Програмиране
за мобилни устройства

Росен Иванов

2022

Книгата е предназначена за читателите, които искат на научат как се разработват *хибридни мобилни приложения*. Тя може да се използва и от студентите от специалност „Софтуерно и Компютърно Инженерство”, които изучават дисциплината „Програмиране за мобилни устройства” и от студентите от специалност „Компютърни Системи и Технологии”, които изучават дисциплината „Мобилни Интернет Системи”. Целта на книгата е да запознае читателите със съвременните тенденции в областта на създаване на мобилни услуги и приложения. След завършване на курса на обучение читателите трябва да могат да създават хибридни мобилни приложения и услуги за мобилни операционни системи Android и iOS, както и прогресивни Web приложения.

Читателите трябва да имат начални знания за HTML5, CSS3 и JavaScript. Темите са разработени достатъчно подробно, за да могат тези от тях, които нямат достатъчни знания в областта на Web програмирането успешно да реализират поставените задачи. За разработването и изграждането на мобилни хибридни приложения се използва облачната платформа Monaca и програмна рамка Framework7. След изучаване на материала в тази книга, читателите ще могат:

- Да създават приложения за операционни системи Android и iOS с нативен потребителски интерфейс.
- Да създават прогресивни Web приложения (PWA).
- Да създават мобилни Single Page Applications (SPA).
- Да обработват събития, генерирани от компонентите на потребителския интерфейс, както и да създават собствени събития.
- Да създават динамично съдържание чрез използване на документи-шаблони.
- Да реализират приложения, които използват вградените в мобилните устройства апаратни модули като акселерометър, компас, GPS приемник, високоговорител и вибратор.
- Да реализират асинхронни комуникационни канали с облачно базирани услуги.
- Да създават приложения с достъп до облачно базирани NoSQL бази данни.
- Да създават приложения, които получават и изпращат push известия.

Книгата може да се използва и от студентите от други специалности, които имат начални знания в областта на Web програмирането. Всички примери от книгата са достъпни за безплатно сваляне от адрес http://kst.tugab.bg/mobile.

Програмиране за мобилни устройства

ISBN 978-954-683-665-6

2022

СЪДЪРЖАНИЕ

Глава 1

Мобилни приложения

I. Въведение

Мобилните приложения се стартират под управлението на мобилна операционна система (ОС). За последните няколко години мобилните ОС Android (Google) и iOS (Apple) заемат над 95% от пазарния дял. По данни на StatCounter за декември 2021 г. пазарният дял на мобилните устройства с Android е 70.01%, а на iOS – 29.24%. Над 75% от мобилните устройства са с Android версия 9+ (35.37% с версия 11.0, 27.02% с 10.0 и 13.46% с 9.0 Pie). Над 62% от мобилните устройства са с iOS версии 14.8 и 15.1 (42.56% с версия 15.1 и 20.18% с 14.8). През 2021 г. мобилните приложения са генерирали приходи от около 693 млрд. долара чрез магазини за приложения и реклама в приложенията.

С увеличаването на употребата на мобилни устройства сред хората по целия свят, нуждата от мобилни приложения нараства бързо. Преминаването от десктоп приложения към мобилни приложения не е случайно. Мобилните приложения са по-бърз начин за достигане до по-широки и разнородни аудитории. Има различни аспекти, свързани с разработването на мобилни приложения, които обуславят успеха, например: възможност за много по-лесно разработване на мобилни приложения чрез използване на нови програмни рамки и езици за програмиране, облачно базирани платформи за развойна дейност, лесен достъп и видимост на приложенията чрез магазините за приложения (App Store за iOS и Google Play за Android) и възможностите за оптимизация на класацията на приложенията в тези магазини – App Store Optimization (ASO).

Съществуват два вида мобилни приложения в зависимост от технологиите, използвани с цел създаването им:

- *Нативни мобилни приложения*. Тези приложения са специално разработени да работят под управлението на конкретна мобилна ОС. Те имат пряк достъп до апаратните и програмни ресурси. Това предполага максимална функционалност, както бърза и надеждна работа.
- *Хибридни мобилни приложения*. Това са мулти-платформени приложения, които могат да работят под управлението на множество мобилни ОС. За целта за разработването им се използват Web технологии (HTML5, CSS3 и JavaScript) или специфични програмни езици и рамки, които гарантират платформената независимост. Хибридните приложения имат ограничен достъп до апаратните ресурси. Този достъп се реализира или чрез плъгини или чрез нативни функции, част от съответната хибридна платформа. Ако е необходимо, може да се симулира нативния интерфейс за избрана мобилна операционна система. Това се реализира чрез библиотеки за изграждане на потребителския интерфейс. Тъй като те се базират основно на JavaScript, хибридните приложения имат по-ниска производителност и време за реакция на потребителския интерфейс.

Трябва да се отбележи, че както нативната, така и крос-платформената разработка на приложения имат значителен пазарен дял. Кой тип приложение ще предпочетете зависи изцяло от това какво е предназначението на приложението; с кои мобилни операционни

системи трябва да работи то; до какви ресурси трябва да има достъп; изисква ли бърза обработка на данни от страна на клиента; какво ниво на защита трябва да се гарантира и др. В Табл. 1-1 е направен сравнителен анализ на нативните и хибридни приложения на базата на конкретни параметри.

