**TЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ** 

**към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ**

**ДИПЛОМНА РАБОТА**

**по професия код**  **523050 „Техник на компютърни системи“**

**специалност код 5230502 „Компютърни мрежи“**

Тема: Защита на контейнеризирани приложения в облачни среди, базирани на Kubernetes

Дипломант: Дипломен ръководител:

*Димитър Мартинов Цветков Валентин Петров*

СОФИЯ

2 0 2 4

* 1. 
  2. **ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ**

**към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ**

Дата на заданието: 15.11.2023 г. Утвърждавам:..............................

Дата на предаване: 15.02.2024 г. / проф. д-р инж. П. Якимов /

**ЗАДАНИЕ**

**за дипломна работа**

**ДЪРЖАВЕН ИЗПИТ ЗА ПРИДОБИВАНЕ НА ТРЕТА СТЕПЕН НА ПРОФЕСИОНАЛНА КВАЛИФИКАЦИЯ**

**по професия код**  **523050 „Техник на компютърни системи“**

**специалност код 5230502 „Компютърни мрежи“**

на ученика Димитър Мартинов Цветков от 12 Г клас

1. Тема: Защита на контейнеризирани приложения в облачни среди, базирани на Kubernetes
2. Изисквания:
3. Да се напише уеб приложение на Python със софтуерната рамка Django.
4. Да се изгради контейнер имидж на приложението от точка 1.
5. Да се създаде Kubernetes клъстер в Google Cloud Platform.
6. Да се създадат необходимите Kubernetes ресурси, за да се постигне работоспособност на приложението и да се постави разпределител на мрежовия трафик пред приложението в клъстера.
7. Да се подсигури композитна защита на приложението като тя съдържа минимално следните компоненти - политики за мрежови достъп, политики за сигурност на ниво контейнер, уеб-базирана защитна стена и приложение за контрол на пароли.
8. Съдържание 3.1 Теоретична част

3.2 Практическа част

3.3 Приложение

[**Становище от дипломния ръководител 5**](#_mdv6r9ryuxd6)

[**Увод 7**](#_ppr5qsix5ka2)

[**1. Първа глава - Проучване на принципи, практики и технологии за защита на контейнеризирани приложения в облачни среди 9**](#_kjgmvj7t34h6)

[1.1. DevSecOps 9](#_734ab0dwwqrq)

[1.2. Cloud Technologies 14](#_fj0wkw7rzobk)

[1.3. Docker 15](#_p5cqyim707qe)

[1.4. Принципи за сигурност на контейнерите 18](#_jdem2a4rjgfu)

[1.5. Kubernetes 19](#_7m51o61kpozm)

[1.6. Kubernetes Resources 21](#_magy44w0wzo1)

[1.7. Мрежови политики на Kubernetes, контекст на сигурността 24](#_sp0rkxv1ipt7)

[1.8. Управление на конфиденциална информация 25](#_1ap06pinzzbi)

[1.9. Решения за управление на чувствителна информация в облачни технологии 28](#_dw6uv66vpoun)

[1.10. Заплахи за сигурността, рискове и защита 29](#_gsqmf9ie7ytu)

[1.11. Облачни технологии 32](#_1p9djscmmmfk)

[1.12. Google Cloud (VPC, GKE, инстанция) 34](#_s2hvhwpu3660)

[1.13. Helm 36](#_pj3uu9htk9q)

[1.14. Ingress Controller - NGINX 38](#_laa0p7zijd16)

[1.15. Git 40](#_2rl10csb6mt8)

[1.16. Web-application firewall 42](#_2w9mg2ar58b3)

[1.17. NGINX ModSecurity, Google Cloud WAF solutions 44](#_sjabwd2ncssj)

[1.18. Python 47](#_1ql45jnpm4q7)

[1.19. Django 49](#_b1yugwba6ir3)

[**2. Втора глава - Функционални изисквания 51**](#_ilqo26xbko9)

[2.1. Блокова схема и принцип на работа 51](#_x05fmswm2gbl)

[2.2. Функционални изисквания за създаване на уеб приложение на Python със софтуерната рамка Django. 52](#_1qjo6zxgen3r)

[2.3. Функционални изисквания за изграждане на контейнер имидж на уеб приложението 53](#_67xvqwb8xyl3)

[2.4. Функционални изисквания за създаване на Kubernetes клъстер в Google Cloud Platform 54](#_psw5ppbiq3xe)

[2.5. Функционални изисквания за създаване на необходимите Kubernetes ресурси, за постигане работоспособност на приложението и поставяне на разпределител на мрежовия трафик пред приложението в клъстера. 56](#_4m80kicl4iyb)

[2.6. Функционални изисквания за подсигуряване композитна защита на приложението 58](#_o28idby7yk0p)

[**3. Трета глава - Реализация на защита на контейнеризирани приложения в облачни среди, базирани на Kubernetes 61**](#_fdp3s1q3axe1)

[3.1. Създаване на уеб приложение на Python със софтуерната рамка Django. 61](#_t2igf7sl8u9d)

[3.2. Изграждане на контейнер имидж на приложението 89](#_f89i6za0zgab)

[3.3. Създаване на необходимите Kubernetes ресурси, за да се постигне работоспособност на приложението и поставяне на разпределител на мрежовия трафик пред приложението в клъстера 93](#_nf7r1g5pec5t)

[3.4. Качване на приложението в GCP 100](#_rucojmk4kndt)

[**4. Четвърта глава - Доказване на работоспособност 104**](#_htxv4adiy7dh)

[4.1. Доказване работоспособността на приложението 104](#_fdakrqhapbvh)

[4.2. Доказване създаден контейнер имидж 105](#_4rs5yvk99t3w)

[4.3. Доказване работоспособността на NGINX 106](#_avyiob9c8g8b)

[4.4. Доказване защита на ниво контейнер 106](#_mz6j7cpvt6t)

[4.5. Доказване работоспособност на WAF 107](#_wea4btg2d93k)

[4.6. Доказване работоспособност на мрежовата политика 107](#_bijvejcirvpf)

[4.7. Доказване работоспособност на клъстер в GCP среда 108](#_jsma5zsaiot5)

[**Заключение 110**](#_yeqcoqujprkk)

[**Използвана литература 112**](#_fy4wwy7xd67a)

# **Становище от дипломния ръководител**

Дипломантът се справи отлично с комплексната тематика „Защита на контейнеризирани приложения в облачни среди, базирани на Kubernetes“. Крайният резултат от дипломната работа се състои в детайлно разглеждане и реализация на всяко поставено изискване. Този труд показва развито и задълбочено разбиране на поставената тематика в сферата на сигурността на облачните услуги и големия потенциал на дипломанта за развитие в сферата на информационните технологии. На базата на тези постижения, предлагам дипломантът Димитър Цветков да бъде допуснат до защита. В ролята на рецензент предлагам Галин Димитров, Главен Инженер, притежаващ експертиза и дългогодишен опит в сферата.

Дата: ..................... Ръководител: .....................

/ Валентин Петров /

# **Увод**

В съвременния цифровизиран свят, облачните технологии заемат централно място в разработката и доставката на софтуерни приложения. С разрастването на сложността на приложенията, използващи контейнеризация и различни видове инструменти станават основен стълб в инфраструктурата, осигуряваща гъвкавост и ефективност в мащабирането и управлението на приложенията в облачна среда.

С нарастването на зависимостта от облачните услуги и развитието на дигиталната инфраструктура, защитата на приложенията в облачни среди става ключов аспект от сигурността на информационните системи. Облачните услуги предоставят гъвкавост и мащабируемост, но същевременно изискват особено внимание и стратегии за защита, като се има предвид постоянната еволюция на киберзловредни атаки и заплахи.

С разрастващия се брой на облачните приложения и внедряването на контейнеризация, базирана на технологии като Kubernetes, възникват нови предизвикателства по отношение на сигурността. Защитата на контейнеризирани приложения в облачни среди е от съществено значение, тъй като тези технологии предоставят гъвкавост и ускоряват процесите на разработка и доставка на софтуер.

Дипломната работа има за цел да разгледа съвременните предизвикателства, свързани със защитата на приложенията в облачни среди и да предложи иновативни решения за усъвършенстване на сигурността в този контекст.

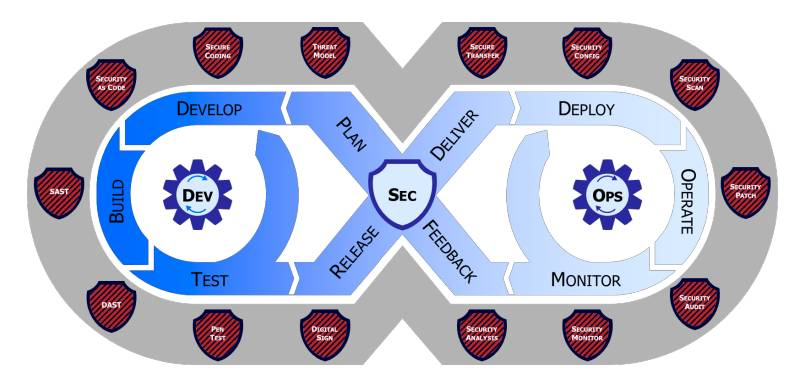
Главна цел е да се изследват и разгледат съществуващите методи и технологии за защита на контейнеризирани приложения в облачни среди, фокусирайки се върху Kubernetes като един от водещите инструменти за управление на контейнери. Анализът ще включва изследване на общите уязвимости, които могат да възникнат при използването на тези технологии, както и разработването на стратегии и инструменти за повишаване на сигурността.

# **Първа глава - Проучване на принципи, практики и технологии за защита на контейнеризирани приложения в облачни среди**

## **DevSecOps**

DevSecOps описва практиките и процесите на организацията, които ѝ позволяват да преодолее пропастта между разработчиците, екипа по сигурността и оперативния екип, да се подобрят процесите чрез съвместните и гъвкави работни процеси, стремеж към по-бърза и по-сигурна доставка на софтуер чрез технология, и постигане на последователно управление и контрол. Не съществува единна DevSecOps практика. Възприемане на DevSecOps изисква от организациите да променят своята методология, да развият съществуващите процеси, да приемат нови технологии и укрепване на управлението.

DevSecOps, съкратено от разработка (Dev), сигурност (Sec) и операции (Ops), е концепция, която интегрира сигурността във всяка фаза от процеса на разработката на софтуер. Това е ключова практика за създаването на ефективен и сигурен софтуер, като се гарантира, че сигурността е вградена във всяка фаза на процеса на разработка и доставка. Фиг. 1.1 илюстрира този процес. Тя показва как сигурността (Sec) свързва разработката (Dev) и операциите (Ops), като включва етапи като планиране, разработка, изграждане, тестване, доставка, разгръщане, работа и мониториране. Всеки от тези етапи е обогатен с конкретни практики по сигурност, които са включени в целия процес.

**

*Фиг 1.1 - Процеси на DevSecOps*

Фигурата изобразява методологията на DevSecOps, който интегрира практиките за сигурност (Sec) в традиционния процес на DevOps. Ето подробно обяснение на всеки от показаните процеси [3]:

* **Планиране:**

Това е началната фаза, в която се определят изискванията, обхватът и сроковете на проекта. Екипите обикновено използват гъвкави методологии за планиране на итерации на разработка, включително сигурностни разглеждания от самото начало.

* **Разработка:**

В този етап софтуерът се кодира и създава. Сигурността се интегрира в процеса на разработка чрез използване на практики за сигурно кодиране, които могат да сканират за проблеми със сигурността, преди кодът да бъде качен в основната кодова база.

* **Създаване:**

Фазата на създаване включва компилиране на код, изпълнение на тестове за единици и интеграция и производство на артефакти, които могат да бъдат разгънати. Инструменти за сигурност като анализ на софтуерния състав (SCA) могат да бъдат използвани за идентифициране на известни уязвимости в отворените източници.

* **Тест:**

Автоматизирани тестове се изпълняват, за да се гарантира, че софтуерът отговаря на необходимите стандарти за качество и сигурност. Това включва динамичен анализ, сканиране за уязвимости и тестване на проникване, за да се намерят слабости в сигурността.

* **Пускане:**

Фазата на пускане е там, където софтуерът се подготвя за пускане в производството. Това включва окончателни прегледи на сигурността и проверки за съответствие, за да се гарантира, че софтуерът спазва необходимите стандарти и регулации за сигурност.

* **Доставка:**

Доставката е процесът на разгръщане на софтуера в производствена среда. Автоматизираните инструменти за разгръщане могат да включват проверки за сигурност, които предотвратяват разгръщането, ако открият проблеми.

* **Разгръщане:**

Това е етапът, в който приложението се разгръща действително в производствената среда. Практиките за сигурност тук включват гарантиране на сигурна конфигурация на средата и използване на инфраструктурата като код, за да се поддържа последователност и проследяемост.

* **Работа:**

След като софтуерът е разгънат, той трябва да бъде управляван и работен. Операциите по сигурност включват непрекъснато наблюдение на приложението за подозрителни дейности и потенциални нарушения на сигурността.

* **Наблюдение:**

Непрекъснато наблюдение на оперативната среда за откриване и реагиране на заплахи в реално време. Системите за управление на информацията и събитията по сигурността (SIEM) често се използват за събиране и анализ на логове за признаци на инциденти по сигурността.

* **Обратна връзка:**

Обратните връзки са критични за процеса на DevSecOps. Възгледите от наблюдението и операциите се връщат обратно в етапите на планиране и разработка, за да информират подобренията както в функционалността, така и в сигурността.

Около целия процес са различни практики за сигурност, вероятно указващи, че сигурността е постоянна грижа на всяка фаза.

* **Анализ на сигурността:**

Непрекъснато анализиране на сигурността на приложението и инфраструктурата.

* **Тестване на сигурността:**

Редовно тестване на мерките за сигурност на място, за да се гарантира, че те са ефективни.

* **Наблюдение на сигурността:**

Непрекъснато наблюдение за инциденти или уязвимости по сигурността.

* **Аудити на сигурността:**

Периодични одити, за да се гарантира съответствие с политиките и стандартите за сигурност.

* **Моделиране на заплахи:**

Идентифициране на потенциални заплахи и уязвимости рано в цикъла на разработка.

* **Анализ на изходния код:**

Преглед на изходния код за проблеми със сигурността.

* **Обучение по сигурност:**

Гарантиране, че всички членове на екипа са обучени в най-добрите практики по сигурност.

Символите “Dev” и “Ops” в центъра на кръга представляват ядрото на “DevOps”, където екипите за разработка и операции сътрудничат тясно, а включването на Sec в средата подчертава важността от интегриране на сигурността в двете тези основни функции.

