## 1.2. Възможности за дигитализация на процесите по управление чрез прилагане на облачни технологии

В последните години облачните технологии се превърнаха във водеща тенденция в софтуерната индустрия. Те предоставят нов начин за изграждане на големи и сложни системи, като по този начин използват пълноценно съвременните практики за разработка на високо-качествен софтуер и налична инфраструктура. Това променя начина на проектиране, интегриране и внедряване на системите. Облачно базираните решения са проектирани да приемат бързо промените, да обслужват голям мащаб от хора и да бъдат устойчиви на всякакъв вид натоварване или хакерски атаки (Vettor, 2022).

Организацията Cloud Native Computing Foundation предлага следното определение: "*Технологиите, базирани на облак, дават възможност на организациите да създават и изпълняват приложения в модерни, динамични среди като публични, частни и хибридни облаци, чрез мрежи от услуги и микроуслуги. Качества на системите са устойчивост, висока наличност и достъпност, мащабируемост и управляемост, които са от критично значение за много от бизнес единиците. Автоматизацията на тези процеси позволява на инженерите да правят промени, с голямо въздействие, но с минимални усилия."*

Приложенията стават все по-сложни, като изискванията, от страна на потребителите, стават все повече и повече, главно насочени към бърза реакция и иновативни функции. Проблеми с производителността или повтарящи се грешки вече не са приемливи.

Предимствата на облачните системи поставят бизнеса една стъпка пред конкурентите. Бизнес системите се развиват от способностите на бизнеса да бъдат инструменти за стратегическа трансформация, която ускорява растежа на компанията. Облачно базираните системи се свързват главно с бързина (Smith, 2022). Незабавното пускане на иновативните идеи на пазара е важна тема за всички модерни  компании, например следните компании са приложили успешно тези техники:

• Netflix има над 600 услуги в производствена среда. Стотици пъти на ден се изпълняват нови внедрявания и разгръщания на съствуващи;

• Uber има над 1000 услуги в производствена среда. Обновяват се няколко хиляди пъти всяка седмица;

Както е видно, бизнесът на тези две компании се базира на системи, които се състоят от стотици независими микроуслуги. Този архитектурен стил им позволява бързо да реагират на пазарните условия като постоянно актуализират малки, но важни области. Скоростта на облачния носител се дължат на редица фактори, като на първо място е инфраструктурата на изчислителните ресурси.

На фигура 1.10. са показани пет основополагащи стълба, осигуряващи основата за базирани в облак системи.



Фиг. 1.10. Фундаментни стълбове на облачните системи (Smith, 2022).

Проектирани да процъфтяват в динамична, виртуализирана облачна среда, облачните системи използват широко „платформата като услуга“ (PaaS) изчислителна инфраструктура и управлявани услуги. Те третират основната инфраструктура като за еднократна употреба - осигуряват се за минути и се преоразмеряват, мащабират или унищожават при поискване - чрез автоматизация.

Нека разгледаме DevOps концепция наречена: „Pets vs. Cattle“ (Menchaca, 2018). В традиционния център за данни сървърите се третират като домашни любимци (pets), като всеки един от тях представлява физическа машина, която трябва да бъде поддържана. Мащабирането се случва като се добавят ресурси към нея. Възникването на проблем в сървър, рефлектира върху всички потребители. Моделът на услугата Cattle е по-различен. При него всеки ресурс се предоставя като виртуална машина или контейнер. Те са идентични и присвояват системни идентификатори като Service-01, Service-02 и т.н. Мащабирането се случва като се създадат нови екземпляри. Ако един от тях стане недостъпен, друг поема неговата роля. Облачните системи поддържат този модел. Те продължават да работят, независимо от машините.

Дванадесетфакторното приложение (Wiggins, A., 2017), представено в таблица 1.2, е известна методология за конструиране на облачно базирани приложения. Изготвена от разработчици в Heroku, компания, преглагаща, платформа като услуга, описва набор от принципи и практики, които разработчиците следват, за да създават приложения, оптимизирани за модерни облачни среди. Много практици смятат Twelve-Factor за солидна основа за изграждане на облачни приложения, защото е приложим за всяко уеб-базирано приложение. Системите, изградени на този принцип, могат да се внедряват и мащабират бързо, както и да добавят нови или да променят съствуващи функции, за да реагират бързо на пазарните промени.