Табл. 1-1 Сравнителен анализ на нативни и хибридни мобилни приложения

Параметър	Тип на приложението	
	Нативно	Хибридно
Произво-дителност	**Висока** Тъй като приложението е разработено специално за конкретна платформа, то ще има оптимална скорост и ефективност. Предполага се по-бързо зареждане и по-бърз потребителски интерфейс.	**Средна до висока** Програмният код на хибридните приложения трябва да се конвертира, когато е необходим достъп до апаратните ресурси. Поради тази причина често хибридните приложения работят по-бавно от нативните.
Дизайн	**Специфичен за ОС** Нативните приложения използват заложения дизайн на потребителския интерфейс за всяка операционна система. Това може да е както предимство, така и недостатък, ако трябва да се постигне специфичен дизайн.	**Специфичен за всяко приложение** Програмистът лесно може да получи специфичен дизайн на потребителския интерфейс. Ако е необходимо, може да се симулира дизайна на нативните приложения чрез използване на програмни рамки като React Native и Framework7.
Цена на разработката	**Висока** Цената за разработка е висока, тъй като са нужни експерти във всяка специфична за дадена ОС технология и програмен език. Времето, необходимо за разработване на нативно приложения за няколко ОС може да достигне и до над година в зависимост от сложността на приложението.	**Ниска до средна** При хибридните приложения се използват основно Web технологии, независимо с коя операционна система трябва да работи приложението. Поради това времето за разработка е много по-кратко и времето за пускане на пазара е по-бързо.
Програмни езици	**Специфични за операционната система програмни езици** Нативните приложения за Android се пишат на Java Android или Kotlin, а приложенията за iOS – на Objective C или Swift. Тези програмни езици изискват дълъг период на обучение, не по-кратък от година.	**Основно Web технологии** Повечето програмни рамки, които се използват за създаване на хибридни приложения се базират на Web технологии. Следователно, на разработчиците ще е необходимо по-малко време за обучение. Това не винаги е така, например Flutter използва Dart, който не толкова популярен език за програмиране.
Актуализация на програмния код	**Изисква повече време** Когато се налага актуализация на едно нативно приложение, екипите трябва да разработят отделни актуализации за всяка платформа.	**Изисква по-малко време** Екипът може да актуализира приложението за всички платформи наведнъж. Това прави поддръжката много по-лесна и съкращава времето, необходимо за решаване на проблеми.
Интеграция и достъпност	**Висока** Директен достъп до нативните функции на устройствата. Когато операционната система се актуализира, нативното приложение може лесно да се адаптира	**Ниска до средна** Хибридните приложения имат по-труден достъп до нативните функции на устройствата. Освен това обикновено има проблеми с интеграцията, а крос-платформената разработка често включва

	към нея.	използването на инструменти на трети страни за постигане на определени резултати.

Нативните приложения се разработват единствено за присъщите им платформи, например iOS или Android. Нативните приложения за Android се пишат на Java за Android или Kotlin и се компилират чрез AndroidStudio. Нативните приложения за iOS се пишат на Objective-C или Swift и се компилират с Xcode. Повечето фирми искат приложението им да е достъпно за няколко платформи, за да си гарантират по-голяма аудитория. Това означава, че трябва да се работи с два или повече екипи, които да разработят приложението, и повече дизайнери, които да приспособят дизайна към конкретна платформата. Но това също така означава, че приложението ще работи безпроблемно на платформите за които е разработено. Нативните приложения са бързи, но в същото време и скъпи. Приложението, написано за една платформа, не може да се използва от друга. Например, приложение, написано на Kotlin, не може да работи на iOS платформи и обратно.

Разработването на хибридни приложения може да се определи като сливане на разработките на нативни и Web приложения, откъдето идва и името хибридни. Тези приложения се разработват с помощта на Web технологии, включително HTML5, CSS3 и JavaScript. Те са по-лесни и бързи за създаване от нативните. Няколко предимства на хибридната разработка на мобилни приложения включват спестяване на разходи и един базов програмен код, чрез който можете да разработвате приложения за няколко платформи. Най-важната причина за приемането от бизнеса на хибридната разработка на мобилни приложения е, че тя е бюджетно ориентирана, тъй като разходите за нея са само наполовина от тези за разработката на нативни приложения. Хибридните технологии гарантират добра мащабируемост, тъй като след приключване на разработката приложенията могат да се използват на различни платформи. При разработката на хибридни приложения няма нужда от одобрение за публикуване, тъй като можете да имате пълен достъп до всички автоматични актуализации. Потребителите имат достъп до хибридните приложения дори в офлайн режим и това е основната характеристика на хибридните приложения (Offline-first Apps). Хибридните мобилни приложения са такъв вид приложения, които се инсталират на мобилно устройство и се разполагат в нативен контейнер, който най-често използва мобилен обект WebView.

