# II. Архитектура на облачна система за управление на поръчките от клиенти

В този раздел са представени основни елементи, които да положат фундамент на софтуерно решение. Първа подточка ще разгледа общи принципи на софтуерната архитектура. След това втора и трета ще дадат детайлна характеристика на концептуалните и функционални модели на подсистемите.

## 2.1 Същност, цел и обхват на софтуерната архитектура

В общ смисъл, софтуерната архитектура е структурирано решение, което може да оптимизира общи атрибути на качеството като: висока производителност, сигурност, контрол, управляемост, мащабируемост, достъпност (Ali Babar et al., 2009). Думата „**архитектура**“ често се използва в контекста на нещо от високо ниво, което е отделено от детайлите на по-ниско ниво. За разлика от него, „**дизайнът**“ предполага структури и решения на по-ниско ниво. Основната цел на архитектурата е да поддържа жизнения цикъл на системата. Добра архитектура прави системата лесна за разбиране, разработване, поддръжка и внедряване. Kрайната цел е да се минимизират разходите за целия живот на системата и да се увеличи максимално продуктивността на програмиста. (Martin et al., 2017)

Кларк и Уокър (2001) въвеждат концепцията за композиционни модели като начин за разработка. Структурни елементи и интерфейси, съставящи системата са:

* Обекти – градивни елементи от ниско ниво;
* Комуникационните канали в архитектурата;
* Картографиране структурата на данните от високо ниво;
* Архитектурни стилове (Model-View-Controller, Ориентираната към микроуслуги и други), които ръководят композицията.

В архитектурата на сградите обосновката експлицира основната философска естетика, която мотивира архитекта. В софтуерната архитектура, обосновката вместо това обяснява удовлетворението на системните ограничения. Тези ограничения се определят от съображения, вариращи от основни функционални аспекти до различни нефункционални аспекти като икономика, производителност и надеждност (DE Perry et al., · 1992). По тази причина дизайнът трябва да обхваща икономически, технологични ограничения и естетически проблеми.

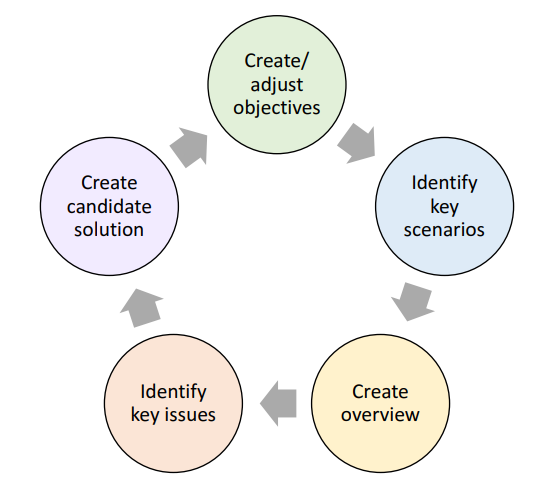
Крайната цел е да се структурира документ от високо ниво, който да не навлиза в подробности за изпълнението. Сложността на операциите трябва да е сведена до минимум, като да обръща внимание на всички изисквания. Приложенията трябва да се съвместими с всички случаи на употреба и бизнес сценарии. Критичните точки трябва да се идентифицират и проучат. Добра практика при проектиране на софтуерна архитектура е да бъде ориентирана към модулност, приемайки лесно новите промени.

***Таблица 2.1****: Принципи на проектиране.*

|  |  |  |
| --- | --- | --- |
| име на български | име на английски | Описание (тук ще има доста цитати |
| Разделяне на грижите | Separation of Concerns | всеки обект и модул трябва да бъде в своя собствена грижа и контекст |
| Капсулиране | Encapsulation |  |
| Инверсия на зависимостта | Dependency Inversion |  |
| Изрични компоненти | Explicit Components |  |
| Единична отговорност | Single Responsibility |  |
| Не се повтаряйте | Don’t Repeat Yourself |  |
| Устойчивост и невежество относно инфраструктурата | Presentation Ignorance |  |
| Ограничени контексти | Bounded Contexts |  |

+ TODO: Препоръки при проектиране

Процесът за проектиране на архитектурa:



