independent part of the NE system.
- The message bus is supported by using SQS. SQS is also independent part of the NE, and it also stores on the AWS side. SQS guarantees, that a message will be delivered at least one time, so it provides durability of the communication. Also AWS will scale up or scale down SQS based on the load provided on it.
- Usage of Dynamo DB instead of SQL Server DB is also a great advantage. NoSQL data storage will provide speed up inserting and key-search mechanisms. As a result, access to the data and data saving will be faster, so EP and JP will have lower timeouts for DB connections.
- Usage of the Redis cache will speed up GMB access to DB notifications, because Redis always stores all data in the RAM. Also inserting process also will be faster, because application does not need to wait while the record will be stored on the file system.
- Updates for trigger's configuration will apply without any restarts of the services. This is possible because configuration is needed only for triggers on the EP side, and AWS Lambda supports real-time reconfiguring.

– In order to add new trigger, administrator should just create new lambda function in the AWS management console and provide correct configuration for this function. This operation does not need any restart for other lambda functions or JP service.

Disadvantages of the designed architecture:

– The biggest disadvantage is that using AWS Lambda, SQS, Dynamo DB, and Redis will limit NE on deployment environments. It means that, if NE migrates to another hosting place (for example Microsoft Azure) a lot of functionality will not work at all. But system could be divided by different hosting places (for example, EP with Lambda, SQS, Dynamo DB, and Redis will still work on AWS, and JP will work on Azure). In this case the problem could be in the timeouts during connections between system parts, and it is also bad case for CRM system. As a result, time of execution of the particular function could be increased in 10-15 times.
– AWS Lambda has limitations for time of execution and for the number of executions. It means that customer should additionally pay for the number of executions and if the execution time of the function became longer than available, function will fail and execution will not provide any result.
– Usage of a number of external (in our case AWS resources) could lead to the problems of management of the system at all. Because configuration of each lambda function is stored in different places, configuration of the SQS is stored in another place, etc.
– In order to use Dynamo DB, firstly, for each environment old data need to be migrated from MS SQL Server to Dynamo DB. This process needs additional time for realization.
– Dynamo DB is a very fast data storage, but it has limitations for provision throughputs (number of units for read and write per second). If these limitations will be exceeded, the application, which sends requests to the DB will receive error: provision throughput exceeded. As a result, data processing will stop for a period of time. Throughput can be changed, but it is rather expensive.

The third architectural solution

The third version of the architecture of the NE is absolutely based on the AWS components, and differs from the first architectures by absence of custom services. Such kind of architecture has its own peculiarities.

EP is designed as a number of independent event generators (lambda functions). They are designed using AWS Lambda. Each trigger in the EP will be presented as a Lambda function.

JP is also designed as a number of independent lambda functions (handlers). Each handler in the EP will be presented as a Lambda function.

Each AWS Lambda function starts by scheduler. Each scheduler has its own configuration (configuration stored in the AWS profile for each lambda function), which could be changed from the AWS management console.

The communication between AWS Lambda and JP will be provided through the message bus. The message bus will be durable – if any issues arise, messages will be retained and processed when services are back online. The AWS Simple Queue Service (SQS) will be used as message-bus service. Each Lambda function will send messages to the dedicated queue (names of queues can be configured from AWS management console).

Each JP handler will listen dedicated queue in the SQS and read messages from it. For each market and for each application in the market, which will use NE, JP will have lambda handlers. Each handler will execute specific jobs according to the data, provided in the message.

DB storage will be moved from MS SQL Server DBMS to the AWS Dynamo DB. Dynamo DB will significantly increase the speed for the read and write requests, because it is NoSQL, object oriented database. Administrative DB will be removed from the system at all, instead of it, a new table with available markets will be created independently in the Dynamo DB storage.

AWS Lambda supports direct access from the function code to the AWS SQS and Dynamo DB, so any new framework does not need, as a result, the creation of the new lambda function will not be a problem.

Also EP will have special lambda function(s) which will have access to AWS Simple Notification Service (SNS) in order to provide direct access for sending push and e-mail notifications. DB notifications will be sent by EP trigger to the SQS as a message, where it will be read by handler later. Handler will process this message and DB notifications will be saved in the ElastiCache Redis, which provided by AWS.

