МИНИСТЕРСТВО НА ОБРАЗОВАНИЕТО И НАУКАТА

ПРОФЕСИОНАЛНА ГИМНАЗИЯ ПО ЕЛЕКТРОТЕХНИКА И ЕЛЕКТРОНИКА
бул. Пещерско шосе № 26
4002 гр. Пловдив, България
тел. 032 / 643-657
info-1690174@edu.mon.bg

VOCATIONAL SCHOOL OF ELECTRICAL ENGINEERING AND ELECTRONICS
26 Peshtersko chaussee blvd.,
4002 Plovdiv, Bulgaria
Phone: 032 / 643-657
info-1690174@edu.mon.bg

професия код 481030 „Приложен програмист“

специалност код 4810301 „Приложно програмиране“

КУРСОВ ПРОЕКТ

за придобиване един голям [redacted]

ТЕМА: ПРОГРЕСИВНИ УЕБ ПРИЛОЖЕНИЯ (PWA) И ТЯХНАТА ПОДДРЪЖКА ОТ ФРЕЙМУЪРКОВЕ

Дипломант: …… /Момчил Калестров/

Ръководител-консултант: …… /Иван Дановски/

Клас: 12. А

e-mail: kalestrov.mo@gmail.com

Пловдив

2025 година

СЪДЪРЖАНИЕ

УВОД

Прогресивните уеб приложения представляват вид уеб приложения, които могат да бъдат инсталирани на различни устройства и да функционират като самостоятелни приложения. Инсталацията на PWA се осъществява чрез офлайн кеша на уеб браузъра, без необходимост от традиционен инсталационен пакет. По този начин потребителите получават приложение, което се държи подобно на нативно, но се разпространява и поддържа чрез уеб технологии.

Концепцията за PWA е въведена през 2016 г. като алтернатива на т. нар. „native“ приложения, специфични за отделни платформи и устройства. Основното им предимство е, че не изискват отделно пакетиране или разпространение за различни операционни системи. Те могат да бъдат използвани на широк набор устройства, като публикуването на приложенията в цифрови платформи за разпространение като Apple App Store, Google Play или Microsoft Store е по избор, а не задължително условие.

Тъй като прогресивните уеб приложения се доставят под формата на уеб страници или уеб сайтове, изградени с общоприети уеб технологии като HTML, CSS, JavaScript и от скоро WebAssembly, те могат да функционират на всяка платформа с браузър, съвместим с PWA стандартите. Към 2025 г. поддръжката на PWA функционалности е налична в различна степен в браузъри като Google Chrome, Safari, Edge, Brave и Firefox, което допълнително увеличава приложимостта на този подход.

В контекста на съвременното уеб разработване, съвременните JavaScript фреймуъркове играят ключова роля при изграждането на прогресивни уеб приложения. Фреймуъркове като Next.js и Angular предоставят вградени или улеснени механизми за работа със Service Workers, управление на кеширане, офлайн функционалност и push известия. Това позволява създаването на надеждни, производителни и мащабируеми PWA решения, които отговарят на нарастващите изисквания на потребителите към модерните уеб приложения.

ГЛАВА 1 АРХИТЕКТУРА НА PWA И КЛЮЧОВИ ТЕХНОЛОГИИ

1.1. Service Workers

Service Workers представляват специален тип JavaScript скриптове, които се изпълняват във фонов режим, независимо от уеб страницата, и действат като посредник между уеб приложението, браузъра и мрежата. Те позволяват прихващане и управление на мрежови заявки, което ги прави основна технология при реализирането на прогресивни уеб приложения. За разлика от стандартните клиентски скриптове, Service Workers нямат директен достъп до DOM и се изпълняват в изолиран контекст, което повишава сигурността и стабилността на приложенията.

Една от основните функции на Service Workers е управлението на кеширане на ресурси. Чрез различни стратегии за кеширане, като cache-first, network-first или stale-while-revalidate, уеб приложенията могат да осигурят бързо зареждане и функционалност дори при липса на интернет връзка. Това позволява реализирането на офлайн режим, при който потребителят може да използва основни функции на приложението без активна мрежова свързаност.

Освен офлайн функционалност, Service Workers играят ключова роля и при реализирането на push известия. Те могат да получават съобщения от push сървър, дори когато уеб приложението не е активно в браузъра, и да показват известия към потребителя. Това значително подобрява ангажираността и доближава уеб приложенията до функционалността на нативните мобилни приложения.