## **Cloud Technologies**

Облачни технологии се отнасят до доставката на различни услуги чрез интернет, включително съхранение на данни, сървъри, бази данни, мрежи и софтуер. Тези технологии позволяват на потребителите и предприятията да използват компютърни ресурси, хоствани във виртуални среди, управлявани от трети страни, вместо да разчитат на локални сървъри или лични компютри. [10]

* **Основни Характеристики**
* Гъвкавост и Мащабируемост: Облачните технологии предлагат изключителна гъвкавост и мащабируемост, като позволяват на бизнесите бързо да добавят или намаляват ресурсите в зависимост от техните текущи нужди, което оптимизира разходите и ефективността.
* Достъпност: Те осигуряват достъп до информация и приложения по всяко време и от всяка точка на света, стига да има интернет връзка, което подобрява мобилността.
* Капиталови и Оперативни Разходи: Преминаването към облачни услуги може значително да намали капиталовите и оперативните разходи, тъй като изисква по-малко инвестиции в хардуер и поддръжка.
* Надеждност: Доставчиците на облачни услуги предлагат висока надеждност чрез използването на допълнителни компоненти или системи, които могат да бъдат използвани в случай на неизправност на основните и резервно копиране, гарантирайки, че данните и услугите са достъпни дори при хардуерни откази.
* **Модели на Обслужване**
* Infrastructure as a Service (IaaS): Предоставя виртуалнизирани компютърни ресурси през интернет. AWS, Azure и Google Cloud са водещи доставчици на IaaS, като предлагат сървъри, съхранение и мрежова инфраструктура.
* Platform as a Service (PaaS): Осигурява среда за разработка и разгръщане на приложения, като облекчава потребителите от управлението на базовата инфраструктура. Примери включват Google App Engine и Microsoft Azure.
* Software as a Service (SaaS): Доставя софтуер като услуга чрез интернет, като елиминира нуждата от инсталация и поддръжка на софтуер на локални компютри. Популярни SaaS продукти включват Google Workspace, Microsoft 365 и Salesforce.

## **Docker**

Docker е иновативна платформа за контейнеризация, която позволява на разработчиците да компресират приложения в контейнери – стандартизирани изпълними компоненти, които комбинират изходния код на приложението с операционната система (ОС) библиотеките и нужните файлове, необходими за това приложение да работи. Това улеснява разработването, тестването и разпространението на приложения, като гарантира, че те ще работят във всяка среда. Основни концепции [1]:

* **Контейнери:**

В същността на Docker са контейнерите. За разлика от традиционните виртуални машини (ВМ), които виртуализират цялата хардуерна система за стартиране на цяла ОС, контейнерите споделят ядрото на хост ОС и значително намаляват натоварването и изискванията за ресурси. Те са по-леки, стартират почти мигновено и предлагат по-голяма портативност.

* **Dockerfile**:

Това е текстов файл, който съдържа всички команди, които потребител може да извика на командния ред за създаване на Изображение. Той автоматизира процеса на създаване на Image, като улеснява разпространението и стандартизацията на средите за разработка и производство.

* **Docker Images:**

Използват се за създаване на контейнери. Те са съставени от серия от слоеве, създадени чрез серия от инструкции във Dockerfile файла. Всеки слой представлява инструкция в Dockerfile и слоевете се кешират, което означава, че често използваните слоеве от образи могат да се споделят между контейнери, което увеличава ефективността и скоростта на разпространение.

* **Docker Hub и Registry:**

Docker Hub е облачна услуга, която позволява споделянето и управлението на Изображения на Docker. Потребителите могат да теглят Изображения от обществени репозитории или да качват своите собствени в частни хранилища. Docker Registry е приложение, което се управлява от потребителя и служи за хостинг и дистрибуция на образи.

* **Docker Compose:**

Инструмент, който улеснява работата с множество контейнери на Docker. С Docker Compose, разработчиците могат да дефинират и стартират мулти-контейнерни Docker приложения с един файл, наречен docker-compose.yml. Този файл позволява конфигуриране на услугите, мрежите и томовете, и осигурява начин за стартиране и свързване на приложенията със зависимостите им.

* **Docker Swarm:**

Оркестрационен инструмент за управление на клъстер от Docker контейнери. С Docker Swarm, разработчиците могат да мащабират приложенията си, като добавят повече контейнери, и да гарантират висока наличност и отказоустойчивост. Docker Swarm осигурява лесен интерфейс за управление на контейнерите в различни хостове и за автоматично разпределение на натоварването между тях.

Безопасността е съществен аспект на Docker. Платформата осигурява средства като Docker Security Scanning и Docker Bench for Security, които помагат на разработчиците да идентифицират и отстраняват уязвимости в образите и контейнерите. Docker също така използва различни механизми на изолация, като например namespaces и cgroups на Linux, за да ограничи потенциалното влияние на уязвимите контейнери.

## **Принципи за сигурност на контейнерите**

Сигурността на контейнерите представлява важен аспект в контекста на разработването и разпространението на приложения в среди, базирани на контейнеризация. Въпреки че контейнерите споделят ядрото на операционната система на домакинската машина и допълнителни ресурси, което ги прави по-ефективни по отношение на ресурсите в сравнение с традиционните виртуални машини, това създава сигурностни предизвикателства. Принципи за осигуряване на сигурността на контейнерите [4]:

* **Минимизация на контейнерните образи:**

Важно е да се използват образи от официални и проверени източници, като се предпочитат официални Изображения, предлагани в регистри като Docker Hub. Това намалява риска от внедряване на зловреден код. Освен това, оптимизацията на сигурността на Изображението чрез използване на минимални базови Изображения, като например Alpine Linux, които включват само необходимите пакети и библиотеки, ефективно намалява повърхността на потенциални атаки.

* **Управление на привилегиите**

Експлоатацията на контейнери с минимални привилегии е от съществено значение за предотвратяване на неоторизиран достъп или увеличаване на привилегии. Разработването на контейнери, които се изпълняват без привилегии, и използването на User Namespaces за разделение на пространствата от идентификатори на потребителите и групите между контейнерите и домакинската система са важни за поддържането на сигурна операционна среда.

* **Управление на зависимостите и актуализациите**

Редовното актуализиране на контейнерните Изображения за включване на последните сигурностни корекции е важно за защита срещу уязвимости. Автоматизираното сканиране за уязвимости преди експлотацията на контейнери осигурява ранно откриване и коригиране на потенциални заплахи.

* **Управление на конфигурациите**

За да се избегне излагането на чувствителна информация, е препоръчително да се използват частни ключове за съхранение на конфиденциални данни вместо да се вграждат директно в контейнерните образи. Правилната конфигурация на мрежовите политики за ограничаване на комуникацията между контейнерите до строго необходимия минимум подпомага защитата на приложенията от мрежови атаки.

## **Kubernetes**

Kubernetes, често съкращаван като K8s, е отворена платформа за автоматизация на разполагането, мащабирането и управлението на приложения, работещи в контейнери. Разработен първоначално от Google и впоследствие дарен на Cloud Native Computing Foundation (CNCF), Kubernetes се е утвърдил като стандарт за оркестрация на контейнери в индустрията. Ключови компоненти на Kubernetes [2]:

* **Подове (Pods):**

Най-малката и основна единица на Kubernetes, под представлява група от един или повече контейнери, които споделят мрежови и съхранителни ресурси.

* **Услуги (Services):**

Абстракция, която дефинира логически набор от подове и политика за достъп до тях, често чрез постоянен IP адрес или DNS име.

* **Разполагане (Deployment):**

Описва желаното състояние на подовете, като например кое изображение на контейнер да се използва и колко копия (реплики) от пода да се поддържат.

* **Контролери за репликации (ReplicaSets):**

Гарантира, че определен брой реплики на под са работещи по всяко време.

* **Именни пространства (Namespaces):**

Изолира групи от ресурси в рамките на едно физическо клъстер.

* **Възли (Nodes):**

Физически или виртуални машини, които изпълняват подовете и са управлявани от Kubernetes мастър.

* **Мастър (Master):**

Контролният възел, който управлява клъстера и координира всички дейности като разполагане на приложения и поддържане на желаното състояние.

* **Архитектура на Kubernetes:**

Kubernetes следва архитектурен модел, който включва един или повече работни възли (worker nodes) и мастър възел (master node). Мастър възелът хоства компоненти като API (Application Programming Interface) сървъра (API Server), контролния мениджър (Controller Manager) и метаданните (etcd), които са от съществено значение за управлението на клъстера.

## **Kubernetes Resources**

Кubernetes е система за управление на контейнери, която позволява на потребителите да управляват и оркестрират приложения в контейнери. Kubernetes Resources са обекти, които представляват използваните ресурси от Kubernetes. Тези ресурси могат да бъдат използвани за управление на приложенията в контейнери, като например за задаване на лимити за паметта и процесора, които могат да бъдат използвани от приложенията. Kubernetes Resources са дефинирани в YAML файлове и могат да бъдат създадени, редактирани и изтривани чрез Kubernetes API.

* **YAML файловете в Kubernetes:**

Използват се за дефиниране, конфигуриране и управление на ресурсите в Kubernetes клъстер. Те предоставят декларативен начин за специфициране на желаните състояния на ресурсите, като например рамка (ingress), услуги (services), прикачване (deployments) и други.

* **Kubernetes API (Application Programming Interface):**

Oсновният компонент за управление и взаимодействие с Kubernetes клъстера. Чрез него потребителите могат да създават, променят, изтриват и получават информация за ресурсите на Kubernetes, като подове (pods), услуги (services), разгръщания (deployments) и др. API-то играе критична роля в екосистемата на Kubernetes, тъй като предоставя консистентен и стандартизиран начин за автоматизиране на операции и управление на клъстера. Структура на Kubernetes API:

- API Versioning: Kubernetes API е има различни версии, което помага за управление на промените и съвместимостта. Всяка API заявка включва версия, която указва стабилността и наличността на функционалностите.

- API Groups: Ресурсите в Kubernetes са организирани в API групи, за да се улесни разширението на API. Например, core групата включва основните ресурси като Pods, Services и други. Други групи като apps, batch, extensions включват специфични ресурси като Deployments, Ingresses и други.

- Resources and Kinds: Всеки ресурс в Kubernetes API има уникален тип, наречен "Kind", например, Pod, Service, Deployment. Тези типове са описани в YAML или JSON формати при изпращане на заявки към API.

Основни операции на Kubernetes API:

- Create: Създаване на нов ресурс.

- Get: Четене на детайли за специфичен ресурс или списък от ресурси.

- Update: Промяна на съществуващ ресурс.

- Delete: Изтриване на ресурс.

- Watch: Наблюдение за промени в ресурсите в реално време.

Kubernetes поддържа множество механизми за автентикация (като сертификати и други) и авторизация (чрез Достъп, базиран на роля (Role-Based Access Control - RBAC), Достъп, базиран на атрибут, принадлежащ към роля (Attribute-Based Access Control - ABAC), и други), за да гарантира, че достъпът до ресурсите е сигурен и ограничен до упълномощени потребители [6].

Kubernetes API предоставя надеждни средства за управление на сигурността на клъстера, включително TLS за шифроване на данни при транспорт и различни стратегии за автентикация и авторизация, за да се контролира достъпът до ресурсите.

Kubernetes API е изключително полезен и удобен интерфейс, който позволява на разработчиците и системните администратори да управляват клъстера по гъвкав и автоматизиран начин, правейки го ключов компонент в управлението на контейнеризирани приложения и услуги.

## **Мрежови политики на Kubernetes, контекст на сигурността**

Мрежовите политики в Kubernetes представляват ключов елемент от сигурността на клъстера, позволяващ администраторите да контролират трафика между подовете (pods) вътре в един Kubernetes клъстер. Тези политики са особено важни в среди с много потребители, където множество приложения или услуги, потенциално с различни нива на сигурност и доверие, споделят обща инфраструктура [10].

* **Основи на Мрежовите Политики в Kubernetes**

Мрежовите политики в Kubernetes се дефинират чрез обекти NetworkPolicy, които специфицират как подовете могат да комуникират помежду си и с други мрежови крайни точки. По подразбиране, подовете в Kubernetes могат да инициират връзки към всяка друга точка вътре и извън клъстера, освен ако не бъде зададена мрежова политика, която ограничава тези комуникации [13].

* **Дефиниране на Мрежови Политики**

Мрежовата политика е декларативно описание, което използва етикети за идентифициране на потоци от трафик, които могат да бъдат разрешени или блокирани. Това позволява на администраторите да дефинират сложни правила за мрежовия достъп, базирани на ролята, функцията или други характеристики на подовете.

* **Контекст на Сигурността**

Мрежовите политики в Kubernetes са част от по-широкия контекст на сигурността, който включва идентификацията и управление на достъпа, шифроване на данни, както и защита на сензитивна информация. Внимателното управление на мрежовите политики е критично за предотвратяване на нежелан достъп и ограничаване на потенциалните зловредни атаки или конфигурационни грешки.

В допълнение към мрежовите политики, е важно да се конфигурират сигурностни контексти за подове и контейнери, които могат да включват настройки за привилегии, контрол на достъпа до файловата система, ограничения за използване на системни ресурси и други мерки за сигурност.

## **Управление на** конфиденциална **информация**

Управлението на чувствителна информация е важен компонент сигурността, включващ безопасното управление, съхранение и контрол на достъпа до чувствителна информация като пароли, API ключове, сертификати и други удостоверения. В контекста на разработката на софтуер и операциите, ефективното управление на важна информация предотвратява неоторизирания достъп и излагането на конфиденциални данни, като по този начин защитава от нарушения на данните и гарантира съответствието с регулаторните стандарти.

* **Основни аспекти на управлението на данни**
* Информацията трябва да се съхранява в криптирана форма, като по този начин не са достъпни в явен текст за неоторизирани потребители или системи. Това се отнася както за съхранението на диск, така и за трансфера през мрежи.
* Достъпът до частна информация трябва да бъде строго контролиран и ограничен до обектите, които ги изискват за своята операция. Това обикновено се постига чрез механизми за контрол на достъпа въз основа на роли (RBAC), където на потребителите или услугите се предоставят разрешения въз основа на техните роли в организацията.
* Поддържането на регистрации за това кога и от кого са достъпени данните осигурява видимост върху използването на чувствителна информация и помага в откриването или разследването на неоторизиран достъп или други сигурностни инциденти.
* Информацията трябва редовно да се променя за минимизиране на риска от компрометиране. Допълнително, трябва да има механизми за бързо отмяна на достъп до данни, ако се подозира, че са компрометирани или вече не са необходими.
* **Реализиране на управление на частна информация**

Има различни инструменти и услуги, проектирани да улесняват управлението на чувствителна информация:

* Специализирани инструменти за управление на данни: Решения като HashiCorp Vault, AWS Secrets Manager, Azure Key Vault и Google Cloud Secret Manager предоставят добри методи за съхранение, достъп и управление на информация.
* Сензитивна информация в Kubernetes: В контейнеризирани среди, Kubernetes предлага обект Secret, който съхранява и управлява чувствителна информация. Данните в Kubernetes се съхраняват в клъстера и могат да бъдат интегирани в подове или достъпни от услуги по нужда, въпреки че имат известни ограничения по отношение на криптирането и контрол на достъп в сравнение със специализираните решения за управление на информация.

Ефективното управление на чувствителна информация е съществено за поддържането на сигурността и целостта на системите и данните във всяка ИТ среда. Чрез прилагането на доказани практики за управление на данни, организациите могат значително да намалят риска от изтичане на данни и да спазват наложени изисквания.

## **Решения за управление на чувствителна информация в облачни технологии**

Управлението на сензитивна информация в облачно-ориентирани среди се отнася до инструментите и практиките, предназначени за сигурното съхранение, управление и достъп до данни - като пароли, частни ключове и сертификати - в среди, базирани на облачни технологии. Тези решения са важни за поддържането на сигурността и целостта на приложенията и инфраструктурата, особено в динамичните, разпределени системи, които често се срещат в облачно-ориентирани архитектури. Те са проектирани да работят безпроблемно с облачни платформи, контейнеризирани приложения, предлагайки съществени функции за сигурност, които са в съответствие с принципите на DevOps, непрекъснато интегриране/непрекъснато разполагане (CI/CD) и практики за разработка.