* + Създаване или коригиране на цели (определяне обхвата): в тaза част от процеса се включват целите и изискванията на високо ниво, технологии, времето за прекарване, идентифициране на екипа за разработка (ДИРЕКТОР, Разработчици, Функционален анализатор), идентифициране на технически ограничения, (ограничения за използване и внедряване), анализ на технологиите за разработка, идентифициране на потребителите, използващи системата едновременно, трафика, времето за отговор и др.
  + Идентифициране на ключови сценарии: вкючва съществени, но неизвестни към момента, рискове. Пример за това биха били протомотиране на кодове за отстъпка. Също така важни случаи на употреба, критични за бизнеса, които са част от основна бизнес област като например интеграция на плащания. Свързва се с UML диаграми на случаи на използване, които ше бъдат разгледани в точка 2.1.1;
  + Общ преглед – определяне на вида на приложението(ята), идентифициране на ограниченията по внедряване и разгръщане, архитектурните стилове, определяне на технологии, библиотеки и инструменти за разработка;
  + Идентифициране на ключовите проблеми - време на изпълнение, дизайн, производителност, стабилност, управляемост, конфигурируемост и други системни проблеми като удостоверяване и оторизация, кеширане, комуникация, управление на конфигурацията, регистриране и управление на изключения, валидиране.
  + Етап но финалния кандидат за разгръщане на нова версия – ЙЙ: свързан е със всичко напред: (създаване на диаграми).

Унифицираният език за моделиране (Unified Modeling Language) e графичен език за визуализиране, специфициране, конструиране и документиране на елементите на една софтуерна система.

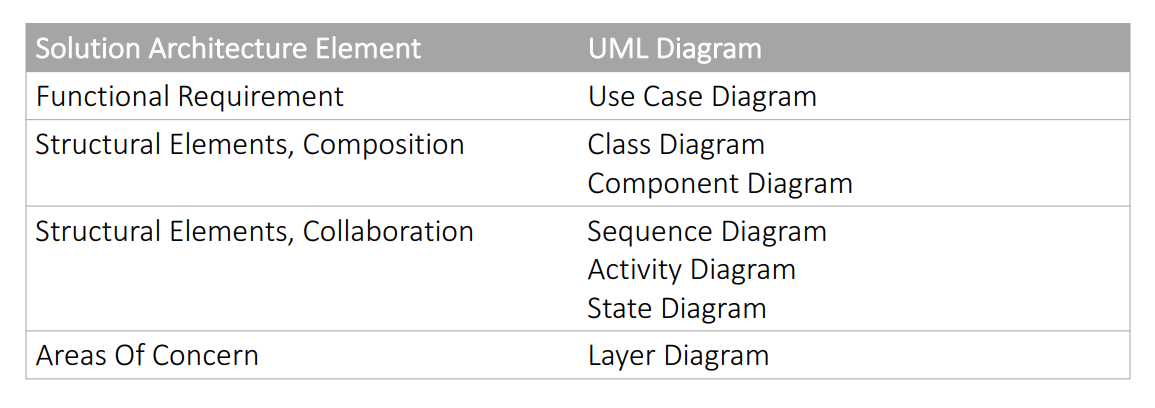
Основни атрибути на UML:

* + Визуално – лесно се вижда представянето на архитектурата;
  + Абстрактно – стои далеч от детайлите на изпълнението;
  + Описателен – показва пълното представяне;
  + Стандарт – UML е отвърден световен стандарт;
  + Поддържа генериране на код – определени секции могат да бъдат конвертирани в код;

Типове UML модели:

* + Бизнес модел – нетехнически, детайлен, приемайки системата като черна кутия.
  + ИТ модел – разделя се на статични (структурни), които изобразяват как се съчетават различните елементи, и динамични (Поведенчески), които се фокусират върху взаимоотношенията.

UML диаграми в архитектурата:



Стратегии за проектиране на UML:

* + UML като скица –предназначен към общи насоки;
  + UML като план - много подробен, може да се пишете код въз основа на диаграмите;
  + UML като валидиране – валидиране на изпълнението спрямо диаграматал;

## 2.2 Концептуален модел на системата

Концептуалните модели са абстрактни представяния за това как трябва да протича изпълнението на задачите. Хората използват концептуалните модели подсъзнателно и интуитивно като начин за систематизиране на процесите (Molina et al., 2000).

+ тодо : https://www.linkedin.com/pulse/conceptual-models-business-dilan-moragolle

### 2.2.1. Структурни диаграми

Структурните диаграми помагат за дефиниране цялостната структура системата, подобно на плана, който определя как изглежда една къща. Структурните диаграми моделират как ще изглежда нашата система в архитектурно отношение. Те ни помагат да дефинираме „речника“ на системата, гарантират съгласуваност от заинтересовани страни в проекта. Идентифицират различни връзки между различните частти на системата.