Service Workers се регистрират и управляват от страна на клиента, като работят само в защитен контекст (HTTPS), с изключение на локална разработка. Техният жизнен цикъл включва фази като регистрация, инсталация, активиране и обновяване, което позволява контролирано внедряване на нови версии на приложението без прекъсване на потребителското изживяване.

В съвременните уеб фреймуъркове, като Next.js и Angular, поддръжката на Service Workers е значително улеснена чрез готови модули и инструменти за конфигурация. Това позволява на разработчиците да внедряват PWA функционалности по по-структуриран и стандартизиран начин, без необходимост от ръчно управление на всички ниско ниво детайли.

1.2. Web App Manifest

Web App Manifest представлява JSON файл, който описва метаданните на уеб приложението и определя начина, по който то се представя и държи при инсталация върху потребителско устройство. Той е основен компонент на прогресивните уеб приложения (PWA) и служи като връзка между уеб приложението и операционната система, позволявайки инсталиране и стартиране на приложението подобно на нативно.

Чрез Web App Manifest разработчиците могат да дефинират основни характеристики на приложението, като име и кратко име, начална страница, режим на показване (display mode), ориентация на екрана, цветове на интерфейса и икони с различни размери и резолюции. Тези настройки влияят пряко върху начина, по който приложението се визуализира в менюто с приложения, на началния екран и при стартиране от потребителя.

Една от ключовите функции на Web App Manifest е възможността за стартиране на уеб приложението в самостоятелен режим, без адресна лента и стандартни браузърни контроли. Това създава усещане за „native“ приложение и подобрява потребителското изживяване. Освен това файлът позволява задаване на предпочитана ориентация и визуална тема, което осигурява по-добра интеграция с операционната система на устройството.

Web App Manifest се зарежда от браузъра чрез препратка в HTML документа и се поддържа от съвременните браузъри в различна степен. Заедно със Service Workers, той е задължително условие за пълноценна PWA функционалност. Докато Service Workers осигуряват логиката за офлайн работа и push известия, Manifest файлът дефинира начина на инсталация и визуално представяне на приложението.

В рамките на съвременните JavaScript фреймуъркове, като Next.js и Angular, създаването и управлението на Web App Manifest е значително улеснено чрез вградени конфигурационни механизми. Това позволява разработчиците да интегрират PWA възможности по стандартизиран и ефективен начин, без необходимост от ръчно управление на всички параметри.

1.3. Кеширане и управление на ресурси

Една от основните цели на прогресивните уеб приложения (PWA) е осигуряване на бързо и надеждно потребителско изживяване, дори при ограничена или липсваща интернет връзка. За постигането на това се използва кеширане и управление на ресурси, които позволяват на приложението да зарежда необходимите файлове локално, без да се налага непрекъснато запитване към сървъра.

Кеширането се реализира основно чрез Service Workers, които контролират кои ресурси да бъдат съхранявани в кеша и как да се обслужват последващите заявки. Съществуват различни стратегии за кеширане, сред които най-често използваните са:

- Cache-first - при заявка на ресурс се проверява първо кеша, а ако ресурсът не е наличен, се прави мрежова заявка. Тази стратегия осигурява бързо зареждане на вече кеширани файлове;
- Network-first - при заявка се опитва да се изтегли ресурсът от мрежата, а ако това не е възможно, се използва кешираната версия. Подходяща е за динамично съдържание, което се променя често;
- Stale-while-revalidate - комбинация между двата подхода, при която се връща кеширан ресурс веднага, но едновременно се актуализира кешът с новата версия от мрежата.

Освен избор на стратегия, важно е и управлението на версията на кеша. При обновяване на приложението старите кеширани ресурси трябва да бъдат правилно изчистени, за да се предотврати конфликт или показване на остаряло съдържание. Това се постига чрез управление на имена и версии на кешовете и контрол на жизнения цикъл на Service Workers.

Чрез ефективно кеширане и управление на ресурси, PWA могат да предложат почти нативно потребителско изживяване – бързо зареждане на интерфейс, работа офлайн и минимално забавяне при всяка навигация в приложението. Тази функционалност е ключова за ангажираността на потребителя и за успешното внедряване на прогресивни уеб приложения на различни платформи и устройства.

ГЛАВА 2 ПОДДРЪЖКА НА PWA ВЪВ ФРЕЙМУЪРКОВЕ

2.1. Изграждане на PWA с Next.js

Next.js е популярен JavaScript фреймуърк, базиран на React, който предоставя разширени възможности за разработка на модерни уеб приложения. Благодарение на поддръжката на server-side rendering, статично генериране и оптимизирано управление на ресурси, Next.js е подходящ избор за изграждане на прогресивни уеб приложения (PWA), които изискват висока производителност и добро потребителско изживяване.