* **Популярни решения за управление на тайни в облачно-ориентирана среда**
* HashiCorp Vault: Широко използван инструмент, който предоставя сигурно управление на данни, криптиране като и услуга за достъп, базиран на идентичност. Vault е проектиран да покрива изискванията на среди с висок обем и трафик.
* AWS Secrets Manager: Услуга от Amazon Web Services, предназначена за сигурно съхранение, управление и извличане на конфиденциална информация, като пароли, частни ключове и други. Тази услуга е специално разработена за улесняване на администраторите и разработчиците в облачната среда при управлението на данни, които се използват от приложения, бази данни и други услуги.
* Azure Key Vault: Предлага управление на сензитивна информация, ключове и сертификати, проектирани да осигурят и строго контролират достъпа до пароли, сертификати, частни ключове и други.
* Google Cloud Secret Manager: Управляема услуга, която предоставя сигурно и удобно съхранение и управление на тайни, интегрирана с IAM на Google Cloud за контрол на достъпа.
* Kubernetes secret - метод за съхранение на чувствителна информация в рамките на Kubernetes клъстер, но тази информация не е криптирана по подразбиране.

## **Заплахи за сигурността, рискове** и **защита**

Заплахите за сигурността, рисковете и защитата в контекста на облачните технологии обхващат широк спектър от предизвикателства и технологии, предназначени да защитят цифровите активи, данните и инфраструктурата от неоторизиран достъп, нарушения и други форми на кибератаки.

* **Сигурностни Заплахи и Рискове**
* Сигурностните заплахи се отнасят до всяка потенциална злонамерена атака или неоторизиран достъп, която цели да открадне, промени или унищожи ресурс. Това включва разнообразие от различни видове атаки като зловреден софтуер, фишинг, DDoS атаки и вътрешни заплахи.
* Сигурностните рискове са потенциални уязвимости в системите или процесите на организацията, които могат да бъдат експлоатирани, за да се получи неоторизиран достъп или да се нанесе вреда. Рисковете често се идентифицират чрез оценка на риска и могат да произтичат от липса на подходящи сигурностни мерки, грешки в конфигурацията, остарели системи и други.
* **Защита в облачни технологии**

Защитата е процесът на сигурност на система чрез намаляване на нейната уязвимост. Това означава конфигуриране на операционни системи, приложения и мрежова инфраструктура за елиминиране на възможно най-много рискове за сигурността. Защитата включва редица практики:

* Редовно актуализиране на софтуера и системите за адресиране на известни уязвимости.
* Прилагане на достъп с най-малко привилегии, технологии за удостоверяване и контрол на достъпа въз основа на роли (RBAC) за да се гарантира, че само упълномощени потребители имат достъп до чувствителни данни и системи.
* Шифроване на данни за защита на чувствителна информация от прехващане или кражба.
* Защита на всички устройства, които се свързват към мрежата, включително работни станции и мобилни устройства, чрез използване на антивирусен софтуер и решения за откриване и реагиране на крайни точки (EDR).
* Редовно сканиране и намаляване на уязвимостите в средата на организацията.
* Разработване на процедури за резервно копиране и възстановяване при извънредни ситуации за гарантиране на целостта и наличността на данните в случай на нарушение или загуба на данни.
* Използване на технологии, предоставени доставчиците на облачни услуги, като управление на идентификацията и достъпа (IAM), сигурностни групи и автоматизирани услуги за следене.
* **Адресиране на проблеми, специфични за облачните технологии**

Облачните среди въвеждат специфични сигурностни разглеждания поради техния модел на споделена отговорност, където сигурността на облачната инфраструктура се управлява от доставчика, докато сигурността в облака (как се използват услугите) се управлява от клиента. Това включва:

* Поверителност на Данните: Разбиране къде се съхраняват данните и гарантиране, че те съответстват на регионалните регулации и закони за поверителност.
* Изолиране на ресурсите и данните в споделена среда за предотвратяване на достъп между различни клиенти.
* Адаптиране на мерките за сигурност за защита на динамично мащабиращи се среди и работни натоварвания.

В заключение, адресирането на сигурностни заплахи, управлението на рискове и прилагането на стратегии за защита в облачни технологии изисква многостранен подход. Организациите трябва да останат информирани за възникващите заплахи, непрекъснато да правят оценки на риска и да осигуряват защита.

## **Облачни технологии**

Облачните технологии (Cloud Native Technologies) представляват набор от решения и практики, проектирани да оптимизират изграждането, доставянето и управлението на приложения в облачна среда. Тези технологии се основават на контейнеризация, микросервизна архитектура, управление и автоматизация, което позволява на организациите да разработват и доставят софтуер бързо, ефективно и в голям мащаб.

* **Контейнеризация**

Контейнеризацията е ключов компонент на облачните технологии. Тя позволява на разработчиците да пакетират приложения и техните зависимости в контейнери, които могат лесно да бъдат доставяни и управлявани във всяка облачна среда. Docker и Kubernetes са две от най-популярните технологии в тази област, като Kubernetes служи като оркестратор на контейнери, управляващ мащабирането и доставянето на контейнеризирани приложения.

* **Архитектура на микроуслугите**

Микросервизната архитектура е метод, при който приложенията се разграждат на по-малки, независими компоненти (микроуслуги), които комуникират помежду си чрез добре дефинирани API-та. Този подход улеснява непрекъснатата интеграция и доставка (CI/CD), улеснява мащабирането и подобрява устойчивостта на приложенията.

* **Декларативно Управление**

Декларативното управление позволява на разработчиците да описват желаното състояние на системата чрез конфигурационни файлове, които се обработват автоматично за постигане на това състояние. Това намалява необходимостта от ръчно управление на ресурсите и улеснява автоматизацията и самовъзстановяването.

* **Автоматизация и непрекъсната доставка**

Облачните технологии акцентират върху автоматизацията на разработката, тестването, следенето и доставянето на приложения. Инструменти за непрекъсната интеграция и доставка (CI/CD) като Jenkins, GitLab CI и GitHub Actions позволяват на екипите да автоматизират своите разработъчни процеси и да доставят нови функции и корекции бързо и надеждно.

* **Облачни услуги и инфраструктура като код**

Облачните приложения често се ползват от множество облачни услуги като бази данни, достъпване и следене като услуга. Инфраструктурата като код (IaC) е практика, при която инфраструктурните ресурси се управляват чрез код, което улеснява автоматизацията и предоставя по-добър контрол и гъвкавост при управлението на облачните ресурси.

Облачните технологии променят начина, по който организациите разработват, доставят и управляват своите приложения, като предлагат мащабируемост, устойчивост и гъвкавост. Тези технологии са особено ценни в динамичната и разнообразна облачна среда, като позволяват на компаниите да използват напълно ползите от облачните ресурси.

## **Google Cloud (VPC, GKE, инстанция)**

Google Cloud Platform (GCP) предлага широка гама от услуги и инструменти за обработка на данни, хостинг на приложения, машинно обучение и много други, като същевременно осигурява сигурност, мащабируемост и гъвкавост. В рамките на GCP, Virtual Private Cloud (VPC), Google Kubernetes Engine (GKE) и Compute Engine Instances са трите ключови компоненти, които играят важна роля в разработването и доставянето на облачни решения [7].

* **Virtual Private Cloud (VPC)**

Virtual Private Cloud (VPC) в Google Cloud е изолирана мрежова среда в рамките на облака, която позволява на потребителите да управляват своето мрежово пространство в рамките на Google Cloud. VPC предоставя гъвкаво управление на IP адреси, мрежови подсегменти, маршрутизация и интернет връзки. С VPC, потребителите могат да конфигурират своята облачна мрежа така, че да отговаря на специфични изисквания за сигурност и изолация, като същевременно осигуряват връзка между ресурсите на Google Cloud и интернет или други облачни услуги [8].

* **Google Kubernetes Engine (GKE)**

Google Kubernetes Engine (GKE) е услуга, която позволява на разработчиците да доставят, управляват и мащабират контейнеризирани приложения с помощта на Kubernetes в облачните среди. GKE автоматизира редица сложни задачи за управление на контейнери, като например доставка на приложения, мащабиране, актуализации и следене. С GKE, потребителите могат да се възползват от автоматично мащабиране, висока наличност, сигурност на данните и интеграция с други услуги на Google Cloud.

* **Compute Engine Instances**

Compute Engine предоставя виртуални машини (VMs), които работят в инфраструктурата на Google Cloud. Тези инстанции позволяват на потребителите да изпълняват своите приложения на виртуални машини, които са конфигурирани според техните изисквания за изчислителна мощ, памет, дисково пространство и мрежови ресурси. Compute Engine предлага гъвкавост при избора на операционни системи, ядра и размери на инстанциите, като по този начин се адаптира към различни видове натоварвания и приложения. Освен това, Compute Engine поддържа автоматично мащабиране и балансиране на натоварването, което позволява на приложенията да управляват високи нива на трафик и да поддържат висока наличност.

## **Helm**

Helm е полезен инструмент за управление на пакети, специално разработен за Kubernetes, който улеснява инсталирането, конфигурирането и актуализирането на приложения в Kubernetes клъстери. Helm се разглежда като еквивалент на apt или yum в Linux системите, но за Kubernetes ресурси. Той използва пакетни файлове наречени "charts" за определяне на набор от ресурси, необходими за инсталиране на приложение или услуга в Kubernetes клъстер. Основни характеристики на Helm:

* **Charts:**

Helm пакетите се наричат charts. Chart е колекция от файлове, които описват свързан набор от ресурси на Kubernetes, необходими за изпълнението на приложение, услуга или друга функционалност. Тези файлове включват YAML конфигурационни файлове и шаблони, които Helm използва за генериране на специфични за Kubernetes манифести.

* **Хранилища (repository):**

Helm позволява на разработчиците да съхраняват и споделят charts чрез Helm репозитории. Това е подобно на други системи за управление на пакети, където потребителите могат да търсят, споделят и използват софтуерни пакети.

* **Управление на зависимостите:**

Helm улеснява управлението на зависимостите между charts. Можете да специфицирате зависимости във вашия chart, които трябва да бъдат инсталирани заедно с него, което гарантира, че всички необходими ресурси се разгръщат заедно в клъстера.

* **Шаблонизиране и конфигурация:**

Helm използва система за шаблонизиране, която позволява на потребителите да запазват charts при доставянето. Това означава, че един и същи chart може да бъде използван при доставката на приложения в различни среди (разработка, тестване, продукция) с различни конфигурации, като се използват само променливи при инсталирането.

* **Лесно актуализиране и възстановяване:**

Helm улеснява актуализирането и възстановяването на версии на приложения, които са били инсталирани с помощта на Helm charts. Това позволява на разработчиците и системните администратори да управляват версиите на своите приложения и услуги в Kubernetes клъстера по ефективен начин.

Helm има два основни компонента: клиент (helm) и сървър (tiller), като в Helm 3, tiller е премахнат за по-добра сигурност и по-проста архитектура, като всички операции се извършват от страна на клиента.

* **Клиент (Helm CLI):**

Клиентът е основното средство, което разработчиците използват за управление на charts, клъстери и инсталации. Той позволява на потребителите да търсят, инсталират, актуализират и управляват Helm charts в Kubernetes клъстер.

* **Charts:**

Charts са основният пакетен формат на Helm и съдържат всички необходими ресурси и конфигурационни файлове за разгръщане на приложение в Kubernetes клъстер.

* **Хранилища:**

Репозиториите съхраняват и разпространяват Helm charts. Те могат да бъдат публични или частни, като официалната на Helm предлага широк набор от готови за употреба charts.

Helm значително улеснява процеса на доставка и управление на приложения в Kubernetes клъстери, като предлага стандартизиран подход за пакетиране, споделяне и управление на Kubernetes ресурси.

## **Ingress Controller - NGINX**

Ingress Controller за Kubernetes, базиран на NGINX, е ключов компонент за управление на достъпа до приложения, работещи в Kubernetes клъстер. Той работи като обратен прокси и балансьор на натоварването, като улеснява маршрутизацията на външен трафик към подходящите услуги вътре в клъстера. NGINX Ingress Controller има големи възможности за конфигурация и оптимизация на NGINX, за да осигури ефективно разпределение на трафика и повишена производителност на приложенията. Основни характеристики [11]:

* **Маршрутизация на трафика:**

Ingress Controller управлява входящия трафик към приложенията в Kubernetes клъстера чрез маршрутизиране на заявките към подходящите бекенд услуги, базирани на хост и път в URL адреса.

* **Балансиране на натоварването:**

Осигурява балансиране на натоварването между множество инстанции на приложенията, подобрявайки по този начин отказоустойчивостта и производителността.

* **SSL/TLS терминиране:**

Поддържа терминиране на SSL/TLS връзки на входа на клъстера, което позволява шифриране на данните и сигурна комуникация.

* **Конфигурация и персонализация:**

NGINX Ingress Controller предлага гъвкави възможности за конфигурация, включително персонализирани правила за маршрутизация, ограничения на скоростта, управление на сесии и повече.

* **Следене и логове:**

Интеграция със системи за следене и логиране, което улеснява наблюдението на трафика и диагностицирането на проблеми.

NGINX Ingress Controller работи като под (pod) в Kubernetes клъстера, като слуша за промени в Ingress ресурси и автоматично прилага новите конфигурации в NGINX. Когато потребител или услуга извън клъстера прави заявка към приложение в клъстера, заявката първо достига до Ingress Controller. Той след това използва конфигурацията, определена в Ingress ресурсите, за да пренасочи заявката към подходящата услуга в клъстера.

NGINX Ingress Controller е ценен инструмент за управление на трафика в Kubernetes клъстери, който предлага гъвкавост, големи възможности за конфигурация и висока производителност. Той е идеален за организации, които търсят ефективно решение за маршрутизация и балансиране на натоварването в своите облачни среди.

## **Git**

Git е свободен и отворен софтуер за контрол на версиите. Той е създаден с цел подобряване на ефективността и надеждността на управлението на проекти с разпределено разработване. Git се отличава със своята бързина, гъвкавост и ефикасност при работа с малки до много големи проекти. Той позволява на разработчиците да записват промени във файлове и да създават "снимки" на техния код, които могат да бъдат запазени във времето, и към които може да се върнат, ако е необходимо. Ключови характеристики на Git:

* **Разпределена архитектура:**

Всяко работно копие на кода е пълноправно хранилище с пълна история на промените и пълни правомощия за управление на версиите. Това означава, че промените могат да бъдат направени и запазени локално, дори когато потребителят е офлайн.

* **Данни, устойчиви на повреди:**

Структурата на Git гарантира, че всяка промяна или снимка на кода се записва по начин, който намалява вероятността от повреда на данните или загуба на информация.

* **Бързина и ефикасност:**

Git е оптимизиран за производителност. Операциите за сливане, разклонение и превключване са изключително бързи, което улеснява управлението на различни версии на кода.

* **Поддържане на разклонения (branching) и сливания (merging):**

Git има инструменти за създаване и управление на разклонения и сливания, което позволява на екипите да работят паралелно по различни функции или проекти, а след това лесно да интегрират своята работа.