As DB notifications need to be accessible very fast (request timeout no more than 100ms for a good 100 Mbit Internet connection), was suggested to use the fastest object oriented data storage – Redis Cache.

The design of the notification engine architecture is shown on the Figure 7.

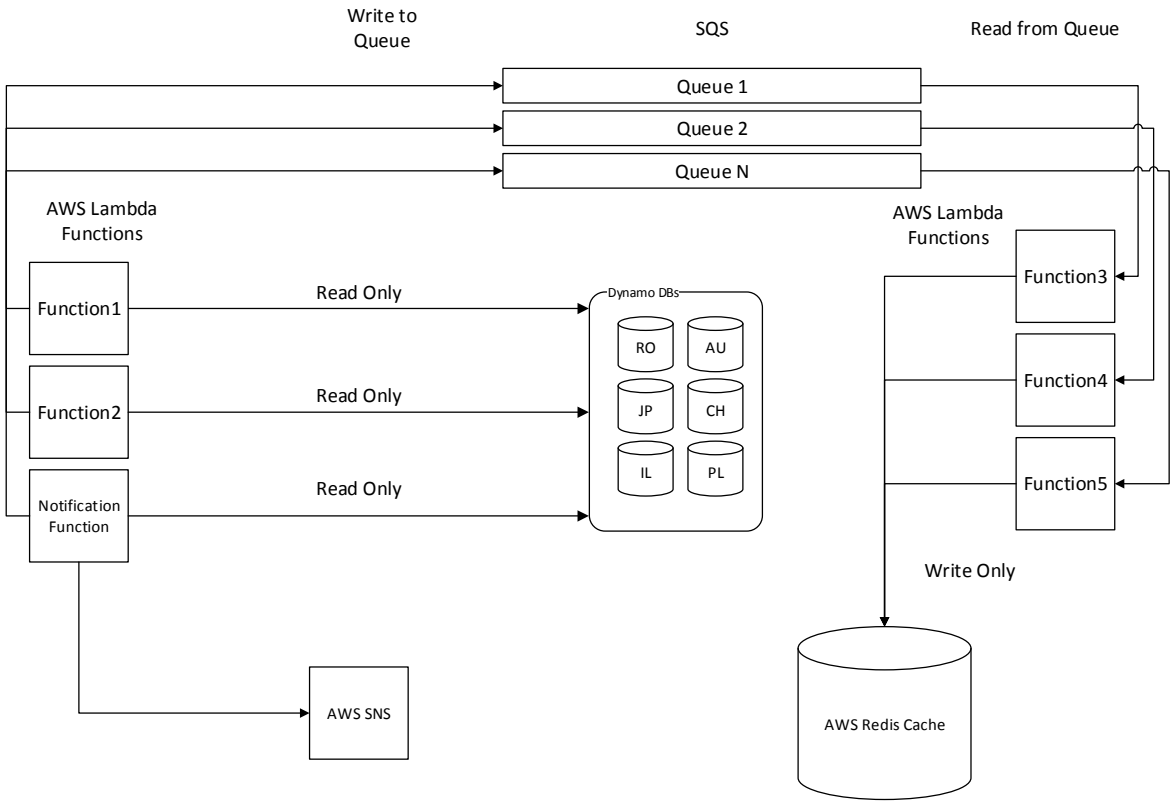


Fig. 7 The third architectural solution of notification engine

Architectural features

Stack of technologies

Stack of technologies, which will be used for this architecture:

1. AWS Lambda is used as EP triggers and JP handlers.
2. AWS SQS is used as a message bus between EP and JP.
3. AWS Dynamo DB is used as main data storage.
4. AWS SNS is used to send push and e-mail notifications.
5. AWS ElastiCache Redis is used to store DB notifications.

Advantages and disadvantages of the architecture

Advantages of the designed architecture:

- The greatest advantage of such architecture is that all parts of the NE are designed in such way, that they don't depend on physical deployment environments. All parts of the system (EP, JP, SQS, DynamoDB, Redis cache) are auto scalable and supported by AWS. This architecture provides the best solution for highly loaded CRM systems.

- EP an JP part is designed as independent functions on the AWS side. It means that AWS will scale up or scale down these functions according to the load, provided on them. It is fully independent part of the NE system.