Една от ключовите стъпки при изграждане на PWA с Next.js е конфигурирането на Service Worker. Важно е да се отбележи, че Next.js не предоставя специализиран или вграден механизъм за регистрация на Service Worker. Вместо това, регистрацията се извършва чрез стандартния интерфейс navigator.serviceWorker, който е част от уеб стандартите. Това означава, че разработчикът трябва ръчно да регистрира Service Worker от клиентската страна на приложението, обикновено при инициализация на приложението или в жизнения цикъл на React компонент.

Web App Manifest също играе съществена роля в процеса. В Next.js той може да бъде конфигуриран чрез статичен JSON файл или чрез динамично генериране, което позволява гъвкав контрол върху метаданните на приложението. Manifest файлът дефинира начина на инсталация, визуалното представяне и режима на стартиране на приложението, като по този начин го доближава до нативните мобилни приложения.

Next.js предоставя и оптимизации, които са особено важни за PWA, като автоматично разделяне на кода, предварително зареждане на страници и оптимизация на изображения. Тези механизми намаляват времето за зареждане и подобряват работата на приложението както при първоначално стартиране, така и при последваща навигация.

В комбинация с React, Next.js позволява изграждането на компонентно-базирани интерфейси, които са лесни за поддръжка и разширяване. Това улеснява внедряването на PWA функционалности и прави Next.js предпочитан инструмент за разработка на прогресивни уеб приложения, които комбинират гъвкавостта на уеб технологиите с усещането за нативно приложение.

2.2. Angular Service Workers и тяхната интеграция

Angular е цялостен уеб фреймуърк, който предоставя вградена поддръжка за изграждане на прогресивни уеб приложения чрез Angular Service Workers. За разлика от Next.js, при който интеграцията на PWA функционалности се извършва ръчно, Angular предлага стандартизиран и официално поддържан механизъм за работа със Service Workers, което значително улеснява разработчиците.

Интеграцията на Service Workers в Angular се осъществява чрез специален модул, който автоматизира процесите по регистрация, кеширане и управление на ресурси. След активиране на PWA поддръжката, Angular генерира конфигурационен файл, който описва кои ресурси да бъдат кеширани и как да се обработват мрежовите заявки. Това позволява лесно реализиране на офлайн функционалност без необходимост от ръчно управление на ниско ниво детайли.

Angular Service Workers използват декларативен подход за конфигурация, при който разработчикът описва желаното поведение чрез конфигурационен файл, вместо да пише императивен JavaScript код. Това намалява риска от грешки и осигурява по-предсказуемо поведение при обновяване на приложението и кешираните ресурси. Освен това Angular автоматично управлява версионирането на кеша и обновяването на Service Worker, като гарантира, че потребителите получават актуалната версия на приложението.

Фреймуъркът предоставя и вградена поддръжка за push известия, като интеграцията с Web Push API е улеснена чрез Angular инструменти и услуги. Това позволява на разработчиците да внедряват известия и фонови събития по структуриран и сигурен начин, съобразен с архитектурата на Angular приложенията.

Благодарение на своята стандартизирана интеграция, Angular е подходящ избор за проекти, при които PWA функционалността е основно изискване, а не допълнение. Вградената поддръжка на Service Workers, заедно с ясния конфигурационен модел, прави Angular предпочитан фреймуърк за изграждане на мащабируеми и поддържани прогресивни уеб приложения.

2.3. Сравнение между Next.js и Angular

Next.js и Angular предлагат различни подходи за изграждане на прогресивни уеб приложения, като основната разлика между тях е степента на вградена поддръжка за PWA функционалности. Angular разполага с официален и стандартизиран механизъм за интеграция на Service Workers, който автоматизира регистрацията, кеширането и управлението на ресурсите чрез декларативна конфигурация. Това го прави подходящ избор за проекти, при които PWA функционалността е основно изискване и се търси предсказуемо и лесно за поддръжка решение.

От своя страна, Next.js не предоставя специализиран инструмент за регистрация на Service Workers и разчита на стандартния navigator.serviceWorker API. Този подход изисква повече ръчна конфигурация, но предлага по-голяма гъвкавост и контрол върху поведението на Service Worker. В комбинация с React и оптимизациите на Next.js, това позволява изграждането на PWA, които са тясно съобразени със специфичните нужди на проекта.

В обобщение, Angular е по-подходящ за разработчици, които търсят готово и стандартизирано PWA решение с минимална конфигурация, докато Next.js е предпочитан в случаи, когато се изисква по-фина настройка и интеграция на PWA функционалности в рамките на по-гъвкава архитектура.