Основни концепции и термини:

* **Репозитории (repository):**

Съхранява историята на всички промени в кода и метаданни за проекта. Репозиториите могат да бъдат локални (на компютъра на разработчика) или отдалечени (например, на сървър или в облачна услуга като GitHub).

* **Комит (commit):**

Запис на промените в кода, който съдържа автора, датата и съобщение, което описва промяната. Комитите създават история на проекта, която може да бъде проследена.

* **Бранч (branch):**

Разклонение на кода, което позволява разработването на нови функции или корекции, без да се засяга основната кодова база. Разклоненията могат да бъдат сливани обратно в основния код, след като работата по тях е завършена.

* **Мърдж (merge):**

Процес на интегриране на промени от един бранч в друг. Съществува и автоматично сливане, но понякога сливанията изискват ръчно разрешаване на конфликти.

* **Таг (tag):**

Етикет, който се използва за маркиране на важни моменти в историята на кода, като например издания на продукта.

## **Web-application firewall**

Web Application Firewalls (WAF) са специализирани системи за сигурност, предназначени да защитават уеб приложенията и API-тата от различни видове атаки на ниво приложение. Те работят като защитна стена между уеб приложението и интернет, анализирайки входящия и изходящия трафик за откриване и блокиране на потенциално вредни заявки, които могат да компрометират сигурността на приложението или данните. WAF се инсталира на границата между уеб приложението и интернет, като може да бъде базиран на хардуер, софтуер, или предлаган като облачна услуга. Той анализира HTTP/HTTPS трафика, за да идентифицира и блокира злонамерени заявки, като използва набор от правила за сигурност, които определят какво се счита за легитимен трафик. Тези правила могат да бъдат базирани на шаблони на атаки, поведенчески анализ и други. Ключови характеристики на WAF:

* **Анализ на трафика:**

WAF анализира всяка заявка към уеб приложението и отговорите от него, използвайки предварително дефинирани или персонализирани правила за идентифициране на заплахи. Това включва атаки като Cross-Site Scripting (XSS), SQL Injection, Cross-Site Request Forgery (CSRF), и други уязвимости на уеб приложения.

* **Блокиране и предотвратяване на атаки:**

При идентифициране на подозрителна или вредна активност, WAF може да блокира заявката, преди тя да достигне уеб приложението, предотвратявайки потенциалното ѝ въздействие.

* **Персонализирани правила и политики за сигурност:**

Администраторите могат да конфигурират и адаптират правилата на WAF според специфичните нужди на приложението.

* **Защита от DDoS атаки:**

Много WAF решения предлагат защита от Distributed Denial of Service (DDoS) атаки, които целят да претоварят уеб приложението с трафик до степен на неговото отказване.

## **NGINX ModSecurity, Google Cloud WAF solutions**

NGINX ModSecurity и Google Cloud WAF са две решения за защита на уеб приложения и API-та срещу широк спектър от уеб базирани атаки.

* **NGINX ModSecurity**

ModSecurity, първоначално разработен като модул за Apache HTTP Server, е популярен open-source Web Application Firewall (WAF), който предлага разнообразие от функции за сигурност за уеб приложения. NGINX версията на ModSecurity позволява интеграцията на тези сигурностни функции в NGINX сървъра, който е известен със своята производителност и ефективност. OWASP Core Rule Set (CRS) е набор от правила за защита, които се използват като добавка към уеб приложни защитни стени (WAFs), за да предпазят уеб приложенията от различни атаки. Тези правила са разработени от Open Web Application Security Project (OWASP), която е международна организация, посветена на подобряването на сигурността на софтуера [9]. Основни характеристики на NGINX ModSecurity:

* Правила за филтриране: Използва сложни правила за филтриране, които могат да бъдат конфигурирани и адаптирани според специфичните нужди на приложението. Тези правила помагат за предпазването от атаки като SQL инжекции, Cross-Site Scripting (XSS) и много други.
* Гъвкавост и конфигурируемост: ModSecurity предлага висока степен на гъвкавост и позволява детайлно настройване на политиките за сигурност, което го прави подходящ за различни среди и приложения.
* Логване и следене: Предоставя подробни логове на трафика, които могат да бъдат използвани за анализ на атаки и потенциални уязвимости в сигурността на приложението.
* **Google Cloud WAF Solutions**

Google Cloud предлага интегрирани WAF решения, проектирани да защитават приложенията и данните, работещи в Google Cloud Platform (GCP), срещу различни уеб атаки и заплахи. Cloud Armor е основният продукт на Google Cloud, който предлага WAF функционалност, заедно с други услуги за защита. Основни характеристики на Google Cloud WAF (Cloud Armor):

* Предварително дефинирани и персонализирани правила: Cloud Armor предоставя набор от предварително дефинирани правила за защита срещу често срещани уеб атаки и позволява създаването на персонализирани правила, за да се отговори на специфичните нужди на приложението.
* Интеграция с Google Cloud Infrastructure: Тясната интеграция с инфраструктурата на Google Cloud означава, че Cloud Armor може лесно да защитава приложения, работещи в средите на Google Compute Engine, Google Kubernetes Engine, и други услуги на GCP.
* Защита от DDoS атаки: В допълнение към функциите на WAF, Cloud Armor предлага и защита срещу DDoS атаки, като използва глобалната инфраструктура на Google за разпределяне и намаляване на вредния трафик.
* Адаптивна защита с помощта на машинно обучение: Cloud Armor използва машинно обучение за анализ на трафика и идентифициране на нови заплахи, което позволява адаптивна защита срещу сложни и еволюиращи атаки.

И NGINX ModSecurity, и Google Cloud WAF решенията предлагат силна защита за уеб приложения и API-та срещу широк спектър от уеб атаки. Изборът между тези две решения зависи от конкретните нужди на организацията, технологичната среда, и предпочитанията към интеграция и управление на инфраструктурата. NGINX ModSecurity е подходящ за организации, които търсят гъвкаво и конфигурируемо решение с отворен код, докато Google Cloud WAF е идеален за тези, които вече използват GCP и искат тясно интегрирано решение за сигурност с допълнителни възможности като защита от DDoS и адаптивна защита.

## **Python**

Python е високо ниво, интерпретируем език за програмиране, който е създаден от Гуидо ван Росум. С дизайна си, който насърчава ясен и логичен код, Python е един от най-популярните програмни езици в света. Той поддържа множество парадигми за програмиране, включително обектно-ориентирано, процедурно и функционално програмиране, което го прави изключително гъвкав и приложим в широк спектър от разработки. Основни характеристики на Python:

* **Лесен за учене и четене:**

Python се характеризира със синтаксис, който е лесен за разбиране и писане, което улеснява процеса на учене и намалява времето за разработка на програми.

* **Вградена библиотека:**

Python идва с обширна стандартна библиотека, която включва модули за различни проекти, като работа с файлове, уеб разработка, математически операции и много други.

* **Поддръжка на множество парадигми:**

Поддържа различни стилове на програмиране, включително обектно-ориентирано, императивно, функционално и процедурно програмиране.

* **Интерпретируем и интерактивен:**

Python е интерпретируем език, което означава, че кодът се изпълнява директно, без нужда от компилиране. Това позволява бърза разработка и тестване на кода.

* **Платформена независимост:**

Python програми могат да се изпълняват на различни операционни системи без промени в кода, благодарение на платформената си независимост.

* **Богата екосистема:**

Съществуват много библиотеки и фреймуърки, разработени за Python, които покриват области като уеб разработка (например Django), научни изчисления (например NumPy, SciPy, Pandas), машинно обучение (например TensorFlow, PyTorch) и други.

Python се използва в множество области на програмирането и разработката на софтуер, включително:

* **Уеб разработка:**

Чрез фреймуърки като Django и Flask, Python е широко използван за създаването на мощни уеб приложения.

* **Научни и математически изчисления:**

С библиотеки като NumPy, SciPy и Matplotlib, Python е предпочитан за научни изследвания и обработка на данни.

* **Автоматизация на задачи и скриптове:**

Python е популярен инструмент за автоматизиране на рутинни задачи, обработка на файлове и системно администриране.

* **Машинно обучение и изкуствен интелект:**

Благодарение на библиотеки като TensorFlow, Keras и PyTorch, Python е в основата на много проекти за машинно обучение и изкуствен интелект.

Python продължава да расте като един от най-предпочитаните езици за програмиране, благодарение на своята гъвкавост, лесно обучение и екосистема, което го прави подходящ за широк спектър от програмни и разработчески задачи.

## **Django**

Django е високо ниво, отворен код, Python базиран уеб фреймуърк, който следва модела на архитектурата "Model-View-Template" (MVT). Създаден с идеята да улесни бързата разработка на сложни, базирани на данни уеб приложения, Django предлага подход, който включва множество готови за употреба компоненти за често срещани задачи в уеб разработката. Този подход позволява на разработчиците да се съсредоточат върху създаването на своите приложения, без да трябва да измислят решения за типични проблеми като управление на потребители, сесии, форми, защита от атаки и други. Django предлага бърз и ефективен начин за разработването на сложни уеб приложения с акцент на повторното използване на код, намаляване на броя на грешките и ускоряване на процеса на разработка. Благодарение на своята гъвкавост и обширна документация, Django е предпочитан избор за много разработчици по света, от стартиращи компании до големи корпорации. Основни характеристики на Django [12]:

* **ORM (Object-Relational Mapping):**

Django включва ORM, който позволява на разработчиците да определят структурата на базата данни чрез Python класове. Това улеснява извършването на сложни заявки към базата данни без да се налага да пишат SQL код.

* **Автоматичен административен интерфейс:**

Django автоматично генерира административен интерфейс за уеб приложенията, което прави управлението на съдържанието лесно дори за нетехнически потребители.

* **Сигурност:**

Фреймуъркът предлага вградена защита срещу много често срещани уеб атаки, като например SQL инжекции, cross-site scripting (XSS), cross-site request forgery (CSRF) и др., като по този начин помага за създаването на по-сигурни приложения.

* **Разширяемост:**

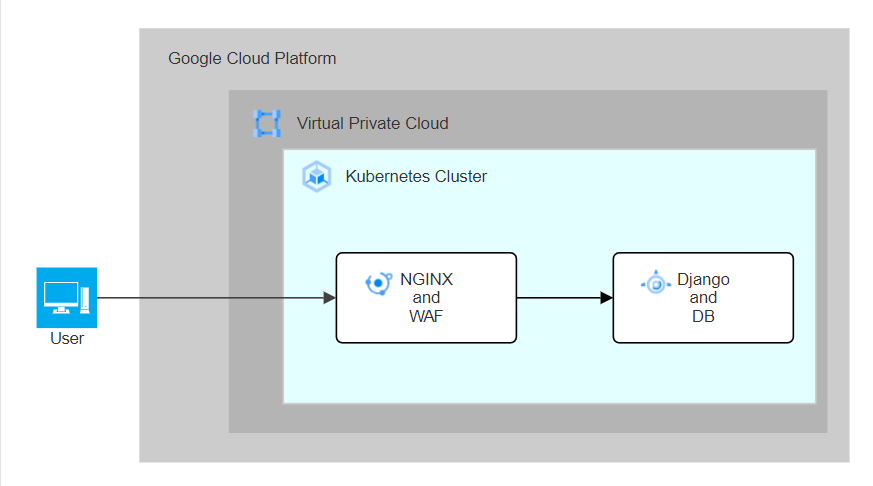
Django е изключително разширяем, позволявайки на разработчиците да добавят допълнителни функционалности чрез "приложения". Това може да бъде или използване на съществуващи приложения, или създаване на собствени.

* **Пълна документация:**

Django е известен със своята обширна и ясна документация, което улеснява процеса на обучение и разработка.

# **Втора глава - Функционални изисквания**

## **Блокова схема и принцип на работа**



*Фиг. 2.1 Блокова схема на начин на работа*

Когато потребител изпраща заявка към сайта, заявката влиза в GCP VPC, където е разположен Kubernetes клъстерът. Nginx управлява входящия трафик към контейнерите, а WAF филтрира заявките за вреден трафик. Приложението обработва заявката с взаимодействие с базата данни. Отговорът се изпраща обратно през Nginx и WAF, излиза от VPC и се връща при потребителя.

## **Функционални изисквания за създаване на уеб приложение на Python със софтуерната рамка Django.**

Уеб приложението представлява онлайн магазин, базирано на софтуерната рамка Django. Функционалните изисквания за създаването на уеб приложение, включват различни аспекти, като създаване на потребителски акаунти, управление на сесии за вход в системата, работа с база данни за съхранение на информация за потребителите, както и функции за навигация и пазаруване в самия магазин.

* **Създаване на потребителски акаунти**

Потребителите трябва да могат да създават акаунт, като попълват формуляр със своите данни, като име и парола. При въвеждане на парола, тя трябва да бъде повторена, с цел сигурност на потребителя за правилно въвеждане. Данните трябва да бъдат валидирани за да отговарят на определени критерии (например, сила на паролата, брой символи, наличие на числа, главни букви и малки букви). Данните за създаден потребител се запазват в база данни (SQLite). Информацията в базата данни се съхранява в криптиран вид, с цел защита.

* **Вход в системата**

Потребителите трябва да влизат в системата като използват въведените данни при създаване на акаунт (име и парола). Данните се проверяват със съхранената информация в базата данни. След успешен вход, се създава сесия за потребителя, което позволява навигация в онлайн магазина без нужда от повторно въвеждане на данни за вход.

* **Функционалности на магазина**

Магазинът трябва да има следните страници: Начална страница, страница, в която са описани продуктите, “За нас” страница, страници за създаване на потребител, вход и за преглед на профила.

## **Функционални изисквания за изграждане на контейнер имидж на уеб приложението**

Функционалните изисквания за изграждането на контейнер имидж на уеб приложение, разработено с Python и Django, определят ключови аспекти относно създаването, конфигурацията и доставката на приложението в контейнерна среда, като Docker. Основните функционални изисквания:

* **Създаване на Dockerfile:**
* Определение на базово изображение: Избор на подходящо базово изображение от Docker Hub, като например python:3.9-alpine, който да съответства на версията на Python, използвана за разработката на приложението.
* Инсталиране на Зависимости: Дефиниране на стъпка в Dockerfile за инсталиране на всички необходими библиотеки и зависимости, изброени в requirements.txt на проекта.
* Копиране на Проектни Файлове: Копиране на файловете на проекта в контейнера, за да може приложението да се изпълнява в изолирана среда.
* Експониране на Портове: Дефиниране на порта, на който уеб приложението слуша (например 8000), за да може той да бъде достъпен извън контейнера.
* **Изграждане и оптимизация на изображението:**
* Минимизиране на размера на Изображението: Използване на техники и практики за намаляване на размера на крайния имидж, като използване на подходящо базово изображение и избягване на ненужни файлове.
* Тагване и Версиониране: Поддържане на добра практика за тагване и версиониране на създадените изображения, за да се улесни управлението на различните версии на приложението.

Тези функционални изисквания осигуряват основата за създаването на надеждно, сигурно и лесно за поддръжка уеб приложение, доставено в контейнерна среда.

