Telenor Bulgaria RFI
Architecture details
Questions / Answers

Revision History

Version	Date	Author	Description
1.0	11.10.2018	Jiri Dolezel, Pavel Chlad	Original Document

Table of Contents

0. Solution overview

Emeldi Commerce® is a unified Omni-channel/CRM focused specifically on the Communication Service Provide (CSP) digital commerce space. Architected and designed specifically to address the needs of Communications Service Providers, Emeldi Commerce delivers e2e digital commerce and CRM functionality across all channels within the telecom commerce domain:

- Built on Microservices architecture
- Unified CPQ, product catalogue, service configuration
- Integration to telco domain with ready-made connectors which understand interoperational semantics
- Architecturally open applications, API
- Rich partner landscape (MIND CTI, NTS Retail, ZIRA, Valomnia, CROSS Network Management)
- Easily integratable to 3rd party apps
- Architecture: single Product Catalogue and Customer Accounts / Products across all channels

1. Digital customer journeys

The customer journeys provided through the Emeldi solution and partner portfolio has the main focus of enabling customers to engage with the CSP at any time required. Frictionless interactions between the digital and the physical channels are guaranteed through the offered platforms and services.

In order to introduce a new digital customer journey it is crucial to understand the business' and customers' needs first - introducing a journey just for the sake of introducing something digital often does not bear the anticipated results. After workshops and analysis the best solution fit is evaluated.

Exposing journeys to 3rd parties: All business logic is available through consumption of services provided by the underlying commerce platform.

2. Solution definition

The solution is built on microservices architecture where the particular microservices are pretty isolated. Of course there are still functional dependencies, however, these dependencies are isolated on a clearly defined integration layer between microservices.

There is a list of the modules:

- Accounts - management of Accounts including their credentials, assigned roles
- Address - physical addresses management
- Banner - banners and carousels management
- Behavior Tracking - user tracking system
- CRM Grouping - universal grouping and mass changes processing module
- CRM Obligations - generic obligations management (like contracts, agreements, commitments)
- CRM Party - Customer data and relationships management
- CRM Tickets - tickets management
- Chat - online chat module
- Discussion - user discussion module
- DMS - document management system
- Dynamic Content - conditioned (by rules) content
- Faq - frequently asked questions
- Forms - dynamic forms
- Location - generic locations management (e.g. POS)
- Customer Notifications - generic notifications module
- Order Management
- OTP - one-time password module
- Product catalogue
- Redirect Rules - GUI redirection rules
- Scheduling - generic jobs planning
- Templates - universal template engine

And there are also some supporting modules providing their functionality for other modules:

- ACL - Access Control List - authrorization module
- Codebook - global enumerations management
- DCR - Dynamic Context Rules
- Entity Meta Param - generic parameters associated to entities
- Localized texts
- Properties - key-value configuration
- Unique Key - universal unique key-values management

With regard to your question "*... For example, can we use only CRM, order management or product catalogue modules.*":

Product catalogue has no dependency on any other module except supporting modules (ACL, DCR) and thus can be used separately.
Both Order management as well as CRM Party module depend on Product catalogue module. Order management module may optionally use CRM Party module.

3. CRM and Data structures

3.1. Question

What business critical data can be managed by the solution?

Every microservice manages different set of business entities. The main entities/data managed by particular microservices are:

- **Product catalogue**: Product and product relationships configuration
- **Accounts**: internal and external accounts
- **CRM Party**: Customer structure and relationships, contacts, product instances
- **Order management**: Shopping cart, Order and Order lifecycle

3.2. Question

What is the level of configurability of your solution - which changes are defined as configuration and which require development?

It depends on a case-by-case basis, every module has different level of configurability. Generally we prefer configuration as much as possible, on the other hand there are always boundaries where further configuration is not possible and development is required.
Few examples:

- authorization module (ACL) is very flexible from configuration point of view - ACL Rules might be changed on the fly, new privileges and roles could be added without outage too. On the other hand, new Resource with complex identification logic (e.g. resource Customer) requires implementation.
- Order management module supports Processors steps configuration (it is possible to change processing steps of particular order fulfilment processor on the fly). On the other hand, new Agent requires implementation (Agent is a component - see 16.4 in Tech- and Functional reqs)
- Dynamic Context Rules (business rules evaluation engine) is highly flexible and fully configurable. Its usage depends on the context provided dynamically on the fly.
- CRM provides very flexible model to represent Customer as well as other CRM related entities. However, the core Customer structure is pre-defined. Only additional Party roles can be added.