Таблица 1.2.  
 Описание на методологията на дванадесетте фактора  
(Wiggins, 2017)

| **Factor** | **Explanation** |
| --- | --- |
| 1 - Code Base | A single code base for each microservice, stored in its own repository. Tracked with version control, it can deploy to multiple environments (QA, Staging, Production). |
| 2 - Dependencies | Each microservice isolates and packages its own dependencies, embracing changes without impacting the entire system. |
| 3 - Configurations | Configuration information is moved out of the microservice and externalized through a configuration management tool outside of the code. The same deployment can propagate across environments with the correct configuration applied. |
| 4 - Backing Services | Ancillary resources (data stores, caches, message brokers) should be exposed via an addressable URL. Doing so decouples the resource from the application, enabling it to be interchangeable. |
| 5 - Build, Release, Run | Each release must enforce a strict separation across the build, release, and run stages. Each should be tagged with a unique ID and support the ability to roll back. Modern CI/CD systems help fulfill this principle. |
| 6 - Processes | Each microservice should execute in its own process, isolated from other running services. Externalize required state to a backing service such as a distributed cache or data store. |
| 7 - Port Binding | Each microservice should be self-contained with its interfaces and functionality exposed on its own port. Doing so provides isolation from other microservices. |
| 8 - Concurrency | When capacity needs to increase, scale out services horizontally across multiple identical processes (copies) as opposed to scaling-up a single large instance on the most powerful machine available. Develop the application to be concurrent making scaling out in cloud environments seamless. |
| 9 - Disposability | Service instances should be disposable. Favor fast startup to increase scalability opportunities and graceful shutdowns to leave the system in a correct state. Docker containers along with an orchestrator inherently satisfy this requirement. |
| 10 - Dev/Prod Parity | Keep environments across the application lifecycle as similar as possible, avoiding costly shortcuts. Here, the adoption of containers can greatly contribute by promoting the same execution environment. |
| 11 - Logging | Treat logs generated by microservices as event streams. Process them with an event aggregator. Propagate log data to data-mining/log management tools like Azure Monitor or Splunk and eventually to long-term archival. |
| 12 - Admin Processes | Run administrative/management tasks, such as data cleanup or computing analytics, as one-off processes. Use independent tools to invoke these tasks from the production environment, but separately from the application. |

В книгата Beyond the Twelve-Factor App авторът Кевин Хофман описва подробно всеки от оригиналните 12 фактора, като добавя три допълнителни, които отразяват модерен дизайн на облачни приложения.

Таблица 1.3.  
 Организационни структури в САП  
(Хофман, 2022)

| **New Factor** | **Explanation** |
| --- | --- |
| 13 - API First | Make everything a service. Assume your code will be consumed by a front-end client, gateway, or another service. |
| 14 - Telemetry | On a workstation, you have deep visibility into your application and its behavior. In the cloud, you don't. Make sure your design includes the collection of monitoring, domain-specific, and health/system data. |
| 15 - Authentication/ Authorization | Implement identity from the start. Consider [RBAC (role-based access control)](https://learn.microsoft.com/en-us/azure/role-based-access-control/overview) features available in public clouds. |

Проектирането и внедряването на облачно базирани работни натоварвания може да бъде предизвикателство. Microsoft Well-Architected Framework (Stanford D. et al, 2022) предоставя набор от ръководни принципи, които се използват за подобряване качеството на работното натоварване. Следната таблица представя пет важни стълба на добра архитектурата.