## **Функционални изисквания за създаване на Kubernetes клъстер в Google Cloud Platform**

Функционалните изисквания за създаването на Kubernetes клъстер в Google Cloud Platform (GCP) обхващат подробен план за конфигурация, управление и следене на ресурсите, необходими за доставянето и поддръжката на високодостъпни, сигурни и мащабируеми приложения. Етапи:

* **Инициализиране на клъстера:**
* Избор на Регион и Зона: Определяне на подходящ регион и зона в GCP, където да се достави клъстерът, вземайки предвид фактори като латентност, цени и наличност на ресурси.
* Конфигурация на Клъстера: Задаване на размера на клъстера, броя на нодовете, типа на машините (например, стандартни, високопроизводителни, оптимизирани за памет), както и версията на Kubernetes.
* Автоматично Мащабиране: Конфигуриране на автоматичното мащабиране на нодовете, за да се осигури автоматично добавяне или премахване на ресурси в зависимост от натоварването.
* **Конфигуриране на Мрежата:**
* VPC (Virtual Private Cloud): Създаване на изолирана виртуална мрежа за клъстера, за да се управлява комуникацията между нодовете и достъпът до интернет.
* Firewall Правила: Настройка на правила за мрежовата защитна стена, за да се контролира достъпът до и от клъстера.

Изграждането на Kubernetes клъстер в Google Cloud Platform изисква внимателно планиране и конфигурация, за да се гарантира, че системата е сигурна, мащабируема и устойчива на отказ.

## **Функционални изисквания за създаване на необходимите Kubernetes ресурси, за постигане работоспособност на приложението и поставяне на разпределител на мрежовия трафик пред приложението в клъстера.**

Функционалните изисквания за създаване на необходимите Kubernetes ресурси, за да се постигне работоспособност на приложението и да се постави разпределител на мрежовия трафик в клъстера, обхващат серия от конфигурации и настройки. Тези ресурси са важни за управлението, разпределението и мащабирането на приложения в Kubernetes среда.

* **Deployment:**

Целта на този конфигурационен манифест е да дефинира желаното състояние на приложението, включително използваните контейнерни образи и броя на репликите, които трябва да бъдат изпълнени в клъстера. Изисквания:

* Определение на контейнерни изображения: Специфициране на Docker образите, които ще се използват за създаване на контейнерите в подовете.
* Конфигурация на реплики: Дефиниране на броя реплики от приложението, които да бъдат поддържани в работно състояние.
* Определяне на порт: Дефинирането на порт е важен аспект от конфигурацията на Kubernetes Deployment, защото определя как комуникацията с контейнерите в подовете ще бъде управлявана.
* **Service:**

Използва се да осигури стабилен и непрекъснат достъп до подовете, чрез дефиниране на абстрактен слой, който да управлява достъпа до приложението. Изисквания:

* Тип на сървиса: Определяне на типа на сървиса (например Service), в зависимост от нуждите за достъп до приложението.
* Селектори на подове: Дефиниране на селектори, които съответстват на етикетите на подовете, за да се управлява трафикът към тях.
* Портове: Конфигуриране на портовете, през които сървисът ще бъде достъпен.
* **Ingress:**

Управлява външния достъп до услугите в клъстера, като се използва NGINX като прокси сървър и разпределител на мрежовия трафик. Изисквания:

* Конфигурация на правила: Дефиниране на правила за маршрутизация, които съответстват на заявки от външни адреси към конкретни сървици в клъстера.
* SSL/TLS Сертификати: Интегриране на SSL/TLS сертификати за криптиране на трафика и осигуряване на сигурен достъп.
* Анотации за NGINX: Използване на специфични за NGINX анотации за допълнителна конфигурация на поведението на ингрес контролера.

Създаването на deployment, service и ingress ресурси в Kubernetes е фундаментален процес, който позволява декларативно описване и управление на приложенията. Тези файлове осигуряват гъвкавост и мощни инструменти за мащабиране, самовъзстановяване и управление на трафика, което е критично за поддържане на висока достъпност и надеждност на приложенията в съвременни микросървисни архитектури.

## **Функционални изисквания за подсигуряване композитна защита на приложението**

Комплексната защита на приложенията в съвременната киберсигурностна среда изисква интегриран подход, който обединява няколко слоя на защита, за да се минимизират потенциалните рискове и да се осигури устойчивост срещу атаки. В контекста на приложения, работещи в контейнеризирана среда с използване на NGINX като прокси сървър и OWASP ModSecurity Web Application Firewall (WAF) за уеб-базирана защита, е важно да се вземат предвид функционални изисквания за подсигуряване комплексна защита на приложението. Тези изисквания обхващат политики за мрежови достъп, политики за сигурност на ниво контейнер, уеб-базирана защитна стена и приложение за контрол на пароли.

* **Политики за Сигурност на Ниво Контейнер [5]**
* Ограничаване на Правата на Потребителите: Изпълнение на контейнерите с непривилегировани потребители (например runAsUser: 1000) и забрана за ескалация на привилегии (allowPrivilegeEscalation: false).
* Защита на Файловата Система: Задаване на файловата система на контейнера да бъде само за четене (readOnlyRootFilesystem: true), за да се предотврати несанкционирано модифициране на файлове и системни настройки.
* Управление на файлови пространства: Дефиниране на безопасни файлови пространства (emptyDir: {}) за временно съхранение, което ограничава риска от постоянни модификации на файловата система.
* Отказване на всички видове capabilities -
* **Уеб-Базирана Защитна Стена (WAF)**
* Интеграция с OWASP ModSecurity: Активиране и конфигуриране на ModSecurity WAF с актуални правила от OWASP Core Rule Set (CRS) за защита срещу уеб атаки като SQL injection, CSRF и други [9].

Този интегриран подход към сигурността на приложенията не само увеличава защитата срещу различни видове кибер атаки, но и улеснява съответствието с регулаторни стандарти за сигурност. Правилната имплементация и непрекъснатото поддържане на тези компоненти са ключови за защитата на съвременните приложения в динамично променящата се заплахова среда.

# **Трета глава - Реализация на защита на контейнеризирани приложения в облачни среди, базирани на Kubernetes**

Целта на дипломната работа е да се създаде примерно уеб приложение, което да работи в облачни среди и да бъде защитено от известни видове атаки. Показва се как с помощта на съвременни технологии и практики може да се създаде, качи и защити уеб приложение в облачна среда, като се осигури неговата сигурност, скалируемост и висока наличност.

## **Създаване на уеб приложение на Python със софтуерната рамка Django.**

За създаването на Django проект, първо е нужно да се инсталира софтуерната рамка Django. Това се случва чрез командата pip, изпълнена в терминал.



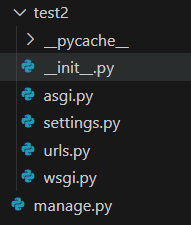
*Фиг. 3.1 Команда за инсталиране на Django*

След инсталирането, е създаден проект с командата django-admin startproject test2 . .



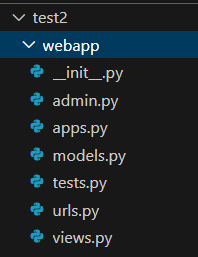
*Фиг. 3.2 Команда за създаване на проект*

Тази команда създава следните файлове, показани на Фиг. 3.3



*Фиг. 3.3 Създадени файлове след командата django-admin startproject test2 .*

След създаването на проект, е създадено и приложение в проекта с командата python manage.py startapp webapp. Създадените файлове, след изпълнението на тази команда са показани на Фиг. 3.4



*Фиг. 3.4 Създадените файлове, след изпълнението на python manage.py startapp webapp*

Описание на файлове, намиращи се в проекта test2:

* **asgi.py:**

| import os from django.core.asgi import get\_asgi\_application os.environ.setdefault('DJANGO\_SETTINGS\_MODULE', 'test2.settings') application = get\_asgi\_application() |
| --- |

*Фиг. 3.5 asgi.py*

* import os: Добавя модула os, който предоставя функционалност за взаимодействие с операционната система.
* from django.core.asgi import get\_asgi\_application: Добавя функцията get\_asgi\_application, която връща ASGI приложение на Django за текущата конфигурация.
* os.environ.setdefault('DJANGO\_SETTINGS\_MODULE', 'test2.settings'): Задава променливата на средата DJANGO\_SETTINGS\_MODULE на 'test2.settings', ако тя още не е зададена. Това указва къде се намират настройките на проекта.
* application = get\_asgi\_application(): Създава ASGI приложение, което сървърите могат да използват за обработка на заявки.
* **settings.py:**

| import os from pathlib import Path BASE\_DIR = Path(\_\_file\_\_).resolve().parent.parent SECRET\_KEY = 'django-insecure-r+ku!(0f!lp&b@7z\_d32lqi^^$%nqz+gi(qkl2vmhktux&n7c7' DEBUG = True ALLOWED\_HOSTS = ["\*"] INSTALLED\_APPS = [  'django.contrib.admin',  'django.contrib.auth',  'django.contrib.contenttypes',  'django.contrib.sessions',  'django.contrib.messages',  'django.contrib.staticfiles',  'webapp' ] MIDDLEWARE = [  'django.middleware.security.SecurityMiddleware',  'django.contrib.sessions.middleware.SessionMiddleware',  'django.middleware.common.CommonMiddleware',  'django.middleware.csrf.CsrfViewMiddleware',  'django.contrib.auth.middleware.AuthenticationMiddleware',  'django.contrib.messages.middleware.MessageMiddleware',  'django.middleware.clickjacking.XFrameOptionsMiddleware', ] ROOT\_URLCONF = 'test2.urls'  TEMPLATES = [  {  'BACKEND': 'django.template.backends.django.DjangoTemplates',  'DIRS': [BASE\_DIR / "/templates"],  'APP\_DIRS': True,  'OPTIONS': {  'context\_processors': [  'django.template.context\_processors.debug',  'django.template.context\_processors.request',  'django.contrib.auth.context\_processors.auth',  'django.contrib.messages.context\_processors.messages',  ],  },  }, ] WSGI\_APPLICATION = 'test2.wsgi.application' DATABASES = {  "default": {  "ENGINE": "django.db.backends.sqlite3",  "NAME": "db",  } } AUTH\_PASSWORD\_VALIDATORS = [  {  'NAME': 'django.contrib.auth.password\_validation.UserAttributeSimilarityValidator',  },  {  'NAME': 'django.contrib.auth.password\_validation.MinimumLengthValidator',  },  {  'NAME': 'django.contrib.auth.password\_validation.CommonPasswordValidator',  },  {  'NAME': 'django.contrib.auth.password\_validation.NumericPasswordValidator',  }, ] LANGUAGE\_CODE = 'en-us' TIME\_ZONE = 'UTC' USE\_I18N = True USE\_TZ = True STATIC\_URL = 'templates/' DEFAULT\_AUTO\_FIELD = 'django.db.models.BigAutoField' MEDIA\_URL = '/static/' MEDIA\_ROOT = os.path.join(BASE\_DIR, 'static') |
| --- |

*Фиг. 3.6 settings.py*

Този файл съдържа конфигурациите на Django приложението.

* import os, from pathlib import Path: Добавя необходимите модули за работа с операционната система и файлови пътища.
* BASE\_DIR: Дефинира базовата директория на проекта.
* SECRET\_KEY: Ключ, използван за криптографски подписи.
* DEBUG: Режим на отстраняване на грешки.
* ALLOWED\_HOSTS: Списък на хостовете/домейните, на които е позволено да се обслужва сайта.
* INSTALLED\_APPS: Списък на инсталираните приложения, които Django ще използва.
* MIDDLEWARE: Списък на мидълуерите, които се изпълняват при всяка заявка към приложението.
* ROOT\_URLCONF: Пътят към модула urls.py на проекта.
* TEMPLATES: Конфигурация на системата за шаблони.
* WSGI\_APPLICATION: Пътят към WSGI приложението на проекта.
* DATABASES: Конфигурация на базата данни. В случая е използвана SQLite
* AUTH\_PASSWORD\_VALIDATORS: Конфигурация за валидиране на паролите.
* LANGUAGE\_CODE, TIME\_ZONE, USE\_I18N, USE\_TZ: Локализация и часови зони.
* STATIC\_URL, MEDIA\_URL, MEDIA\_ROOT: Конфигурации за статични и медийни файлове.
* **urls.py:**

| from django.contrib import admin from django.urls import include, path from django.conf import settings from django.conf.urls.static import static urlpatterns = [  path('', include('webapp.urls')),  path('about/', include('webapp.urls')),  path('items/', include('webapp.urls')),  path('signin/', include('webapp.urls')),  path('signout/', include('webapp.urls')),  path('signup/', include('webapp.urls')),  path('profile/', include('webapp.urls')),  path('admin/', admin.site.urls), ] + static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT) |
| --- |

*Фиг. 3.7 urls.py*

Конфигурация на URL адресите за проекта.

* from django.contrib import admin: Добавя административния модул на Django.
* from django.urls import include, path: Добавя функции за дефиниране на URL пътища.
* from django.conf import settings: Добавя настройките на проекта.
* from django.conf.urls.static import static: Добавя функция за обслужване на статични и медийни файлове в разработчески режим.
* urlpatterns: Списък на конфигурациите на web страници за проекта, който включва пътища към админ панела и приложението webapp.
* **wsgi.py:**

| import os from django.core.wsgi import get\_wsgi\_application os.environ.setdefault('DJANGO\_SETTINGS\_MODULE', 'test2.settings') application = get\_wsgi\_application() |
| --- |

*Фиг. 3.8 wsgi.py*

* Сходен на asgi.py, но за WSGI (Web Server Gateway Interface) – стандарт за Python приложения в уеб, вместо ASGI, който е асинхронен.

Файлове в приложение webapp:

* **admin.py:**

| from django.contrib import admin from .models import Product admin.site.register(Product) |
| --- |

*Фиг. 3.9 admin.py*

Регистрира моделите в админ панела на Django, за да могат да бъдат управлявани през нея.

* from django.contrib import admin: Добавя модула admin от django.contrib. Този модул е необходим за работата с административния интерфейс на Django.
* from .models import Product: Добавя модела Product от модула models.py, който е в същия пакет (означено с точка преди models). Това позволява на admin.py да регистрира модела Product, за да може да бъде управляван през административния интерфейс на Django.
* admin.site.register(Product): Регистрира модела Product в административния сайт на Django, което позволява на модела да бъде редактиран през административния интерфейс.
* **apps.py:**

| from django.apps import AppConfig class WebappConfig(AppConfig):  default\_auto\_field = 'django.db.models.BigAutoField'  name = 'webapp' |
| --- |

*Фиг. 3.10 apps.py*

Дефинира конфигурацията на приложението webapp.

* from django.apps import AppConfig: Импортира класа AppConfig от модула django.apps. AppConfig се използва за конфигуриране на някои аспекти на приложението.
* class WebappConfig(AppConfig): Дефинира конфигурационен клас за приложението, наследяващ AppConfig. Този клас описва конфигурацията на приложението webapp.
* default\_auto\_field = 'django.db.models.BigAutoField': Задава типа на автоматично генерираните полета за ID като BigAutoField, което е подходящо за приложения, очакващи голям брой записи.
* name = 'webapp': Указва името на приложението, което е webapp. Това име се използва от Django за идентификация на приложението в рамките на проекта.
* **models.py:**

| from django.db import models  class *Product*(models.*Model*):  name = models.*CharField*(max\_length=100)  description = models.*TextField*()  image = models.*ImageField*(upload\_to='static/') class *User*(models.*Model*):  username = models.*CharField*(max\_length=100)  password = models.*CharField*(max\_length=50) |
| --- |

*Фиг. 3.11 models.py*

Описва моделите (структурите на данни), които се използват в приложението. Дефинира модели за Product и User.