Last but not least, every module supports configuration provided by centralized Properties component. Every property has default value which can be overridden by custom value (modifiable even during runtime).

See response 2.3 in Tech- and Functional reqs.

3.3. Question

Describe how the system ensures order capture and monitor each order end-to-end?

Order capture and processing is in the scope of the Order management module. Order could be captured either using GUI (using shopping scenarios) or by integration interface (ordering gateway API).
Order management module takes care about Order lifecycle from creating order, populating shopping cart, gather checkout data, confirm (validate and accept order) to its final fulfilment. Different order scenarios might be processed by different processors which are responsible to run the pre-configured order processing steps.
For further information refer to 16.4 in Tech- and Functional reqs.

3.4. Question

Can you describe how a new product configuration is propagated to the impacted modules of the solution or other parts of the ecosystem?

Product catalogue, whenever the product configuration is modified, emits the event to the event bus (by default we use Apache Kafka).
The interested modules subscribe on these events and handle them as needed. For example Order management and well as CRM Party module stores simplified view of products configuration for their particular use-cases, mainly because of performance reasons. When such Product catalogue event comes, the local product configuration is correspondigly updated.
Of course, any other module - even not delivered by Emeldi - may subscribe to these events too.
Note that this behavior isn't specific for Product catalogue module only. Other modules submit relevant events too.

4. Rating and billing

No rating / billing in the solution portfolio.

5. Integration

5.1. Question

Does the solution provide standardised API integration layer? Please, provide whitepaper.

Every microservice provides a well documented REST API, see linked document Emeldi-Commerce-Whitepaper.pdf. There is also attached an interface specification for the product catalogue: Emeldi_Commerce_Product_Catalogue_Interface_Specification.pdf

5.2. Question

Explain how you avoid system development to meet the business requirements? Such development imply changes to the solution components and the integration between these components.

Whenever possible we prefer configuration over system development. So if the affected component provides a way how to meet business requirement by configuration, it's used.

When the development is necessary, there are two major ways:

- least invasive is implementation using plugin; some modules are pluggable in the sense that they enable to register plugins which affects their behaviour (e.g. shopping cart stage handlers)
- inter module customization where the internal module implementation is changed

The separate topic is the integration impacts when system development takes place. General rule is that the API has to be backward compatible. So the possible changes must not break the services guaranteed by the API so far. In the rare cases when the backward compatibility cannot be maintained, new API version is introduced, and the previous version is kept untouched. Our solution is developed with API change resiliency in mind. It means the API clients follow the robustness principle: be conservative in what you send, be liberal in what you accept.

6. Ecosystem components

6.1. Question

Can you list the most often vendors that you are partnering with?

NTS Retail Austria, ZIRA Ltd. Bosnia Hercegovina, MIND CTI, Israel, Valomnia, France, CROSS Network Intelligence, Czech Republic,

7. Environment

7.1. Question

What kind of virtualization does the solution support (public and provide cloud)?

Emeldi Commerce® is built with virtualization in mind and it supports both public and private cloud. Even our internal development and continuous-integration platforms are fully virtualized.

7.2. Question

Does your solution require use of Oracle? -If yes, what components and what is the roadmap for Oracle alternatives?

We natively support two database vendors: Oracle and PostgreSQL from which you can pick. There is no other Oracle technology used within the solution.

7.3. Question

Can you provide more information about third party components dependency of the solution (like operating systems, databases, application servers, etc.)?

Our solution has very few requirements regarding the underlying technologies. The microservices are developed in java running on JVM. From that point of view it can run on any platform where the JVM is implemented. Our primary environments are unix-like systems, Linux can be recommended.

As an application server any common java-based application server may be used (JBoss, Tomcat, Jetty), there is no dependency on any particular one.

Going further, we deliver and operate our components as Docker containers running in Docker Swarm. Using this setup, the solution is even more independent of specific technologies.

Databases were answered in previous question. Oracle and PostgreSQL is supported.

Generally we use open-source products as much as possible.

Used technology stack:

- Application framework: Spring Framework, Spring Cloud, Spring Boot
- Application framework (front-end): Spring MVC, Spring Security
- REST API documentation: Swagger
- Java bean mapper: Selma
- Distributed in-memory cache: Hazelcast
- Messaging: Apache Kafka

- Service registry: Eureka
- Web browser client: AngularJS, TypeScript

7.4. Question

Would you provide more information about the support options and main drivers for the support?

We recommend DevOps approach where particular developers (or developer mini-teams) are simultaneously in-charge of the operation of the module(s) they are responsible for.

In other words in this setup there are several self-service & autonomous teams responsible to deliver, test and operate the service in production. Thanks to that there is also no single administrator. This setup enables frequent and incremental changes.