Таблица 1.4.  
Стандартни за добри практики на облачаната индустрията  
(адаптирано от автора по Croll, 2022)

| **Tenets** | **Description** |
| --- | --- |
| [Cost management](https://learn.microsoft.com/en-us/azure/architecture/framework/#cost-optimization) | Focus on generating incremental value early. Apply Build-Measure-Learn principles to accelerate time to market while avoiding capital-intensive solutions. Using a pay-as-you-go strategy, invest as you scale out, rather than delivering a large investment up front. |
| [Operational excellence](https://learn.microsoft.com/en-us/azure/architecture/framework/#operational-excellence) | Automate the environment and operations to increase speed and reduce human error. Roll problem updates back or forward quickly. Implement monitoring and diagnostics from the start. |
| [Performance efficiency](https://learn.microsoft.com/en-us/azure/architecture/framework/#performance-efficiency) | Efficiently meet demands placed on your workloads. Favor horizontal scaling (scaling out) and design it into your systems. Continually conduct performance and load testing to identify potential bottlenecks. |
| [Reliability](https://learn.microsoft.com/en-us/azure/architecture/framework/#reliability) | Build workloads that are both resilient and available. Resiliency enables workloads to recover from failures and continue functioning. Availability ensures users access to your workload at all times. Design applications to expect failures and recover from them. |
| [Security](https://learn.microsoft.com/en-us/azure/architecture/framework/#security) | Implement security across the entire lifecycle of an application, from design and implementation to deployment and operations. Pay close attention to identity management, infrastructure access, application security, and data sovereignty and encryption. |

Облачните системи поддържат ориентирания към микроуслуги архитектурен стил за конструиране на системи. Това е подход за изграждане на сървърно приложение като набор от малки, но високо-качествени подуслуги. Съотвено, клиентите, на сървърните услуги, могат да бъдат отделни приложения, които да се поддържат и управляват самостоятелно. Всяка услуга работи в собствен процес и комуникира с други процеси, използвайки различен тип и вид протоколи като: HTTP/HTTPS, WebSockets, AMQP и мн други. Всеки микросървис притежава специфична бизнес способност. Предимства на това архитектурен стил са:

• Всяка микроуслуга може да бъде проектирана, разработена и внедрена независимо една от друга, което осигурява възможно за независима работа по отделни области на приложението;

• Работата може да бъде дистрибутирана между отделни екипи;

• Проблемите са по-изолирани;

• Позволява използването на различни технологии;

Микроуслугите насърчават фактор #6 от принципите на дванадесетфакторното приложение.

………………

Всяка микроуслуга притежава своя собствена логика и данни, в рамките на автономен жизнен цикъл. Концептуалните модели се различават между подсистемите или микроуслугите. Този принцип е заложен в дизайнът, управляван от домейн (DDD), където всяка услуга притежава свой модел на домейн (данни плюс логика и поведение).

Традиционният (монолитен) подход, използван в много приложения, притежава единична централизирана база данни (или няколко бази), която често е нормализирана SQL база, използвана за цялото приложение и всички негови вътрешни подсистеми. Този подходът изглежда по-прост, позволява повторно използване на обекти. В крайна сметка се стига до огромни таблици, които обслужват много различни подсистеми, включващи атрибути и колони, които в повечето случаи не са необходими. Монолитните приложения: ACID транзакции и SQL език, като и двете работят във всички таблици и данни, свързани с приложението. Това допринася за сравнително лесно комбиниране данни от множество таблици.

Достъпът до данни, става много по-сложен в архитектурата на микроуслуги. Дори когато използвате ACID транзакции в рамките на микроуслуга или ограничен контекст, е изключително важно да се има предвид, че данните, притежавани от всяка микроуслуга, са частни за тази и трябва да бъдат достъпвани синхронно, чрез нейните API крайни точки (REST, gRPC, SOAP и т.н.) или асинхронно чрез съобния (AMQP). Капсулирането на данните гарантира, че микроуслугите са слабо свързани и могат да се развиват независимо една от друга. Ако множество услуги имат достъп до едни и същи бази данни, актуализациите на схемата изискват координация. Това би нарушило автономността на жизнения цикъл. Когато един бизнес процес обхваща множество микроуслуги трябва се да използва т.н. „евентуална последователност“. Това е много по-трудно за изпълнение от обикновените SQL съединения. Различните микроуслуги често използват различни видове бази данни. Съвременните приложения съхраняват и обработват различни видове данни. За някои случаи на употреба NoSQL база данни като Azure CosmosDB или MongoDB може да има по-удобен модел, както и да предлага по-добра производителност и мащабируемост от SQL база данни. Микроуслугите често използват смесица от SQL и NoSQL бази данни, което се нарича подход на „полиглотна устойчивост“ (Polyglot Persistence).