* from django.db import models: Импортира модула models от django.db, който съдържа класове и функции за работа с базата данни.
* class Product(models.Model): Дефинира модел Product, който наследява models.Model. Това означава, че Product е модел на Django, представляващ таблица в базата данни.
* name = models.CharField(max\_length=100): Дефинира текстово поле с максимална дължина 100 символа за името на потребителя.
* description = models.TextField(): Дефинира текстово поле за описание на продукта без ограничение за дължината.
* image = models.ImageField(upload\_to='static/'): Дефинира поле за изображение, където файловете се качват в директория static/.
* class User(models.Model): Дефинира допълнителен модел User, който също наследява от models.Model и представлява потребител с полета за потребителско име, и парола.
* **urls.py:**

| from django.urls import path from webapp import views from django.contrib import admin urlpatterns = [  path('', views.home, name='home'),  path('about/', views.about, name='about'),  path('items/', views.items, name='items'),  path('signin/',views.signin, name='signin'),  path('signout/',views.signout, name='signout'),  path('signup/',views.signup, name='signup'),  path('profile/',views.profile, name='profile'), ] |
| --- |

*Фиг. 3.12 urls.py*

Определя URL пътищата, специфични за приложението webapp.

* from django.urls import path: Импортира функцията path от модула django.urls, която се използва за дефиниране на URL пътища в приложението.
* from webapp import views: Импортира модула views от приложението webapp. Този модул съдържа функции за обработка на заявки (views), които се асоциират с URL пътища.
* from django.contrib import admin: Импортира модула admin, за да може да се използва административният интерфейс на Django.
* urlpatterns = [...]: Дефинира списък от URL пътища (urlpatterns) за приложението. Всеки път се асоциира с функция от views.py, която да обработва съответната заявка.
* **views.py:**

| from django.shortcuts import render from django.template import loader from django.http import HttpResponse from django.views.generic import TemplateView from django.shortcuts import render, redirect from django.contrib.auth.forms import UserCreationForm from django.contrib.auth import authenticate, login from django.contrib.auth.forms import AuthenticationForm from django.contrib.auth import logout from django.shortcuts import HttpResponseRedirect   def signup(request):  if request.user.is\_authenticated:  return redirect('/')  if request.method == 'POST':  form = UserCreationForm(request.POST)  if form.is\_valid():  form.save()  username = form.cleaned\_data.get('username')  password = form.cleaned\_data.get('password1')  user = authenticate(username=username, password=password)  login(request, user)  return redirect('/')  else:  return render(request, 'signup.html', {'form': form})  else:  form = UserCreationForm()  return render(request, 'signup.html', {'form': form}) def home(request):  return render(request, 'home.html')    def signin(request):  if request.user.is\_authenticated:  return render(request, 'home.html')  if request.method == 'POST':  username = request.POST['username']  password = request.POST['password']  user = authenticate(request, username=username, password=password)  if user is not None:  login(request, user)  return redirect('/profile') #profile  else:  msg = 'Error Login'  form = AuthenticationForm(request.POST)  return render(request, 'login.html', {'form': form, 'msg': msg})  else:  form = AuthenticationForm()  return render(request, 'login.html', {'form': form}) def profile(request):  return render(request, 'profile.html') def signout(request):  logout(request)  return redirect('/') def home(request):  return render(request, "home.html")  def about(request):  return render(request, "about.html") def items(request):  return render(request, "items.html") |
| --- |

*Фиг. 3.13 views.py*

“views.py” обработва различни HTTP заявки (GET, POST) от потребителя и връща подходящ отговор. Всеки изглед (view) обикновено кореспондира на определено действие или страница в приложението. Например, функцията signup обработва регистрацията на нови потребители, докато signin управлява процеса на вход в системата. Функциите home, about, items, и profile показват различни страници на уебсайта, като използват шаблони за да създадат HTML съдържание, което се изпраща обратно към потребителя.

* from django.shortcuts import render, redirect: Импортира функциите render и redirect. render се използва за създаване на HTML страници, като съчетава шаблон с контекст (данни), докато redirect пренасочва потребителя към друг URL.
* from django.template import loader: Импортира loader, който се използва за зареждане на шаблони.
* from django.http import HttpResponse: Импортира класа HttpResponse за създаване на отговори към HTTP заявки.
* from django.views.generic import TemplateView: Импортира TemplateView за създаване на прости класове базирани на изгледи, които служат предимно статично съдържание.
* from django.contrib.auth.forms import UserCreationForm, AuthenticationForm: Импортира формите UserCreationForm за регистрация на нови потребители и AuthenticationForm за вход на съществуващи потребители.
* from django.contrib.auth import authenticate, login, logout: Импортира функциите authenticate, login, и logout за управление на потребителски сесии.
* def signup(request): Дефинира функция signup, която приема request като аргумент. Тази функция се използва за регистрация на нови потребители.
* if request.user.is\_authenticated: return redirect('/'): Проверява дали потребителят, извършващ заявката, вече е автентикиран. Ако е така, функцията пренасочва потребителя към началната страница (/) чрез redirect.
* if request.method == 'POST': Проверява дали методът на заявката е POST. Това обикновено означава, че потребителят е изпратил данни чрез формата за регистрация.
* form = UserCreationForm(request.POST): Създава инстанция на UserCreationForm, като подава данните от заявката (request.POST) към формата. Тази форма е предоставена от Django и се използва за създаване на нови потребители.
* if form.is\_valid(): Проверява дали формата е валидна. Това включва валидация на потребителското име и паролата, според правилата, дефинирани в UserCreationForm.
* form.save(): Ако формата е валидна, запазва новия потребител в базата данни.
* username = form.cleaned\_data.get('username'); password = form.cleaned\_data.get('password1')

Извличат потребителското име и паролата от обработените и валидирани данни на формата.

* user = authenticate(username=username, password=password): Използва функцията authenticate за да автентикира новосъздадения потребител с въведените потребителско име и парола.
* login(request, user): Ако автентикацията е успешна, използва функцията login за да влезе потребителят в системата (създава потребителска сесия).
* return redirect('/'): Пренасочва потребителя към началната страница след успешна регистрация и вход.
* else: return render(request, 'signup.html', {'form': form}): Ако формата не е валидна или методът на заявката не е POST, показва страницата за регистрация отново, като предава текущата инстанция на формата към шаблона signup.html. Това позволява на потребителя да види грешките във формата и да опита отново.

Файлове за визуализация на отделните страници на уеб приложението, намиращи се в папка webapp/templates:

* **base.html**

| <!DOCTYPE html> <html lang="en"> <head> <title>  {% block title %}Stefi's art{% endblock %}</title>  <style>  header {  background-color: #333;  color: #fff;  padding: 10px;  text-align: center; }  footer {  background-color: #333;  color: #fff;  padding: 10px;  text-align: center;  } </style>  {% load static %}  <link rel="stylesheet" type="text/css" href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap.min.css"/>  <meta charset="UTF-8" name="viewport" content="width=device-width, initial-scale=1"/> </head> <body>  <div class="container">  <header>  <h1>Stefi's art</h1>  </header>  <hr>  <main>  {% block content %}  {% endblock %}  </main>  <hr>  </div>  <footer>  <p>&copy;All rights reserved</p> </footer> </body> </html> |
| --- |

*Фиг. 3.14 base.html*

Файлът base.html служи като основен шаблон за другите HTML страници на уебсайта. Този шаблон определя общата структура на страницата, включително заглавието (<title>), стиловете (<style>), и основните HTML елементи като <header>, <main>, и <footer>. Всеки от тези елементи е проектиран да бъде преизползваем в различни страници на уебсайта, което улеснява разработката и поддръжката на сайта. Файлът използва Django шаблонни тагове ({% block title %}, {% endblock %} и {% block content %}), които позволяват на другите страници, които наследяват base.html, да персонализират съдържанието в тези блокове. Например, страници могат да предоставят свои собствени заглавия и основно съдържание, което ще бъде вмъкнато в основния шаблон на местата, определени от блоковете.

* **index.html**

| {% extends 'base.html' %}  {% block title %}Login{% endblock %}  {% block content %}  <h2>Welcome!</h2>  {% if user.is\_authenticated %}  Hi {{ user.username }}!  <a href="/signout" class="btn btn-danger">Logout</a>  {% else %}  <a href="/signin" class="btn btn-primary"><span class="glyphicon glyphicon-log-in"></span>Login</a>  <a href="/signup" class="btn btn-success">Signup</a>  {% endif %} {% endblock %} |
| --- |

*Фиг. 3.15 index.html*

index.html е страница, която наследява основния шаблон base.html. Тази страница персонализира блока за заглавие ({% block title %}) със своето собствено заглавие ("Login") и дефинира основното съдържание ({% block content %}) на страницата. Съдържанието включва приветствие и условие, което проверява дали потребителят е автентикиран ({% if user.is\_authenticated %}). В зависимост от това, автентикираният потребител е поздравен с името си и има възможност да излезе от акаунта си, докато неавтентикираният потребител вижда опции за вход и регистрация.

* **About.html**

| <!DOCTYPE html> <html> <head>  <title>About us</title>  <style>  body {  font-family: Arial, sans-serif;  margin: 0;  padding: 0;  background-color: #f0f0f0;  }  header {  background-color: #333;  color: #fff;  padding: 10px;  text-align: center;  }  nav ul {  list-style: none;  padding: 0;  }  nav ul li {  display: inline;  margin-right: 20px;  }  main {  max-width: 1200px;  margin: 0 auto;  padding: 20px;  background-color: #fff;  }  .product {  margin-bottom: 20px;  padding: 10px;  background-color: #fff;  }  .product img {  max-width: 100%;  }  .product h2 {  color: #333;  }  .product p {  color: #777;  }  .product a {  display: inline-block;  background-color: #333;  color: #fff;  padding: 5px 10px;  text-decoration: none;  border-radius: 5px;  }  footer {  background-color: #333;  color: #fff;  padding: 10px;  text-align: center;  }  </style> </head> <body>  <header>  <h1>Stefani's Art</h1>  <nav>  <ul>  <li><a href="/"<button style="color: azure;">Home</button></a></li>  <li><a href="/items"<button style="color: azure;">Products</button></a></li>  </ul>  </nav>  </header>  <main>  <section>  <h2>About Us</h2>  <p>Ние сме малък екип от художници, които обичат да създават красиви и уникални ръчно изработени дървени рисунки. Използваме естествено дърво и висококачествени материали за изработването на нашите произведения, които са вдъхновени от природата, животните и пейзажите. Нашите рисунки са идеални за декориране на вашия дом, офис или всяко друго пространство. Те са и чудесен подарък за вашите близки.  Започнахме нашия уебсайт през 2024 година, за да споделим нашата страст и визия със света. Надяваме се да се насладите на разглеждането на нашата галерия и да намерите нещо, което ви допада. Ако имате въпроси, предложения или персонализирани заявки, моля, не се колебайте да се свържете с нас. Ще се радваме да чуем от вас.  Благодарим ви, че посетихте нашия уебсайт и подкрепихте нашето изкуство.</p>   </section>  </main>  <footer>  <p>&copy; All rights reserved</p>  </footer> </body> </html> |
| --- |

*Фиг. 3.16 about.html*

Файлът About.html съдържа информация за онлайн магазина. Включва заглавие "About Us" и описващ абзац.

* **Home.html**

| <!DOCTYPE html> <html> <head>  <title>Home</title>  <style>  body {  font-family: Arial, sans-serif;  margin: 0;  padding: 0;  background-color: #f0f0f0;  }  header {  background-color: #333;  color: #fff;  padding: 10px;  text-align: center;  }  nav ul {  list-style: none;  padding: 0;  }  nav ul li {  display: inline;  margin-right: 20px;  }  main {  max-width: 1200px;  margin: 0 auto;  padding: 20px;  background-color: #fff;  }  .product {  margin-bottom: 20px;  padding: 10px;  background-color: #fff;  }  .product img {  max-width: 100%;  }  .product h2 {  color: #000000;  }  .product p {  color: #151515;  }  .product a {  display: inline-block;  background-color: #333;  color: #fff;  padding: 5px 10px;  text-decoration: none;  border-radius: 5px;  }  footer {  background-color: #333;  color: #fff;  padding: 10px;  text-align: center;  }  </style> </head> <body>  <header>  <h1>Stefani's art</h1>  <nav>  <ul>  <li><a href="/"<button style="color: azure;">Home</button></a></li>  <li><a href="/items"<button style="color: azure;">Products</button></a></li>  <li><a href="/about"<button style="color: azure;">About us</button></a></li>  <li><a href="/signup"<button style="color: azure;">Sign Up</button></a></li>  <li><a href="/signout"<button style="color: azure;">Sign Out</button></a></li>  <li><a href="/profile"<button style="color: azure;">Profile</button></a></li>  <li><a href="/signin"<button style="color: azure;">Login</button></a></li>    </ul>  </nav>  </header>  <main>  <section class="product">  <img src="product1.png" alt="Snimka">  <h2>Сайт за продажба на ръчно направени изделия</h2>  <p>Изделията биват дъски, нарисувани с пирограф, картини и други.</p>  </section>  </main>  <footer>  <p>&copy; All rights reserved</p>  </footer> </body> </html> |
| --- |

*Фиг. 3.17 home.html*

Файлът Home.html е началната страница на уебсайта. Тя представя кратко компанията, изделията, които предлага (ръчно направени дървени изделия), и включва примерни изображения на продуктите. Също така, предлага навигационно меню за достъп до другите раздели на сайта.

* **Items.html**

| <!DOCTYPE html> <html> <head>  <title>Products</title>  <style>  body {  font-family: Arial, sans-serif;  margin: 0;  padding: 0;  background-color: #f0f0f0;  }  header {  background-color: #333;  color: #fff;  padding: 10px;  text-align: center;  }  nav ul {  list-style: none;  padding: 0;  }  nav ul li {  display: inline;  margin-right: 20px;  }  main {  max-width: 1200px;  margin: 0 auto;  padding: 20px;  background-color: #fff;  }  .product {  margin-bottom: 20px;  padding: 10px;  background-color: #fff;  }  .product img {  max-width: 100%;  }  .product h2 {  color: #333;  }  .product p {  color: #777;  }  .product a {  display: inline-block;  background-color: #333;  color: #fff;  padding: 5px 10px;  text-decoration: none;  border-radius: 5px;  }  footer {  background-color: #333;  color: #fff;  padding: 10px;  text-align: center;  }  </style> </head> <body>  <header>  <h1>Stefani's art</h1>  <nav>  <ul>  <li><a href="/"<button style="color: azure;">Home</button></a></li>  <li><a href="/items"<button style="color: azure;">Products</button></a></li>  <li><a href="/about"<button style="color: azure;">About us</button></a></li>  </ul>  </nav>  </header>  <main>  <section class="Product">  <img src="static/moon.jpg" alt="Product Image">  <h2>Product 1</h2>  <p>Handmade wooden drawing</p>  <p><strong>Price: 1000 lv</strong></p>  </section>  </main>  <footer>  <p>&copy; All rights reserved</p>  </footer> </body> </html> |
| --- |

*Фиг. 3.18 items.html*

Файлът Items.html е страницата за продукти, в която потребителите могат да разгледат всички предлагани артикули. Включва изображения, заглавия, описания и цени на продуктите.