- The message bus is supported by using SQS. SQS is also independent part of the NE, and it also stores on the AWS side. SQS guarantees, that a message will be delivered at least one time, so it provides durability of the communication. Also AWS will scale up or scale down SQS based on the load provided on it.

- Usage of Dynamo DB instead of SQL Server DB is also a great advantage. NoSQL data storage will provide speed up inserting and key-search mechanisms. As a result, access to the data and data saving will be faster, so EP and JP will have lower timeouts for DB connections.
- Usage of the Redis cache will speed up GMB access to DB notifications, because Redis always stores all data in the RAM. Also inserting process also will be faster, because application does not need to wait while the record will be stored on the file system.

- Updates for trigger's configuration will apply without any restarts of the services. This is possible because configuration is needed only for triggers on the EP side, and AWS Lambda supports real-time reconfiguring.

- In order to add new trigger or handler, administrator should just create new lambda function in the AWS management console and provide correct configuration for this function. This operation does not need any restart for other lambda functions or JP service.

Disadvantages of the designed architecture:

- The biggest disadvantage is that this architecture absolutely depends on the resources of the Amazon services. If the GMB migrates to another hosting, NE system will not work with GMB at all. Of course if GMB migrate to another big vendor of cloud computing (such as Microsoft Azure), the NE also should be modified in order to support the same functionality, but on the side of Azure. For example, Azure has its own cloud functions (analog of AWS Lambda), Azure table storage (analog of AWS Dynamo DB) etc.

- Usage of a number of external (in our case AWS resources) could lead to the problems of management of the system at all. Because configuration of each lambda function is stored in different places, configuration of the SQS is stored in another place, etc.

- In order to use Dynamo DB, firstly, for each environment old data need to be migrated from MS SQL Server to Dynamo DB. This process needs additional time for realization.

- Dynamo DB is a very fast data storage, but it has limitations for provision throughputs (number of units for read and write per second). If these limitations will be exceeded, the application, which sends requests to the DB will receive error: provision throughput exceeded. As a result, data processing will stop for a period of time. Throughput can be changed, but it is rather expensive.

Estimation of distributed application development

In order to estimate the cost of implementing of proposed architectural solutions refer to PNN Soft company practices of counting of economical justifications for different architectural solutions realization. The data for the costs of the deployment environment where provided by Amazon AWS, for EU west-1 data center for Windows OS.[]

The first architectural solution

The table below shows necessary resources, needed to develop and deploy the notification engine:

Table 2. Economical justification for the first architectural solution

№	Name	Number of needed resources/prices	Costs	Total
1	Developers	3 developers	25 ($/h)	75($/h)
2	QA Engineers	2 QAs	20 ($/h)	40 ($/h)
3	Development time	120 hours	75($/h)	9000$
4	Time for testing	40 hours	40 ($/h)	1600$
5	AWS dev environment	160 hours, 2 machines	0.034 ($/h)	10.88$
6	AWS staging environment	160 hours, 2 machines	0.034 ($/h)	10.88$
7	AWS QA environment	40 hours, 2 machines	0.258 ($/h)	20.64$
8	AWS prod environment	20 hours, 2 machines	0.517 ($/h)	20.68$
9	Domain address	160 hours, 4 items	13 ($/year)	3.5$

Total number of team members for NE is: 5 people (3 developers, 2 QA engineers).

Total time for the development and testing is: 160 hours.

Total price of NE in current approach is: 10 666.58$.

Table 2. Economical justification for the second architectural solution

№	Name	Number of needed resources/Prices	Costs	Total
1	2	3	4	5
1	Developers	3 developers	25 ($/h)	75($/h)
2	QA Engineers	2 QAs	20 ($/h)	40 ($/h)
1	2	3	4	5
3	Development time	130 hours	75($/h)	9750$
4	Time for testing	40 hours	40 ($/h)	1600$
5	AWS dev environment	170 hours, 1 machine	0.034 ($/h)	5.44$
6	AWS staging environment	170 hours, 1 machine	0.034 ($/h)	5.44$
7	AWS QA environment	30 hours, 1 machine	0.258 ($/h)	10.32$
8	AWS prod environment	20 hours, 1 machine	0.517 ($/h)	10.32$
9	AWS SQS	3000000 requests, 4 queues	0.00000040 ($/request)	14.4$
10	ElastiCache Redis	170 hours, 4 instances	$0.638	433.84$
11	DynamoDB tables	170 hours, 20 tables (for 1 environment), 4 environments	0.0145 ($/h) 0.0029 ($/h)	236.64$
12	Lambda functions	10 functions (for 1 environment), 4 environments	$9.13 ($/month)	365.2$

Total number of team members for NE is: 5 people (3 developers, 2 QA engineers).