ГЛАВА 3 ОФЛАЙН ФУНКЦИОНАЛНОСТ В ПРОГРЕСИВНИТЕ УЕБ ПРИЛОЖЕНИЯ

3.1. Стратегии за кеширане

Стратегиите за кеширане са ключов елемент при изграждането на прогресивни уеб приложения, тъй като определят начина, по който приложението обработва мрежовите заявки и използва локално съхранените ресурси. Чрез правилен избор на стратегия може да се постигне бързо зареждане, намалено натоварване на мрежата и надеждна работа при нестабилна или липсваща интернет връзка.

Една от най-често използваните стратегии е cache-first, за която е представена примерна имплементация на приложение 1, при която ресурсите се зареждат първо от кеша, а мрежова заявка се изпраща само ако ресурсът не е наличен локално. Този подход е подходящ за статични файлове като HTML, CSS, JavaScript и изображения, които рядко се променят.

Обратният подход е network-first, при който приоритет се дава на мрежовата заявка, а кешът се използва като резервен вариант при липса на връзка. Тази стратегия е подходяща за динамично съдържание, което трябва да бъде актуално.

Компромисен и често предпочитан вариант е стратегията stale-while-revalidate, при която кешираният ресурс се връща незабавно, а паралелно с това се изпраща заявка към сървъра за обновяване на кеша. Така се постига баланс между бързо зареждане и актуалност на данните. Изборът и комбинирането на различни стратегии за кеширане позволява на PWA да предложи оптимално потребителско изживяване в зависимост от типа съдържание и начина на използване на приложението.

3.2. Работа без интернет връзка

Работата без интернет връзка е една от основните характеристики, които отличават прогресивните уеб приложения (PWA) от традиционните уеб сайтове. Чрез използването на Service Workers и механизми за кеширане, PWA могат да предоставят функционалност и достъп до съдържание дори при пълна липса на мрежова свързаност. Това повишава надеждността на приложението и осигурява по-добро потребителско изживяване в условия на нестабилна интернет връзка.

При офлайн режим предварително кешираните ресурси, като интерфейсни елементи, скриптове и стилове, се зареждат директно от локалното хранилище на браузъра. В зависимост от избраната стратегия за кеширане, приложението може да показва последно наличното съдържание или специално подготвени офлайн страници. За динамични данни често се използват локални хранилища, които позволяват временно съхраняване на информация до възстановяване на интернет връзката.

След възстановяване на свързаността, Service Workers могат автоматично да синхронизират локално запазените промени със сървъра. Това позволява на потребителите да продължат работа без прекъсване и без загуба на данни. По този начин офлайн функционалността превръща PWA в надеждно решение, което доближава уеб приложенията до поведението и удобството на нативните приложения.

ГЛАВА 4 PUSH ИЗВЕСТИЯ И ВЗАИМОДЕЙСТВИЕ С ПОТРЕБИТЕЛЯ

4.1. Web Push API

Web Push API представлява уеб интерфейс, който позволява на уеб приложенията да изпращат push известия към потребителите, дори когато приложението не е активно или браузърът не е отворен. Тази технология е основен компонент на прогресивните уеб приложения (PWA) и допринася за повишаване на ангажираността на потребителите чрез навременни и контекстуални известия.

Работата на Web Push API се осъществява в тясна връзка със Service Workers, които получават push съобщенията във фонов режим и отговарят за тяхното обработване и визуализиране. За да получава push известия, потребителят трябва изрично да даде своето съгласие, което гарантира защита на личните данни и контрол върху получаваната информация. Съобщенията се изпращат от сървър чрез push услуга, предоставяна от браузъра, и достигат до клиента дори при затворено приложение.

Web Push API е широко поддържан от съвременните браузъри, макар и с определени ограничения в зависимост от платформата и операционната система. В комбинация с останалите PWA технологии, push известията позволяват на уеб приложенията да се доближат по функционалност до нативните приложения, като осигуряват постоянна връзка между приложението и потребителя.

4.2. Сценарии за използване и ограничения

Типични сценарии за използване на push известия включват уведомления за нови съобщения, актуализации на поръчки, напомняния, новини или промоционални кампании. Те са особено ефективни при приложения, които разчитат на редовна ангажираност на потребителя, като комуникационни платформи, електронна търговия и информационни системи. При правилно използване push известията могат значително да подобрят потребителското изживяване и да увеличат честотата на взаимодействие с приложението.