* **Login.html**

| {% extends 'base.html' %}  {% block title %}Login{% endblock %}  {% block content %}  <h2>Login</h2> {{ msg }}  <form method="post">  {% csrf\_token %}  {{ form.as\_p }}  <button type="submit" class="btn btn-primary">Login</button>  </form> {% endblock %} |
| --- |

*Фиг. 3.19 login.html*

Страницата за вход (login.html) позволява на потребителите да влязат в своите акаунти, като предоставят своето потребителско име и парола. Включва форма с две полета за въвеждане - едно за потребителско име и друго за парола, и бутон за изпращане на данните. Целта е да се предостави бърз и безпроблемен начин за потребителите да получат достъп до своите акаунти.

* **Profile.html**

| {% extends 'base.html' %}  {% block title %}Login{% endblock %}  {% block content %}  <h2>Welcome!</h2>  {% if user.is\_authenticated %}  Hi {{ user.username }}!  <a href="/signout" class="btn btn-danger">Logout</a>  <a href="/" class="btn btn-danger, color: #7af61c">Home page</a>  {% else %}  <a href="/signin" class="btn btn-primary"><span class="glyphicon glyphicon-log-in"></span>Login</a>  <a href="/signup" class="btn btn-success">Signup</a>  {% endif %} {% endblock %} |
| --- |

*Фиг. 3.20 profile.html*

Страницата за профил (profile.html) позволява на потребителите да преглеждат своя личен профил след като са влезли в системата. Тази страница показва потребителската информация като име. Целта е да се предостави персонализирано потребителско пространство, където потребителите могат да прегледат своите настройки и информация.

* **Signup.html**

| {% extends 'base.html' %}  {% block content %}  <h2>Sign up</h2>  <form method="post">  {% csrf\_token %}  {% for field in form %}  <p>  {{ field.label\_tag }}<br>  {{ field }}  {% for error in field.errors %}  <p style="color: red">{{ error }}</p>  {% endfor %}  </p>  {% endfor %}  <button type="submit" class="btn btn-success">Sign up</button>  </form> {% endblock %} |
| --- |

*Фиг. 3.21 signup.html*

Страницата за регистрация (signup.html) е предназначена за нови потребители, които искат да създадат акаунт. Тази страница съдържа форма с повече полета за въвеждане в сравнение със страницата за вход - потребителско име, парола (и потвърждение на паролата). Целта е да се събере необходимата информация за създаване на нов потребителски профил.

## **Изграждане на контейнер имидж на приложението**

Изграждането на контейнер имидж е направено чрез Dockerfile. Този Dockerfile е инструкционен файл, който се използва от Docker за създаване на образ (image) на контейнер, който да съдържа всички необходими зависимости и конфигурации за стартиране на Python приложението. Този файл се вижда на Фиг. 3.22.

| FROM python:3.9-alpine ENV PYTHONUNBUFFERED 1 RUN mkdir /code WORKDIR /code COPY requirements.txt /code/ RUN apk add --update pkgconfig mariadb-dev build-base gcc python3-dev linux-headers RUN pip install -r requirements.txt COPY . /code/ EXPOSE 8000 CMD ["python3", "manage.py" , "runserver", "0.0.0.0:8000"] |
| --- |

*Фиг. 3.22 Dockerfile*

* **FROM python:3.9-alpine**

Тази команда задава основния образ (base image), от който ще се изгради контейнерът. В случая, използва се официалният Python образ с версия 3.9, който е базиран на Alpine Linux. Alpine Linux е избран заради своята малка големина и ефективност, което прави крайния образ по-лек и бърз за изтегляне и стартиране.

* **ENV PYTHONUNBUFFERED 1**

Задава променлива на средата PYTHONUNBUFFERED на стойност 1. Това гарантира, че Python изходът се показва в конзолата в реално време, което е полезно за наблюдение на логовете и отстраняване на грешки.

* **RUN mkdir /code**

Изпълнява команда в контейнера, за да създаде директория /code. Тази директория се използва за съхранение на кода на приложението.

* **WORKDIR /code**

Задава /code като работна директория за контейнера. Всички следващи RUN, CMD, ENTRYPOINT, COPY и ADD инструкции са изпълнявани в тази директория.

* **COPY requirements.txt /code/**

Копира файла requirements.txt, който съдържа списък на Python пакетите необходими за приложението, от локалната машина в контейнера, в директорията /code.

* **RUN apk add --update pkgconfig mariadb-dev build-base gcc python3-dev linux-headers**

Инсталира необходимите пакети и зависимости чрез apk, мениджъра за пакети на Alpine Linux. Тези зависимости са необходими за компилиране и инсталация на Python пакетите, които могат да изискват нативни разширения.

* **RUN pip install -r requirements.txt**

Изпълнява pip за инсталиране на Python пакетите, изброени в requirements.txt.

* requirements.txt е текстов файл, използван за дефиниране на нужните на приложението библиотеки

| asgiref beautifultable blinker certifi charset-normalizer click colorama contourpy cycler Django filelock Flask fonttools idna itsdangerous Jinja2 kiwisolver MarkupSafe matplotlib numpy packaging Pillow psutil pyparsing pysondb python-dateutil requests six sqlparse tzdata urllib3 wcwidth Werkzeug |
| --- |

*Фиг. 3.23 requirements.txt*

* **COPY . /code/**

Копира всички файлове от текущата директория (където се намира Dockerfile) в контейнера, в директорията /code.

* **EXPOSE 8000**

Сигнализира Docker, че контейнерът слуша на порт 8000.

* **CMD ["python3", "manage.py", "runserver", "0.0.0.0:8000"]**

Задава командата по подразбиране, която се изпълнява, когато контейнерът стартира. Стартира Django разработческия сървър на адрес 0.0.0.0 и порт 8000, което прави сървъра достъпен отвън за хост машината.

## **Създаване на необходимите Kubernetes ресурси, за да се постигне работоспособност на приложението и поставяне на разпределител на мрежовия трафик пред приложението в клъстера**

Kubernetes ресурсите определят и конфигурират доставка, обслужване, вход (ingress) и ConfigMap за приложение, наречено "marketplace". Всеки файл дефинира специфичен аспект от желаната структура и поведение на приложението в Kubernetes среда. Тези ресурси заедно създават сигурна и мащабируема доставка на приложението "marketplace" в Kubernetes среда, осигурявайки управление на трафика, сигурност и гъвкавост при достъпа до приложението. За използването на тези ресурси е необходимо инсталирането на kubectl. Създадени са следните файлове:

* **deployment.yaml**

| apiVersion: apps/v1 kind: Deployment metadata:  name: marketplace  labels:  app: marketplace spec:  replicas: 1  selector:  matchLabels:  app: marketplace  template:  metadata:  labels:  app: marketplace  spec:  containers:  - name: marketplace  image: registry.gitlab.com/diplom-works-2023/securing-containerized-applications-in-cloud-environments-based-on-kubernetes/marketplace:2  ports:  - containerPort: 8000  securityContext:  capabilities:  drop:  - all  runAsUser: 1000  allowPrivilegeEscalation: false  readOnlyRootFilesystem: true  volumeMounts:  - name: write  mountPath: /write  volumes:  - name: write  emptyDir: {} |
| --- |

*Фиг. 3.24 deployment.yaml*

Този файл дефинира Deployment ресурс, който управлява доставката и скалирането на набор от реплики на контейнера на приложението. За добавянето на ресурсa в клъстера, се използва командата kubectl apply -f deployment.yaml

* apiVersion: apps/v1: Специфицира версията на API, която се използва за създаването на Deployment.
* kind: Deployment: Определя типа на ресурса (Deployment).
* metadata: Съдържа метаданни за Deployment, като например име (name: marketplace) и етикети (labels: app: marketplace), които помагат за идентификацията и организацията на ресурсите.
* spec: Описва желаното състояние на Deployment, включително броя на репликите, селектора за подбор на реплики и шаблона на пода (template), който дефинира как да бъде създаден всеки контейнер.
* replicas: 1: Задава броя на репликите на пода, които да бъдат поддържани.
* selector: Определя как Kubernetes идентифицира кои подове да управлява чрез този Deployment.
* template: Дефинира шаблона на подовете, които да бъдат създадени, включително етикетите и спецификациите на контейнера, като изображението, портовете и сигурността.
* containers: Описва контейнерите, които трябва да бъдат стартирани в пода, включително името, изображението и портовете, които контейнерът използва.
* securityContext: Дефинира настройки за сигурността на контейнера, като например отказване на всички капацитети, задаване на потребителски ID и забрана за ескалация на привилегии.
* volumeMounts и volumes: Конфигурира директории за запазване на данни, като в случая се създава временно хранилище с emptyDir.
* **service.yaml**

Този файл дефинира Service ресурс, който управлява достъпа до един или повече подове, базиран на селектори и портове. За добавянето на ресурсите в клъстера, се използва командата kubectl apply -f service.yaml

| apiVersion: v1 kind: Service metadata:  name: marketplace-svc spec:  selector:  app: marketplace  ports:  - protocol: TCP  port: 8000  targetPort: 8000  name: https |
| --- |

*Фиг. 3.25 service.yaml*

* apiVersion: v1: Специфицира версията на API за Service.
* kind: Service: Определя типа на ресурса (Service).
* metadata: Съдържа метаданни за Service, като име.
* spec: Описва конфигурацията на Service, включително селектора за подбор на подове, които ще обслужва, и дефиниции на портовете.
* **ingress.yaml**

| apiVersion: networking.k8s.io/v1 kind: Ingress metadata:  name: marketplace-ingress  annotations:  nginx.ingress.kubernetes.io/force-ssl-redirect: "true"  nginx.ingress.kubernetes.io/ssl-protocols: "TLSv1.2 TLSv1.3"  nginx.ingress.kubernetes.io/rewrite-target: "/"  nginx.ingress.kubernetes.io/enable-modsecurity: "true"  nginx.ingress.kubernetes.io/modsecurity-snippet: |  SecRuleEngine On  SecRequestBodyAccess On  SecAuditEngine RelevantOnly  SecAuditLogParts ABIJDEFHZ  SecAuditLog /var/log/modsec\_audit.log  SecRule REQUEST\_HEADERS:User-Agent "@contains fern-scanner" "id:1000001,deny,status:403" spec:  tls:  - hosts:  - marketplace.com  secretName: marketplace-tls  ingressClassName: nginx  rules:  - host: marketplace.com  http:  paths:  - path: /  pathType: Prefix  backend:  service:  name: marketplace-svc  port:  number: 8000 |
| --- |

*Фиг. 3.26 ingress.yaml*

Този файл дефинира Ingress ресурс, който управлява външен достъп до услугите в Kubernetes клъстера, чрез HTTP/HTTPS. За добавянето на ресурсите в клъстера, се използва командата kubectl apply -f ingress.yaml

* apiVersion: networking.k8s.io/v1: Версия на API за Ingress.
* kind: Ingress: Тип на ресурса.
* metadata: Метаданни, включително име и анотации за настройка на поведението на nginx ингрес контролера.
* spec: Описва правилата за маршрутизиране на трафика към услугите, TLS сертификатите за HTTPS и класа на ингреса.
* **modsecurity\_configmap.yaml**

| apiVersion: v1 kind: ConfigMap metadata:  name: ingress-nginx-controller data:  allow-snippet-annotations: "true"  enable-modsecurity: "true"  enable-owasp-core-rules: "true" |
| --- |

*Фиг. 3.27 modsecurity\_configmap.yaml*

Този файл създава ConfigMap, който се използва за конфигуриране на nginx ингрес контролера, за активиране и настройка на ModSecurity, WAF (Web Application Firewall). За добавянето на ресурсите в клъстера, се използва командата kubectl apply -f modsecurity\_configmap.yaml

* apiVersion: v1: Версия на API за ConfigMap.
* kind: ConfigMap: Тип на ресурса.
* metadata: Метаданни за ConfigMap.
* data: Стойности, които конфигурират поведението на ингрес контролера, разрешаване на ModSecurity и OWASP Core Rules.
* **network\_policy.yaml**

| kind: NetworkPolicy apiVersion: networking.k8s.io/v1 metadata:  namespace: default  name: deny-from-other-namespaces spec:  podSelector:  matchLabels:  ingress:  - from:  - podSelector: {} |
| --- |

*Фиг. 3.28 network\_policy.yaml*

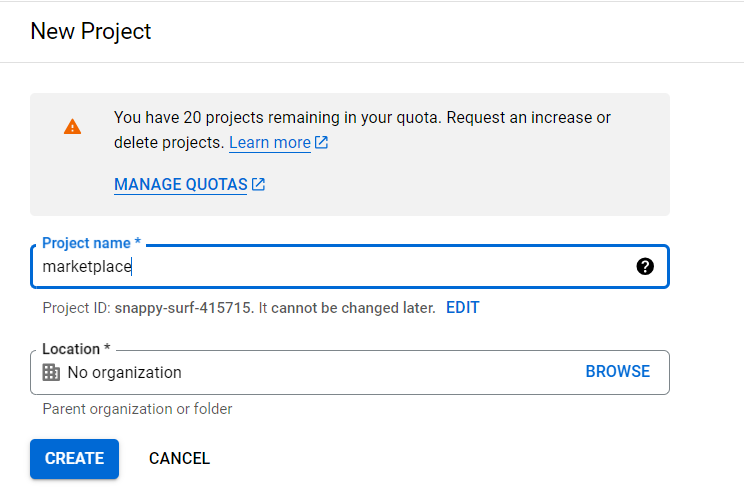
NetworkPolicy ефективно блокира входящия трафик от подове в други namespaces, като по този начин ограничава комуникацията само между подовете в същия namespace (default). Това не налага никакви ограничения върху подовете в default namespace, което означава, че всички подове в този namespace могат да комуникират помежду си свободно.

* kind: NetworkPolicy - Определя типа на обекта, който се създава. В този случай, обектът е мрежова политика.
* apiVersion: networking.k8s.io/v1 - Указва версията на API-то, което се използва за създаване на този обект. networking.k8s.io/v1 е стабилната версия на API-то за мрежови политики.
* metadata: - Секция, която съдържа метаданни за NetworkPolicy обекта.
* namespace: default - Определя namespace-а, в който ще бъде приложена тази мрежова политика. В този случай, политиката се прилага в default namespace-а.
* name: deny-from-other-namespaces - Името на мрежовата политика. Това име трябва да бъде уникално в рамките на namespace-а.
* spec: - Определя спецификациите на мрежовата политика.
* podSelector: - Указва подовете, които ще бъдат засегнати от тази политика, чрез съвпадение на техните етикети (labels). В този случай, podSelector е празен, което означава, че политиката се прилага към всички подове в указания namespace.
* ingress - Определя правилата за входящ трафик към засегнатите подове.
* from: - Определя източниците, от които може да идва входящият трафик.
* - podSelector: {} - Този ред указва, че входящият трафик може да идва от подове, които отговарят на критериите, зададени в podSelector. Тъй като podSelector е празен, това означава, че входящият трафик е разрешен от всички подове в същия namespace.