Концепцията за микроуслуга произлиза от модела на ограничен контекст (Bounded Context) в управлявания от домейн дизайн (DDD). DDD се занимава с големи модели, като ги разделя на множество BC и определя техните граници. Всеки BC има собствен модел и база данни. По същия начин всяка микроуслуга притежава свързаните с нея данни. В допълнение, BC притежават така нареченият повсеместен език, който помага на комуникацията между разработчици на софтуер и експерти в бизнеса. Ограничен контекст може да бъде отделна услуга или е просто логическа граница.

……..

Предизвикателства и решения за управление на разпределени данни

Определянето на границите на микросервиз основно предизвикателство, което съсредоточава върху логическите модели на приложението и свързаните с него данни. Всеки контекст може да има различни бизнес термини. Термините и обектите, може да звучат подобно, но понякога имат различни цели в различни контексти. Например, даден потребител може да бъде посочен като потребител на системата, като клиент в контекста на ERP и т.н. Начинът, по който се идентифицират границите между множество контексти на приложения с различен домейн, е сходен с начина за ограничаване границите за всяка бизнес микроуслуга. Зависимостите между микроуслугите трябва да е сведена до минимум.

Таблица 1.5.  
Принципи за проектиране на облачни системи  
(адаптирано от автора по Croll, 2022)

|  |  |  |
| --- | --- | --- |
| име на български | име на английски | Описание (тук 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 |  |

В книгата Cloud Native Patterns, авторът Корнелия Дейвис (2019) отбелязва, че „Контейнерите са чудесен инструмент за облачния софтуер“. Фондацията Cloud Native Computing поставя контейнеризацията на микроуслуги като първа стъпка в Cloud-Native Trail Map - насоки за предприятия, които започват поддръжка в облака.

За изграждане, доставка и изпълнение на системи, изградени както като монолитни приложения, така и като ориентирани към услуги, се препоръчва използването на контейнеризирани технологии. Контейнеризацията е подход, в сферата на разработката на софтуер, при който кодът на приложение, всички негови зависимости и конфигурации са пакетирани в двоичен файл, наречен изображение. Изображенията са „шаблони“ само за четене и се съхраняват в регистър, който работи като хранили или библиотека за изображения. Изображението се трансформира в работещ екземпляр на контейнер, който може да се стартира, спира, премества и изтрива. Създават се контейнери за различните части от приложението: уеб услуга, база данни, кеширане и др. Точно както транспортните контейнери позволяват транспортирането на стоки, независимо от товарите вътре, софтуерните контейнери се възприемат като стандартна единица за внедряване на софтуер, която може да съдържа различен код и зависимости. Контейнеризирането на софтуера дава възможност на разработчиците и ИТ специалистите автоматично да подновяват новите промени в различни среди. Контейнерите също така изолират приложенията едно от друго в споделена операционна система. Приложения се изпълняват върху хостът на контейнерите. От гледна точка на приложението, инстанцирането на изображение означава създаването на контейнер. Друго предимство на контейнеризацията е мащабируемостта. Разширяването става бързо: създават се нови контейнери за краткосрочни задачи. Контейнерите предлагат предимствата на изолация, преносимост, гъвкавост и контрол в целия жизнения цикъл на приложението.

Управлението на контейнери се извършва със специална софтуерна програма, наречена контейнер оркестратор. Когато работите в мащаб с много независими работещи контейнери, оркестрацията е от съствено значение. Следващата таблица описва общи задачи за оркестрация.