Структурните диаграми на UML изобразяват елементите на система, които са независими от времето и които предават концепциите на системата и как те се свързват помежду си. Елементите в тези диаграми приличат на съществителните в естествения език и връзките, които ги свързват, винаги показват структурни или семантични връзки. Например, структурна диаграма на система за резервация на превозно средство може да съдържа елементи като кола, резервация, шофьорска книжка и кредитна карта и да съдържа съединители, свързващи тези елементи. Опитните моделисти също ще покажат връзките с поведенческите елементи на тези диаграми.

https://www.sciencedirect.com/topics/computer-science/structure-diagram

#### 2.2.1.1. Диаграма на класовете UML

Kласове, методи и полета, асоциации, обобщения и кардиналност

#### 2.2.1.2. Диаграмата на компонентите UML

Модулни градивни блокове, внедрени и необходими интерфейси

### 2.2.2. Поведенчески диаграми

#### 2.2.2.1. Диаграмата на последователността UML

Последователност от повиквания: извикващ клас, извикан метод и тип на връщаните данни.

#### 2.2.2.2. Диаграмата на състоянията UML

Състояния или дейности, преходи

#### 2.2.2.3. Диаграма на дейността UML

работен процес, едновременни действия, декларации

## 2.3 Функционална структура на системата

### 2.3.1. Подсистема за обслужване на потребителите

В различни източници се използва много различна терминология: удостоверител за самоличност, услуга предоставяща токени за сигурност (Security Token), сървър за оторизация и много други, но сами по себе си те са еднакви, имат сходни изисквания:

* + Защитават ресурсите;
  + Удостоверяват потребителите;
  + Осигуряват управление на сесии;

Важни функции на сървъра за самоличност са:

* + услуга за удостоверяване, която да работи в централизиран процес;
  + Единично влизане/излизане за множество приложения;
  + Покрива индустриалните стандарти OpenID Connect и OAuth 2.0;
  + Шлюз към Google, Facebook и др;

Сложните уеб системи имат различни типове комуникацииони модели (между клиент и сървър, между вътрешен и външен сървър, и др.), като също така всеки слой трябва да защитава ресурсите на съответното ниво чрез удостоверяване и оторизация. По нози начин изнасянето на функциите за сигурност към отделна услуга предотвратява дублирането. В случая, сървъра за самоличност се явява като междинен софтуер, който добавя съвместими със спецификациите OpenID Connect и OAuth 2.0 крайни точки.

Токените са сигурност се използват, когато искаме да получим достъп до защитен ресурс. Удостоверяването чрез токени е механизъм без състояние, тъй като никаква информация за потребителя не се съхранява в паметта на сървъра или базaтa от данни (Gichuhi , 2021). (за разлика от бисквитките). Потокът: Потребителят предоставя идентификационни данни на сървър за оторизация и получава токен, след това токенът може да бъде изпратен до API за достъп до ресурси. API трябва да потвърди токена, преди да върне каквито и да е данни. Стандарт (RFC 7519) за уеб приложенията е JSON уеб токен (JWT). Той се състои от три части:

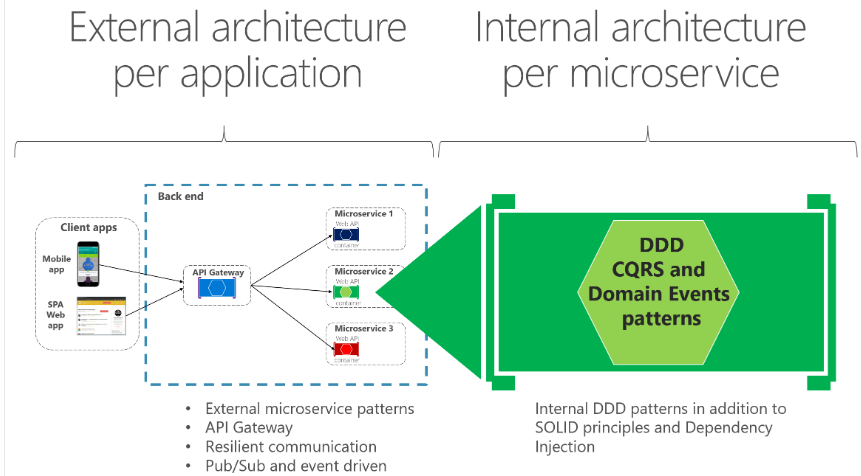
* + Заглавна - JSON обект, кодиран във формат base64. Съдържа информация за типа на токена и алгоритъма за криптиране;
  + Полезен товар - съдържа информация за текущия потребител (потребителско име, роля и др). Тук не трябва да се включват чувствителни данни, защотп лесно се могат да бъдат декодирани със публични сайтове като jwt.io;
  + Подпис – Използва се от сървъра, за да провери дали токенът е валиден. Той се генерира чрез комбиниране на двете части (заглавна и полезен товар) заедно. Базира се на таен ключ, който само сървърът за удостоверяване знае. По този начин злонамерен потребител не може да фалшифицира валиден токен;