## Качване на приложението в GCP

* **Създаване на проект**

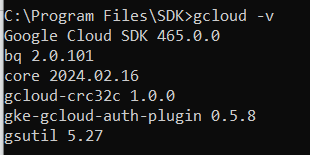
За да се качи приложението, е създаден проект в GCP, както е показано на Фиг. 3.29



*Фиг. 3.29 Създаване на проект*

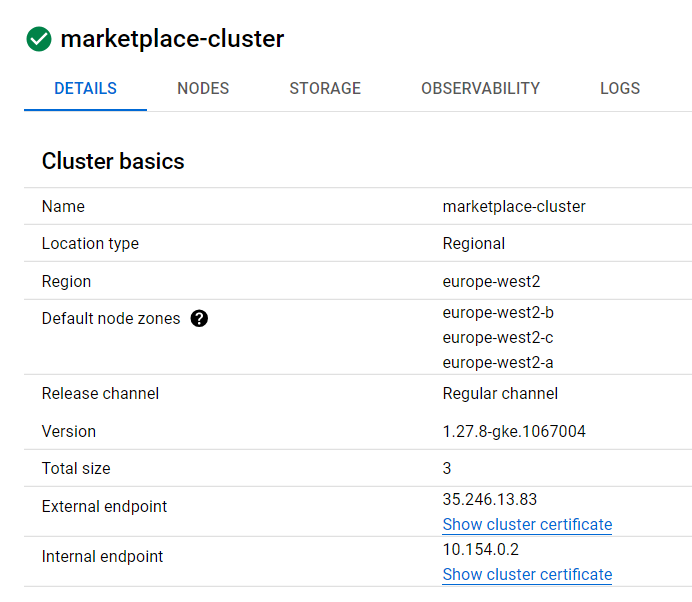
* **Инсталиране на Google SDK и влизане в профил**

Google SDK (Software Development Kit) е набор от инструменти, предназначени за разработка на приложения, които взаимодействат с Google Cloud APIs и управляват ресурсите в проекти в Google Cloud.



*Фиг. 3.30 Инсталиран Google SDK*

* **Създаване на Kubernetes клъстер**

В интерфейса на Google Cloud се инициира създаването на Kubernetes клъстер. Зададени са параметри: име - marketplace-cluster, брой на node - 3, регион - europe-west2 и вид на машините- e2-medium. 

*Фиг. 3.31 Създаден Kubernetes клъстер*

* **Качване на локалния имидж в GCP**

За да се качи локалния имидж в GCP, първо трябва да се изпълни командата ‘docker tag [LOCAL\_IMAGE\_NAME]:[TAG] gcr.io/[PROJECT-ID]/[IMAGE]:[TAG]’.

* [LOCAL\_IMAGE\_NAME]:[TAG] - името и тага на локалния имидж (в случая marketplace-image:1)
* [PROJECT-ID] - името и ID-то на проекта (marketplace-415016)
* [IMAGE]:[TAG] - името и така на имиджа

След това, чрез командата ‘docker push gcr.io/[marketplace-415016]/marketplace-image:1

* **Добавяне на конфигурационните манифести в GCP клъстер**

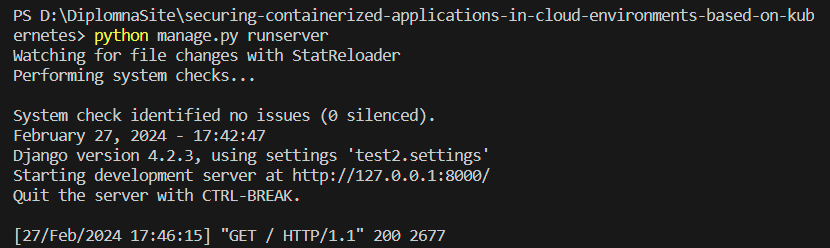
Добавяне на конфигурационните манифести в GCP клъстера се изпълнява чрез командите

* kubectl apply -f deployment.yaml
* kubectl apply -f ingress.yaml
* kubectl apply -f service.yaml
* kubectl apply -f modsecurity\_configmap.yaml
* kubectl apply -f network\_policy.yaml

# **Четвърта глава - Доказване на работоспособност**

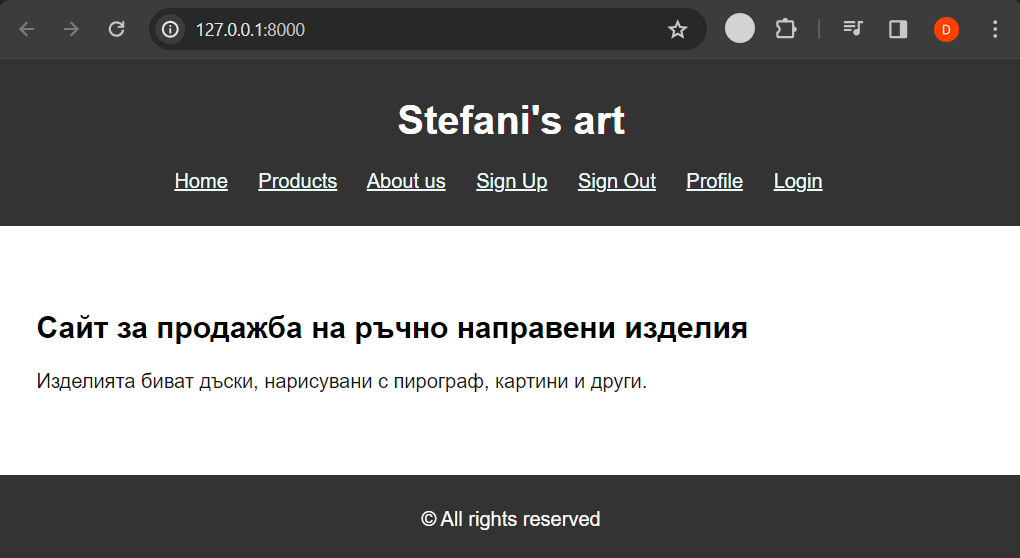
## **Доказване работоспособността на приложението**

Приложението е стартирано локално чрез командата python “manage.py runserver”. На Фиг. 4.1 се вижда изпълнението на командата и активността в сайта.



*Фиг. 4.1 Локално стартиране и информация за уеб приложение*

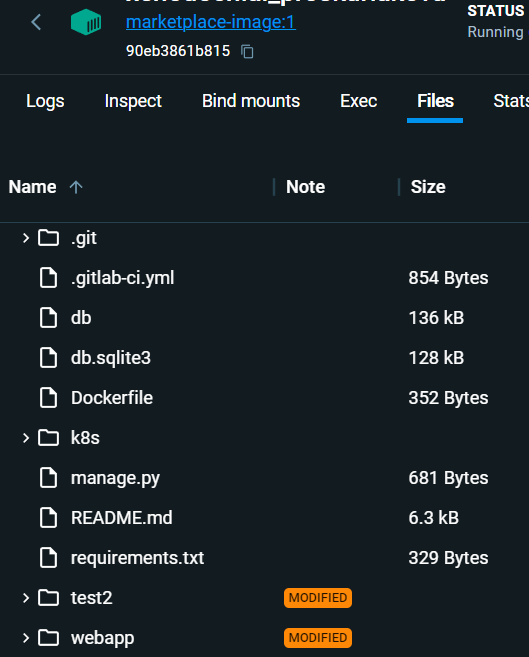
На Фиг. 4.2 е показано стартирано локално уеб приложението и начална страница, което е контейнеризирано и представено в



*Фиг. 4.2*

## Доказване създаден контейнер имидж

На Фиг. 4.3 е показан работещ контейнер имидж, съдържащ всички необходими файлове



*Фиг. 4.3*

## Доказване работоспособността на NGINX

Nginx Ingress Controller е ключов компонент, който управлява достъпа до приложенията, работещи в клъстера. Той служи като входна точка (ingress point) за входящия трафик към клъстера и позволява маршрутизиране на заявки към различни услуги (services) на базата на заявените правила. Всички входящи HTTP(S) заявки към приложението в Kubernetes клъстера първоначално се насочват към Nginx Ingress Controller. На Фиг. 4.4 е показана заявка за страницата “About us”.



*Фиг. 4.4*

## Доказване защита на ниво контейнер

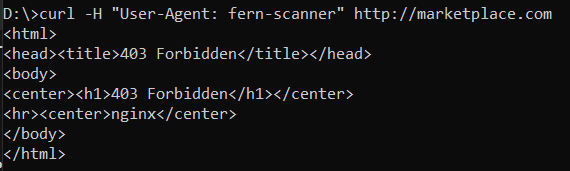
За целта, на Фиг. 4.5, се показва опит за промяна на конфигурационен файл на приложението ‘settings.py’. На нея се вижда, че няма достъп до промяна на файла, защото е ReadOnly.



*Фиг. 4.5*

## **Доказване работоспособност на WAF**

Конфигурацията на ModSecurity е директно интегрирана в Ingress чрез анотация. Това включва специфично правило, което блокира заявки с User-Agent, съдържащ fern-scanner. Пример за заявка с User-Agent, съдържащ fern-scanner, може да изглежда така когато се прави през командния ред с инструмента curl. Заявката цели да имитира достъп от потенциално злонамерен скенер за уязвимости, като използва стойността на User-Agent за идентификация. На Фиг 4.6 е показано изпълнението на командата ‘curl’



*Фиг. 4.6 Засичане на злонамерено сканиране*

## Доказване работоспособност на мрежовата политика

За целта е създаден нов под, в различен namespace (different) с командата ‘kubectl create namespace different’.

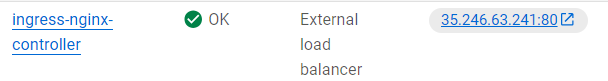
След това се стартира временен контейнер в namespace different, използвайки лекия образ alpine и предоставя достъп до shell, откъдето се изпълнява командата wget за тестване на достъпа до ресурс наречен web във default namespace. Ако надвиши определения timeout, без да има отговор, подът автоматично се изтрива. Това се случва чрез командата ‘kubectl run test-$RANDOM --namespace=foo --rm -i -t --image=alpine -- sh/ # wget -qO- --timeout=2 http://web.default’.



*Фиг. 4.7 Блокиран трафик в различен namespace*

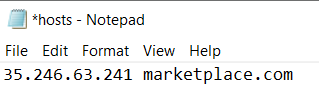
## Доказване работоспособност на клъстер в GCP среда

След създадените конфигурационни манифести, се вижда публично IP адрес на NGINX Controller-a, през който минава целия трафик.

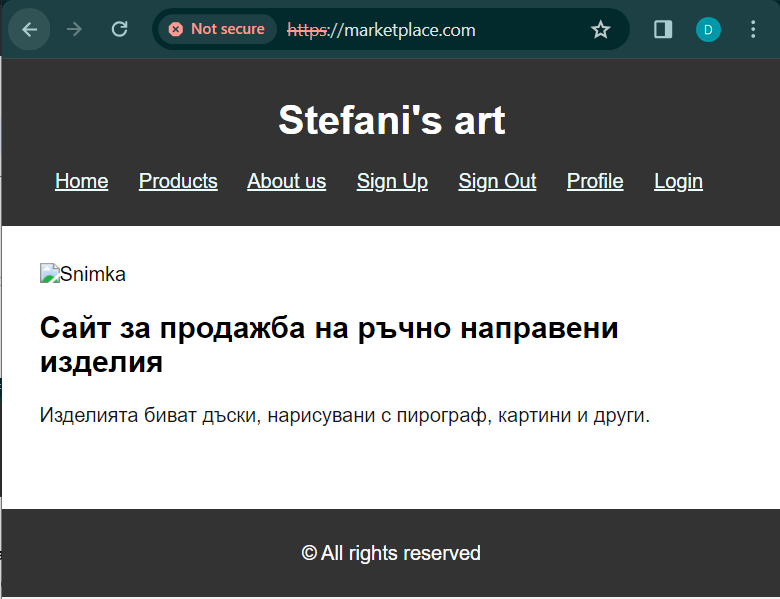


*Фиг. 4.8 Публичен адрес на NGINX Controller*

За постигане на отваряне на сайта , е променен HOSTS файла на машината.



*Фиг. 4.9 Промяна на HOSTS файл на локална машина*



*Фиг. 4.10 Работещо приложение в GCP*

# **Заключение**

В заключение на настоящата дипломна работа на тема "Защита на контейнеризирани приложения в облачни среди, базирани на Kubernetes", всички изисквания за успешната реализация на темата са покрити, като е постигнато значително задълбочаване в разбирането и прилагането на защитни мерки за контейнеризирани приложения. Проучването демонстрира различни стратегии и практики за осигуряване на високо ниво на сигурност за приложенията, които се изпълняват в облачни среди.

По време на изследването са разгледани различни аспекти на сигурността. Също така, е даден акцент на важността от интеграцията на сигурността в процеса на разработка на софтуер, за да се гарантира, че сигурността е вградена в приложението от самото начало.

За бъдещо развитие на темата могат да се добавят следните функционалности:

* Добавяне на контейнер скенер в CI/CD build процеса: Това позволява автоматично сканиране на контейнери за известни уязвимости в зависимостите на приложението преди тяхното разгръщане в продукция. Така може да се намали рискът от експлоатация на известни уязвимости.
* Добавяне на runtime container scanner: Реализацията на скенери, които работят в реално време, дава възможност за непрекъснато наблюдение и анализ на контейнерите, което помага за идентифицирането на злонамерени действия или уязвимости, които се проявяват по време на изпълнението.
* Внедряване на Endpoint Detection and Response (EDR) системи: Тези системи предоставят средства за откриване на заплахи и реагиране при инциденти, което може значително да увеличи видимостта и контрола над сигурността на контейнеризираните приложения.
* Използване на multistage build за hardening на образите: Този подход е ефективен за намаляване на атакуемата повърхност на контейнерните образи, като се изграждат образи с минимално необходимите зависимости и конфигурации.

Такива подобрения ще допринесат за повишаване на сигурността на контейнеризираните приложения в облачни среди, осигурявайки по-добра защита срещу постоянно развиващите се заплахи. Реализацията на предложените мерки за бъдещо развитие ще доведе до създаването на още по-здрава и устойчива архитектура на приложенията

# **Използвана литература**

[1] Docker Deep Dive - Nigel Poulton

[2] The Kubernetes Book - Nigel Poulton

[3] DoD Enterprise DevSecOps Fundamentals - <https://dl.dod.cyber.mil/wp-content/uploads/devsecops/pdf/DoDEnterpriseDevSecOpsFundamentals.pdf>

[4] Container Security: Fundamental Technology Concepts that Protect Containerized Applications - Liz Rice

[5] Application Container Security Guide - <https://nvlpubs.nist.gov/nistpubs/specialpublications/nist.sp.800-190.pdf>

[6] Kubernetes Security - <https://cdn2.hubspot.net/hubfs/1665891/Assets/Kubernetes%20Security%20-%20Operating%20Kubernetes%20Clusters%20and%20Applications%20Safely.pdf>

[7] Google Cloud overview - <https://cloud.google.com/docs/overview/>

[8] Virtual Private Cloud (VPC) overview - <https://cloud.google.com/vpc/docs/overview>

[9] OWASP Top Ten - <https://owasp.org/www-project-top-ten/>

[10] Network Policies - <https://kubernetes.io/docs/concepts/services-networking/network-policies/>

[11] NGINX - <https://docs.nginx.com/nginx-ingress-controller/>

[12] Django Documentation - <https://docs.djangoproject.com/en/5.0/>