Таблица 1.5.  
тодо  
(тодо)

| **Tasks** | **Explanation** |
| --- | --- |
| Scheduling | Automatically provision container instances. |
| Affinity/anti-affinity | Provision containers nearby or far apart from each other, helping availability and performance. |
| Health monitoring | Automatically detect and correct failures. |
| Failover | Automatically reprovision a failed instance to a healthy machine. |
| Scaling | Automatically add or remove a container instance to meet demand. |
| Networking | Manage a networking overlay for container communication. |
| Service Discovery | Enable containers to locate each other. |
| Rolling Upgrades | Coordinate incremental upgrades with zero downtime deployment. Automatically roll back problematic changes. |

Оркестраторите на контейнери възприемат принципите за еднократност (Factor #9) и едновременност (Factor #8) от приложението с дванадесет фактора.

Въпреки че съствуват няколко контейнерни оркестратора, Kubernetes се превърна в де факто стандарт. Това е преносима, разширяема платформа с отворен код за управление на работни натоварвания в контейнери.

Облачните системи зависят от много различни спомагателни ресурси, като хранилища за данни, брокери на съобния, мониторинг и услуги за идентичност. Тези услуги са известни като поддържащи услуги. Фигура 1.14 показва много общи услуги за поддръжка, които използват облачните системи.



Фиг. 1.11. Спомагателни услуги използвани от облачните системи (Smith, 2022).

Облачните доставчици предлагат богат асортимент от спомагателни услуги. Облачния доставчикът управлява ресурса и носи отговорност за производителността, сигурността и поддръжката. Мониторингът, резервирането и наличността са вградени. Добрата практика за тези услуги е да се третират като прикачен ресурс, динамично свързан с главна микроуслуга чрез конфигурация (URL и идентификационни данни). Това ръководство е изложено в Приложението на дванадесетте фактора, обсъдено по-рано в главата. С този модел поддържащите услуги може да бъдат контролирани без промени в кода. Облачните доставчици предоставят API, за комуникация, чрез патентовани библиотеки, капсулиращи сложността. Добра практика е да се въведе междинен слой или API, който да предлага общи операции, „обвивайки“ кода на доставчика вътре в него. Това „слабо“ свързване позволява обновявания, без да се налага да се правят промени в кода на основната услуга.

## 1.3. Кибер защита на данните

ERP системите осигуряват персонализиран достъп на потребителите. Най-често се формират групи потребители с точно определени права за достъп (например счетоводители, касиери, стоковеди, мениджъри производство, технолози, логистични мениджъри). Всеки нов потребител се причислява към определена група.

Софтуерните приложения трябва да имат познание за потребителя или процеса, който ги извиква. Потребителят или процесът, взаимодействащ с приложение, е известен като принципал за сигурност, а процесът на удостоверяване и упълномощаване на тези принципали е известен като управление на самоличността (Vettor, 2022). Простите приложения могат да включват цялото им управление на самоличността в приложението, но този подход не се мащабира добре с много приложения и много видове принципали за сигурност. Windows, например, поддържа използването на Active Directory за предоставяне на централизирано удостоверяване и оторизация. Въпреки че това решение е ефективно в рамките на корпоративни мрежи, то не е предназначено за използване от потребители или приложения, които са извън домейна.

Удостоверяването е процес на определяне на самоличността на потребител. Упълномощаването е актът на предоставяне на удостоверено разрешение за извършване на действие или достъп до ресурс (Wike R, 2022).

Мобилни и базирани на JavaScript приложения се изпълняват на ниво клиент. Те трябва да комуникират с публичен API, което означава че не може да се използва защита като VPN. Изпращането на потребителско име и парола (Basic Authentication) при всяка заявка от клиент към API е лоша идея, защото например с анализатор на мрежови протоколи, има вероятност някои да проследи паролата и така да получи достъп до много информация.

Вместо да се изпраща комбинацията от потребителско име и парола при всяко повикване към API, се използват токени. Те представляват съгласие, например съгласие, дадено от потребителя за достъп. Токените се издават от централен компонент за решаване на общи задачи, свързани с идентичността. Отговорност на доставчика на самоличност е да удостовери потребител и безопасно да предостави доказателство на приложение. Често, доставчика на самоличност е и портал към задачи, свързани с потребителя и управлението на акаунта.