Total time for the development and testing is: 170 hours.

Total price of NE in current approach is: 12431.6$.

The data for the development and testing costs were provided by software development company “PNN Soft”.

The data for the costs of the deployment environment where provided by Amazon AWS, for EU west-1 data center for Windows OS.

Table 4. Economical justification for the third architectural solution

№	Name	Number of needed resources/Prices	Costs	Total
1	2	3	4	5
1	Developers	2 developers	25 ($/h)	50($/h)
2	QA Engineers	2 QAs	20 ($/h)	40 ($/h)
1	2	3	4	5
3	Development time	90 hours	50($/h)	4500$
4	Time for testing	30 hours	40 ($/h)	1200$
5	AWS SQS	3000000 requests, 4 queues	0.00000040 ($/request)	14.4$
6	ElastiCache Redis	120 hours, 4 instances	$0.638	306.24$
7	DynamoDB tables	120 hours, 20 tables (for 1 environment), 4 environments	0.0145 ($/h) 0.0029 ($/h)	167.04$
8	Lambda functions	20 functions (for 1 environment), 4 environments	$9.13 ($/month)	730.4$

Total number of team members for NE is: 4 people (2 developers, 2 QA engineers).

Total time for the development and testing is: 120 hours.

Total price of NE in current approach is: 6 918.08$.

The data for the development and testing costs were provided by software development company “PNN Soft”. The data for the costs of the deployment environment where provided by Amazon AWS, for EU west-1 data center for Windows OS.

Conclusion

Approach for choosing architectural solution for distributed application designing is proposed in this paper. Systematization of terminology allows to organize collaboration between software architectures and researches in sphere of facilitation software components and services reuse. Also typical requirements for distributed application tasks are systematized. Advantages and disadvantages of different architectural solutions let estimate limitations of and effectiveness of future software. Functional and non-functional requirements for distributed applications allow to refine software requirement specification of future software. Then according to project budget it is defined if proper architectural solution for future project. Proposed advantages and disadvantages for every architectural solution let estimate cost of supporting and deployment processes.

Further works

Design a formal method for defining constrains from text of requirement specification. Then gather collections of features identifying every architectural solution. Prepare profile for problem domain “distributed application architectures”. Estimate this profile by following SOLID design principles using the approach, proposed in[].

Using this profile and constrains propose a method for requirement specification elicitation.

Bibliography

[Sharmaa, Kumarb, Agarwal, 2015] Anubha Sharmaa, Manoj Kumarb ,Sonali Agarwal A Complete Survey on Software Architectural Styles and Patterns Proceedings of the 4th International Conference on Eco-friendly Computing and Communication Systems. Published by Elsevier Volume 70, 2015, pp. 16–28 doi: 10.1016/j.procs.2015.10.019

http://ac.els-cdn.com/S187705091503183X/1-s2.0-S187705091503183X-main.pdf?_tid=9be93464-dc31-11e6-aded-00000aacb35f&acdnat=1484601810_4a7b6416ce69969da4b27cf755c2dfa3

[Chebanyuk and Markov, 2016] Elena Chebanyuk, Krassimir Markov An approach to class diagram verification according to SOLID design principles. In: MODELSWARD 2016, Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development. Edited by S.

Hammoudi, L.F. Pires, B. Selic and P. Desfray. SCITEPRESS – Science and Technology Publications, Lda. Portugal, 2016. ISBN: 978-989-758-168-7. pp. 435-441.