Support is provided with several tools to do their jobs. Some tools are specific to particular module (e.g. Order management module provides tool to cancel or retry the Order).

We provide also extensive centralized application logging which we understand as crucial for operation, monitoring and problem solving. However, such logging is not so straighforward in distributed and virtualized environment. Therefore we use data collectors, indexing and visualisation tools for that (involves fluentd, Elasticsearch, Kibana or Graylog). The logs also contains identifiers like Application instance and Request identifier to enable aggregation of relevant logs together.

The application entities are also provided with so-called audit tables where all changes made on entity are recorded (every insert/update/delete operation with all modified data). Audit record contains also who, when and in which transaction made the operation.

7.5. Question

Would you provide the SLAs for the solution?

Emeldi will provide L3 support for delivered solution based on the SLAs. The SLA parameters, scope of supported solution and support level will be contractually agreed between Emeldi and Telenor BG.

7.6. Question

Can you explain how a new tenant can use the solution, featuring data model, software components, hardware and other important elements?

Emeldi Commerce® can be deployed in Cloud, however it is not a multi-tenant application. We considered multitenancy architecture in the past, but from our experience in Telecom sector we do not see much demand for multitenancy, particularly in Telco online and CRM solution domains. Both domains are considered mission critical and require frequent changes, flexibility and in general fast time to market to implement changes. With multi-

tenant software Telco operators frequently list concerns with privacy, security, availability, performance isolation and increased development and maintenance complexity.

7.7. Question

Can you describe the high availability, disaster recovery and geo redundancy capabilities and limitations of the solution?

Emeldi Commerce® is built as a fully distributed application with all pros and cons the distributed environment brings. The non-functional characteristics like high-availability, scalability and performance are the key benefits of our distributed architecture. The solution is fully horizontally scalable on application layer. We use Docker Swarm to keep as many instances of particular component as needed. Vertical scaling is of course possible too, but does not work well with hardware abstraction and isolation technologies.

Horizontal scalability on database layer is more difficult and depends on capabilities of particular chosen database vendor. Generally both supported platforms (PostgreSQL and Oracle) supports very well "hot-standby" mode. On the other hand, thanks to the microservices architecture nature, each component may (but not forced to) be operated on dedicated database instance. It can be used with advantage to increase performance and scalability of the solution.

7.8. Question

Can you provide performance figures (accomplished with sample hardware environment) for example but not limited to processed records, calls, invoices, concurrent users, sessions or transactions and other major figures representing overall system performance?

As an example we might provide statistics from O2 CZ OneCRM project. There are 12 application servers for presentation layer and 8 application servers for backend layer. These servers host all Emeldi Commerce® modules.

The platform was capable to handle this amount of orders during business hours within one day (October 10):

Hour	Orders created	Orders finished	WS out	WS in	JMS out	JMS in
06-07	351	338	4529	27348	2238	8025
07-08	1062	930	9159	37018	4616	13609
08-09	1916	2490	16508	92290	11372	28796
09-10	2292	1460	20013	84429	5217	23141
10-11	2030	407	21349	67198	1273	16623

11-12	2052	486	17613	73600	1337	17636
12-13	3628	4396	22136	93400	18848	32044
13-14	2785	1452	21288	107566	6378	40337
14-15	3084	3701	22445	121623	14064	39547
15-16	2950	2286	21352	89189	10109	32519
16-17	2861	2146	21091	95793	9468	32527
17-18	2092	1647	15550	16049	7769	14697
18-19	1636	1217	13777	10268	5910	11849
19-20	1411	1129	12633	8448	5643	10592

First columns are hours, second column is amount of orders created within an hour, and third column is amount of orders finished and fulfilled within the same hour. Forth column is amount of synchronous outgoing webservice calls, fifth column is amount of incoming dispatched synchronous webservice requests. Sixth column is amount of outgoing JMS messages and seventh column is amount of processed incoming JMS messages.

This performance were achieved with a relatively low CPU load on the servers (approximately 30%). During performance tests the platform were capable to handle 10.000 orders per hour.

7.9. Question

Can you provide more information regarding the scalability of the solution depending on the load on different layers of the solution like front end, integration or backend processes it should be able to be scaled to meet different loads (faster billing process, ability to handle more parallel front end users/sessions or more requests through integration and order management)?

Every module is build and deployed as a separate microservice which is scaled independently. It's perfectly possible to run different amount of instances based on the load the particular microservice has to handle. Thanks to the virtualization and dynamic service discovery the amount of instances could be easily changed during runtime.