Регистрация на потребители, управление на акаунти, прилагане на политики за пароли с правила за силата, блокиране на потребители и т.н. са задачи към подсистемата за управление на идентичност и достъп. Тя позволява съвместимo използване в различни приложения, както и възможност цялата система да бъде тествана за пробив (penetration test). Важна функционалност е многофакторното удостоверяване, при което потребителят трябва да предостави допълнителна информация (освен име и парола), за да докаже своята самоличност, като например цифров пропуск, издаден от смартфон, който има приложение за удостоверяване.

OAuth2 е отворен протокол, който позволява сигурно оторизиране по стандартен метод от уеб, мобилни и настолни приложения. OAuth2 дефинира токен, който може да бъде поискан от клиентско приложение за да получи достъп до API, като го предаде по HTTP. OAuth определя крайни точки за оторизация, дефинирани по стандарт, като също така посочва как да бъдат използвани от различни типове приложения.

Следната таблица представя различни доставчици на самоличност които прилагат OAuth2.

Таблица 1.6.  
тодо  
(тодо)

|  |  |
| --- | --- |
| Okta |  |
| Trust Builder |  |
| Identity Server |  |

OpenID Connect е слой за идентичност, разширяващ, прилагащ принципите на OAuth2, който помага на приложенията да получат токен за самоличност/достъп. Той се използва за влизане в клиентско приложение, докато същото приложение използва токена за достъп до API.

Фигура 1.12. илюстрира комуникацията за получаване на токен.



Фиг. 1.12. Схема на OpenID Connect (Smith, 2022).

Създава се заявка от клиентското приложение, което пренасочва потребителя към доставчика на самоличност, където потребителят доказва кой е, чрез предоставяне на потребителско име (или например имейл, телефонен номер) и парола. Това действие може да е познато от влизане в сайт чрез Google или Microsoft. Това е началото на OpenID Connect потока. В крайна сметка доставчикът на самоличност създава токен, подписва го и го връща на клиентското приложение. Така то има доказателство за самоличност. OpenID Connect е проектиран да работи за различни видове приложения (уеб приложения от страна на сървъра или от страна на клиента, мобилни приложения, уеб услуги и др.), като дефинира типове клиенти и потоци.

Има два дефинирани типа клиенти: поверителни и публични клиенти. Поверителният клиент в състояние да запази поверителност на идентификационните си данни, които са клиентски идентификатор и тайна. Това не са идентификационните данни на потребителя. В OpenID Connect може да се удостовери както потребител, така и самото клиентско приложение. Идентификационните данни се съхраняват на ниво приложение, като например ASP.NET Core MVC уеб приложение, което работи на уеб сървър. Тъй като това приложение работи на сървъра, то може безопасно да съхранява идентификационните си данни, тъй като не са достъпни от потребителя. Публичните клиенти не са в състояние да запазят поверителност, тъй като изпълнението на програмата е в браузъра или на мобилно устройство, на ниво потребител. „Тайна“, съхранена в JavaScript или на мобилно устройство, не е тайна. Винаги е достъпна. В частност мобилните операционни системи позволяват използването на хардуерни API за безопасно съхраняване на тайни, но все о се считат за публични клиенти, поради факта, че тайната се съхранява на устройството.

Потокът в OpenID определя как кодът и/или токените се връщат на клиента. В случая разглеждате поток като редица заявки и отговори, често между клиентските приложения и доставчика на идентичност. Пример за такъв поток е сайт, който предлага опция за влизане с акаунт от Microsoft или Google. Браузър пренасочва към сайт, където се предоставя потребителско име и парола. След това се пренасочва обратно към основния сайт. В повечето от тези потоци има набор от заявки и отговори, които се случват, без да бъдат визуализирани от браузъра. OpenID потока се определя в зависимост от изискванията на клиента.