Въпреки предимствата си, push известията имат и съществени ограничения. Те изискват изрично съгласие от страна на потребителя, а прекомерната или неподходяща употреба може да доведе до отхвърляне на разрешенията или до негативно възприемане на приложението. Освен това поддръжката на push известия варира в зависимост от браузъра и операционната система, като повечето платформи поддържат само част от свойствата и възможностите на Push Notification интерфейса. Някои разширени функционалности, като определени типове действия, визуални елементи или поведение на известията, може да не са налични или да се държат различно в различни среди.

Поради тези различия ефективното използване на push известия изисква внимателен подбор на сценарии и балансиран подход, съобразен както с техническите ограничения на платформите, така и с очакванията и удобството на потребителите.

ЗАКЛЮЧЕНИЕ

Just use React Native IDK. I will probably get sent to die on the front lines for Israel or something anyways, so what is the point.

If you're asking for my genuine opinion, the progression towards web just promotes lazy development. Like I get it, sure, you can push out binaries for more platforms faster and yes, ~~chances are users won't mind, since their devices are most likely powerful enough to run them~~, but that doesn't mean it's OK.

I remember reading it somewhere, due to a memory leak in Electron, which I would consider „patient zero“ of this phenomenon, Discord's PC client would have to restart if it went over 4 GB of RAM, which would technically free up the un-`free()`'d memory. It goes without saying, creating a custom client in C for example would have prevented this. Discord client most certainly doesn't need any platform specific libraries, they could've gotten away with using GTK, Qt or even OpenGL.

Even worse, this „PWA“ clusterf[redacted]ck even creeped into OS-es, too. The startbar on Windows 11 is apparently written in React Native. And most horrifying – even Linux has been caught up in this mess. I'll give you a hint – ask Miroslav what his Hyprland config is written in. Ironically enough it's just MacOS „holding the front“.

It's just like the clusterfuck happening in gaming. Barely any developer optimizes their games and instead tell you to just use DLSS or FSR.

This is not progress. I mean for hardware it is, sure, but most certainly not for software. The developer of Rollercoaster Tycoon wrote the game in x86 assembly just so it could run on a super minimal Intel i386 setup (don't quote me on the exact processor model, it doesn't matter).

I hope you liked this summary. I know it almost has nothing to do with what I wrote in the actual chapters, but I don't care.

ИЗПОЛЗВАНИ ИЗТОЧНИЦИ

1. Mozilla foundation (2025). ServiceWorker API. Достъпно към 20.12.2025г. от https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API
2. Vercel (2025). How to build a Progressive Web Application (PWA) with Next.js. Достъпно към 19.12.2025г. от https://nextjs.org/docs/app/guides/progressive-web-apps
3. Wikimedia (2025). Progressive web app. Достъпно към 15.12.2025г. от https://en.wikipedia.org/wiki/Progressive_web_app
4. Google (2025). Angular service worker overview. Достъпно към 20.12.2025г. от https://angular.dev/ecosystem/service-workers
5. Google (2025). Progressive Web Apps. Достъпно към 19.12.2025г. от https://web.dev/explore/progressive-web-apps
6. And let's not lie to each other I used a f[redacted]k-ton of ChatGPT to properly phrase this

ПРИЛОЖЕНИЕ 1

```
const cacheName = 'offlineCacheV1';

const offlineAssets = [
     '/offline/offline.html',
     '/offline/style.css',
     '/languages/en.json',
     '/languages/bg.json'
     // more files to cache
];
// create new cache
self.addEventListener('install', event => {
    event.waitUntil(
        caches
            .open(cacheName)
            .then(cache => cache.addAll(offlineAssets))
            .then(() => self.skipWaiting())
    );
});
// delete previous caches
self.addEventListener('activate', event => {
    event.waitUntil(
        caches.keys().then(cacheNames =>
            Promise.all(
                cacheNames.map(cache => {
                    if (cache !== cacheName)
                        return caches.delete(cache);
                })
            )
        )
    );
});
```

```
self.addEventListener('fetch', event => {
     const url = new URL(e.request.url);
     event.respondWith(
          caches.match(event.request).then(response => {
               if (response) return response;

               return fetch(e.request)
                    .then((res) => {
                         caches.open(cacheName)
                              .then((cache) =>
                              cache.put(
                                   event.request,
                                   res.clone()
                              ));
                         return res;
                    })
                    .catch(() => {
                         event
                              .request
                              .headers
                              .get('accept')
                              .includes('text/html')
                         ?    caches.match(
                                   '/offline/offline.html'
                              )
                         :    new Response(
                                   null,
                                   { status: 404 }
                              );
                    });
          })
     );
});
```