DOI: 10.5220/0005830104350441

[Ciuffoletti, 2015] Augusto Ciuffoletti Automated deployment of a microservice-based monitoring infrastructure rs. Procedia Computer Science. Published by Elsiver. Volume 68 (2015) pp. 163 – 172 doi: 10.1016/j.procs.2015.09.232

http://ac.els-cdn.com/S187705091503077X/1-s2.0-S187705091503077X-main.pdf?_tid=937c55e2-dc2f-11e6-8f44-00000aab0f01&acdnat=1484600937_7d6e20f713eac3e85c067f9964396752

[Data Distribution Service (DDS), 2015] Data Distribution Service (DDS)

http://www.omg.org/spec/DDS/1.4/

[IBM a), 2015] IBM Knowledge Center

http://www.ibm.com/support/knowledgecenter/SSGMCP_4.2.0/com.ibm.cics.ts.eventprocessing.doc/concepts/dfhep_definition.html

[IBM b), 2015] IBM Knowledge Center

http://www.ibm.com/support/knowledgecenter/en/SSETD4_9.1.3/lsf_admin/job_processor_binding.html

[IBM c), 2015] IBM Knowledge Center

http://www.ibm.com/support/knowledgecenter/SSGMCP_4.2.0/com.ibm.cics.ts.eventprocessing.doc/concepts/dfhep_definition.html

[IBM d), 2015] IBM Knowledge Center

http://www.ibm.com/support/knowledgecenter/en/SSETD4_9.1.3/lsf_admin/job_processor_binding.html

[IBM, 2017] Service-Oriented Architecture expands the vision of web services, Part 1

http://www.ibm.com/developerworks/webservices/library/ws-soaintro/ws-soaintro-pdf.pdf

[IEEE, 1998] IEEE Standard for Conceptual Modeling Language - Syntax and Semantics for IDEF1X97

https://standards.ieee.org/findstds/standard/1320.2-1998.html

[ISO/IEC 16085:2006, 2006] ISO/IEC 16085:2006 Systems and software engineering -- Life cycle processes -- Risk management http://www.iso.org/iso/catalogue_detail.htm?csnumber=40723

[ISO/IEC/IEEE 24765:2010, 2010] ISO/IEC/IEEE 24765:2010

https://webstore.ansi.org/RecordDetail.aspx?sku=ISO%2FIEC%2FIEEE%2024765:2010&source=google&adgroup=iso-iec-ieee&gclid=Cj0KEQiA_HDBRD2lomhoufc1JkBEiQA0TVMmr0KV9TkTcCQRCmhms_mfaLD2fTktvSgfjE7uBxcdBwaAmdA8P8HAQ

[Kangasharju, 2008] Jussi Kangasharju Distributed Systems: What is a distributed system? University of Helsinki https://www.cs.helsinki.fi/u/jakangas/Teaching/DistSys/DistSys-08f-1.pdf

[Microsoft, 2003] A Guide to Building Enterprise Applications on the .NET Framework https://msdn.microsoft.com/en-us/library/ms954601.aspx

[Microsoft, 2012] Components and Web Application Architecture https://msdn.microsoft.com/en-us/library/bb727121.aspx#mainSection

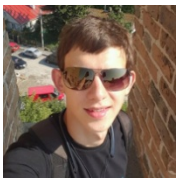
[Oxford, 2016] Oxford Living Dictionaries. Oxford University Press, 2016. https://en.oxforddictionaries.com/definition/us/distributed_system

[SAD, 2016] Software Architecture & Design Tutorial https://www.tutorialspoint.com/software_architecture_design/pdf/distributed_architecture.pdf

[Unified Component Model for Distributed, RealTime and Embedded Systems, 2013] Unified Component Model for Distributed, Real-Time and Embedded Systems RFP http://www.omg.org/cgi-bin/doc?mars/2013-09-10

Authors' Information

Elena Chebanyuk *– Software Engineering Department, National Aviation University, Kyiv, Ukraine,*
Major Fields of Scientific Research: *Model-Driven Architecture, Model-Driven Development, Software architecture, Mobile development, Software development,*
e-mail: chebanyuk.elena@ithea.org



Mykhailo Kostiuk - .Net software developer at PPN Soft, *Kyiv, Ukraine*
Major Fields of Scientific Research: Software development, Software architecture, Test-Driven Development
e-mail: hell.gunshe@gmail.com