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1laWQiOiI0YTY2ZWNmNC1iZDdjLTQ3ODQtYmViOS1jZGM0MzQzZGY3MWYiLCJ1bmlxdWVfbmFtZSI6Im15QG15LmNvbSIsIm5iZiI6MTU5NjEzMzk3OCwiZXhwIjoxNTk2NzM4Nzc4LCJpYXQiOjE1OTYxMzM5Nzh9.W7k3UXA1g3TKxt-hR9a-mgCAcCsKjEyGTxBv5Dt79y8

***Таблица 2.1****: Пример за токен.*

### 2.3.2. Подсистема за управление на поръчките

Този подраздел се фокусира върху услуга, която обслужва данните за поръчки. Това е сложна подсистема с непрекъснато променящи се бизнес правила. Архитектурните модели, използвани в този подраздел, се основават на подходи, управлявани от домейн (DDD) и разделяне на отговорността за команди и заявки (CQRS).



***Фигура 2.2****: Архитектура на външна среда спрямо модели на вътрешна архитектура. Източник:* [*https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/*](https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/)

CQRS е архитектурен модел, който разделя моделите за четене и запис на данни. Свързаният термин Command Query Separation (CQS) първоначално е дефиниран от Bertrand Meyer в книгата му Object-Oriented Software Construction. Основната идея в разделението на операциите на две категории:

* + Заявки - връщат резултат и не променят състоянието на системата. Нямат странични ефекти;
  + Команди - променят състоянието на системата;

Разделянето на отговорността за командване и запитване (CQRS) бе въведено от Грег Йънг и силно насърчавано от Уди Дахан. Базиран е на принципа на CQS. Дизайнът на услугата за поръчки в приложението се основава на тези принципи.

Управляван от домейн дизайн (DDD) препоръчва моделиране въз основа на реалността, реалните бизнес сценарии. В контекста на изграждането, DDD говори за проблеми като домейни. Той описва независими проблемни области като ограничени контексти (Bounded Context) и подчертава общ език за обсъждане на тези проблеми. Също така предлага много технически концепции, като обект с богати модулярност (а не анемичен), стойностни обекти, агрегати и други. Този раздел представя дизайна и изпълнението на тези вътрешни модели.

Структурата на папките на приложениео е добре оформена, по следния функционален, управляван от домейн дизайн:

├───**Orders**

│ ├───**src** – индикира път към основния код

│ │ ├─── **Orders.Web** - логика за представяне

│ │ ├─── **Orders.Business** – бизнес логиката на приложението

│ │ ├─── **Orders.Core** – експонирани за клиенти модели

│ │ ├─── **Orders.Domain** – вътрешни модели на приложението

│ │ └─── **Orders.Persistence** – логика за достъпа до данните

│ └───tests – индикира път към тестовете на кода

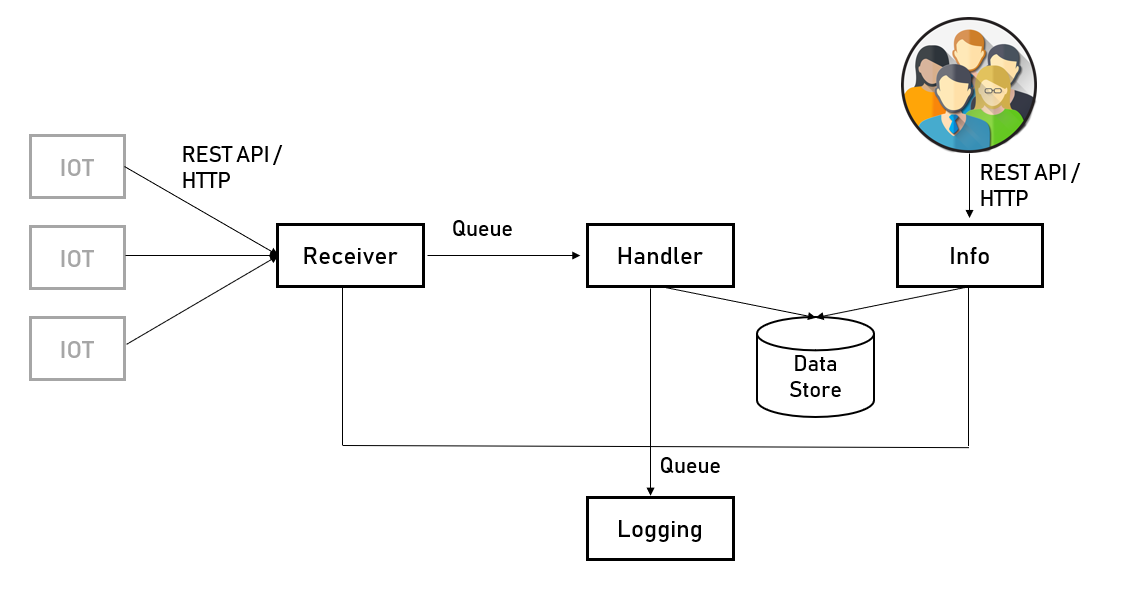
│ └─── **Orders.Tests** – пълен набор от автоматизирани (интеграционни) тестове