Заявките за оторизация се изпращат до крайна точка на ниво доставчик на идентичност. Тази крайна точка се използва от клиентското приложение за получаване на удостоверяване за токени за самоличност и/или оторизация за токени за достъп от потребителя. Това става чрез пренасочване на клиентското приложение към доставчика на идентичност. Код за оторизация или токен могат да бъдат върнати към клиентското приложение. TLS (SSL) е изискване за OpenID Connect, трафикът трябва да винаги е криптиран. Липсата на криптиран трафик прави потока уязвим към атаки от тип „човек по средата“.

Крайна точка за пренасочване е URI адресът, към който IDP пренасочва обратно клиента. Той е на ниво клиентско приложение и се използва от доставчика на идентичност, за да достави кода за оторизация или токен на клиентското приложение.

Крайна точка за токен е на ниво доставчик на идентичност. Клиентските приложения могат програмно да изискат токени. Обикновено става без пренасочване чрез методът HTTP POST. Повикванията могат да бъдат удостоверени с идентификатор и тайна за поверителните клиенти.

Както за поверителни, така и за публични клиенти се препоръчва използването на вариации на потока „Кода за оторизация“ (Authorization Code Flow) подплатен с PKCE защита (OAuth 2.0 Security Best Current Practice RFC / <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics-19>). Потокът „код за оторизация“ получава името си от факта, че започва с код за оторизация. Този код за оторизация може да се разгледа като краткотраен идентификационен номер за еднократна употреба, който се използва за проверка дали влезлият на ниво доставчик на идентичност е същият, който е започнал на ниво клиентско приложение. Този поток позволява дълготраен достъп за поверителни клиенти, но за публичните е ограничен. Разрешена е вариация на токен за опресняване. Изборът на правилен вариант на поток е важен, защото може да доведе до уязвимости в сигурността. Технически, много от подходите работят, но повечето доведат до сериозен риск.

Всеки OpenID Connect поток започва със заявка към крайната точка за оторизация. Пример за URI е представен на следната фигура.



***Фигура 1.12****: URI for the authentication code flow endpoint.*

На първо място е самата крайна точка на ниво IDP. След това са идентификаторът на клиентското приложение (client\_id), URI за пренасочване. Това е URI на ниво клиентско приложение, до което бъде доставен отговорът от заявката за оторизация. В случая, зададения обхват показва, че приложението изисква достъп до и до идентификатора на потребителя (openid) и твърденията, свързани с профила, като собствено име и фамилия. След това е стойността на типа отговор (response\_type), която определя потока, който се използва. Стойностите, зададени като response\_type, са нещата или токените, върнати от доставчика на идентичност към клиента, чрез пренасочване от браузъра. Фигура 1.13 представа модела на този поток.



***Фигура 1.12****: authentication code flow model.*

Уеб приложение създава и изпраща заявка за удостоверяване с код. На ниво IDP потребителят предоставя комбинация от потребителско име и парола. В този момент доставчикът на самоличност знае кой е потребителят. Клиентското приложение все о не. Доставчикът на самоличност по избор иска съгласие от потребителя, като например да изиска разрешение за получаване на достъп до информацията от профила. След това доставчикът на самоличност изпраща обратно към уеб приложението чрез URI пренасочване или POST формуляр. Това е комуникация, която е видима за браузъра. Кода за оторизация може да се разглежда като много краткотраен токен, който е доказателство за удостоверяване. Той е свързан с потребителя, който току-що е влязъл в доставчика на самоличност. В следващата стъпка уеб клиентът извиква крайната точка за токен. Тази заявка не използва пренасочване и следователно не е видима за браузъра. Това е HTTP заявка от сървър към сървър, към която се предава кода за оторизация и комбинация от clientid и clientsecret. Доставчикът на самоличност генерира токен и го връща на клиента в HTTP отговора. На ниво клиент токенът се валидира и извлича идентификатора на потребителя.

Съвременните решения за самоличност обикновено използват токени за достъп, които се издават от защитена услуга/сървър (STS) на принципал за сигурност, след като тяхната самоличност бъде определена (Anil N, 2022).

Удостоверяването чрез токени е механизъм без състояние, тъй като никаква информация за потребителя не се съхранява в паметта на сървъра или базaтa от данни, за разлика от бисквитките (Gichuhi , 2021).

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;