### 2.3.3. Подсистема за управление на доставките

### Подсистемата отчита състоянието на устройствата в колите за доставка. IoT (Интернет на нещата) представляват малки, винаги свързани устройства, използвани всеки ден, като GPS. Клиентите използват и управляват в реално време. Подсистемата събира информация за състоянието от регистрирани IoT устройства и форматира данните в структуриран вид, което позволява на клиента да знае точното местоположение на доставка. Освен това клиентът може да изпълни някои предварително зададени заявки за достъп до повече информация за устройствата.

### Важно е да се отбележи, че за първа фаза на системата, описана в дисертацията, клиентът не добавя или актуализира никакви данни. Информацията за състоянието се получава директно от устройствата. Данните просто се представят.

Преглед oт високо ниво на архитектурата:



Както е показано на диаграмата, приложението се състои от четири отделни, независими, слабо свързани услуги. Всяка има свои собствени задачи и комуникира с другите, използвайки стандартни протоколи.

Всички те са изградени като услуги без състояние, което означава, че не се губят данни от внезапното спиране. Единствените места за данни в приложението са опашката и хранилището, като и двете сериализират данните на диска, като по този начин го предпазват от случаи на изключване.

Във връзка с платформа за разработка, тази архитектура ще спомогне за създаването на модерна, здрава, лесна за поддръжка и надеждна система, която може да служи успешно на компанията за години напред.

Архитектурата се състои от следните услуги:

* + Приемник – получава актуализации на от различните устройства и ги добавя към опашка за по-нататъшна обработка. Приемникът поставя силен акцент върху производителността и основната му задача е да гарантира, че актуализацията е получена и съхранена. Той не предприема никакви действия по актуализацията, което е ролята на услугата Handler;
  + Handler – потвърждава и анализира актуализацията. Програма, която обработва актуализациите от опашката (където ги е поставил получателят), проверява съдържанието им (например да провери текущият статус дали е в правилния диапазон) и ги конвертира в унифициран формат, така че да няма значение от на кое устройство е получен статусът. След обработка на съобщението то ще бъде съхранено в хранилище за данни;
  + Услуга за информационно обслужване – Разкрива API, което приема заявки от крайни потребители и връща необходимите данни за доставката;

## 2.4 Характеристика на комуникационните модели - (реферат по дисциплината Интернет технологии и комуникации)

### 2.4.1. Синхронна комуникация

### 2.4.2. Асинхронна комуникация

### 2.4.3. Достъп до подсистемите

# Литература:

1. <https://www.sciencedirect.com/topics/computer-science/structured-solution> (Anil Kumar Thurimella, T. Maruthi Padmaja, in Economics-Driven Software Architecture, 2014)
2. Clean code
3. Clean architecture
4. Foundations for the Study of Software Architecture Dewayne E. Perry AT&T Bell Laboratories 600 Mountain Avenue Murray Hill, New Jersey 07974 dep@research.att.com Alexander L. Wolf
5. JG Molina, MJ Ortín, B Moros, J Nicolás. (2000) *Towards use case and conceptual models through business modeling*. Conceptual Modeling, 2000 - Springer.
6. <https://www.section.io/engineering-education/cookie-vs-token-authentication/>
7. <https://martinfowler.com/bliki/CommandQuerySeparation.html>
8. https://martinfowler.com/bliki/CQRS.html