При разработката на хибридни приложения е много важно да се избере подходящата хибридна програмна рамка и технологии.

II. Технологии за разработка на хибридни приложения

Когато става въпрос за разработване на мобилно приложение е важно да се избере правилния *технологичен стек*. Това ще повлияе на целия жизнен цикъл на разработвания продукт и неговата крайна цена. При хибридна разработка може да се избира между множество технологии, всяка от които има своите плюсове и минуси. В момента има няколко хибридни програмни рамки, които превъзхождат останалите по популярност. Това са *React Native*, *Flutter* и *Ionic*. Когато се избира хибридна програмна рамка трябва да се отчитат и други характеристики, освен популярността, например: изисква ли се заплащане за използваните технологии от програмния стек; какви инсталации са необходими да се направят и има ли специфични изисквания към апаратната част и операционната система; има ли налична развойна среда (IDE) или не; налага ли се работа с команди от конзолата или не; компилирането на приложенията

налага ли използването на специфично програмно осигуряване; има ли възможност за емулиране на приложението на виртуално и реално устройства; автоматизиран или не е процесът на създаване на програмен пакет за публикуване в съответния магазин за приложения и др.

Следва анализ на най-често използваните хибридни програмни рамки и технологии, използване с цел създаване на хибридни мобилни приложения.

2.1 ReactNative

Програмната рамка ReactNative е разработена от Facebook през 2015 г. Първата стабилна компилация беше пусната през 2019 г. Тя е с отворен код, така че няма нужда от лицензиране и притеснения за проблеми с авторските права. По-важните ѝ характеристики са следните:

- Език за програмиране: JavaScript
- Поддържани платформи: iOS и Android
- Време за разработка: средно.
- Разходи за разработка: средни.
- Популярни приложения, разработени с тази рамка: Facebook, Instagram, Skype.
- Уебсайт: https://facebook.github.io/react-native/

Основната причина за високата популярност на ReactNative е използването на JavaScript като език за програмиране. Това прави ReactNative по-лесен за научаване и използване. Ресурсите, създадени през годините за React са от голяма полза за екипите, работещи с React Native. Изграждането на приложение чрез ReactNative обаче изисква известно количество нативен програмен код. Някои функции трябва да бъдат адаптирани към конкретна платформа, като се използва нативния за конкретната платформа език за програмиране. Това означава, че разходите за разработване на ReactNative приложения може да не са толкова ниски, колкото при някои други хибридни програмни рамки, тъй като са необходими експертни познания и в областта на нативния програмен код.

Един от най-добрите случаи, в които React Native може да се прояви е когато имате вече съществуващо Web приложение и искате да разширите достъпа до потенциалните клиенти чрез мобилно приложение. Заради JavaScript езика за програмиране вашият екип ще може да използва повторно части от програмния код на Web приложението при изграждането на мобилното приложение. Това ще направи разработката много по-бърза. Освен това, ако вече разполагате със собствен екип за разработка на Web приложения, използването на ReactNative не би трябвало да е твърде трудно за тях, така че няма нужда от допълнителни програмисти.

Рамката ReactNative използва мост за свързване на кода на JavaScript и нативния код. Това е начинът, по който ReactNative приложенията работят. Този подход гарантира платформена независимост, но и намалява производителността на приложенията. ReactNative е подходящата платформа за бърза разработка на не много сложни приложения.

ReactNative позволява да публикувате актуализации много бързо. Това е особено полезно за приложения, които ще се променят често, така че екипът не трябва да чака толкова дълго между различните актуализации.

Предимства на ReactNative:

- Използва JavaScript, който е лесен за изучаване език и има силната подкрепа на огромна общност от разработчици.

- ReactNative използва много различни готови за вграждане компоненти, които помагат за ускоряване на целия процес на разработка.
- Лесно откриване на всички видове грешки.
- Спестяване на разходи с максимална повторна употреба на кода.
- Стартиране на нативен програмен код.
- Презареждане в реално време с два екрана (LiveReload) - един за промяна на кода и втори за преглед на промените

Недостатъци на ReactNative:

- Документацията не е достатъчно подробна.
- Ограничени възможности за интегриране на код от трети страни.
- Проблеми със съвместимостта и стабилността на работа на приложенията.
- Не е най-доброто решение за мобилни приложения с множество преходи между изгледи на потребителския интерфейс.
- Нуждае се от подобрение при достъпа до хардуерни компоненти.
- Проблеми с някои навигационни компоненти.

2.2 Flutter

Програмната рамка Flutter се поддържа от Google и е сравнително нова рамка - официално е пусната през 2018 г. Въпреки това, Flutter има значителна популярност и вече се счита за основна алтернатива на другите хибридни програмни рамки. По-важните характеристики на тази програмна рамка са следните:

- Език за програмиране: Dart (комбинация от Java и Kotlin).
- Поддържани платформи: Android, iOS, Windows, Mac, Linux, Fuchsia, Webt.
- Време за разработка: кратко.
- Разходи за разработка: ниски.
- Популярни приложения разработени с тази рамка: Reflectly, GoogleAds, CoachYourself, Xianyu (Alibaba).
- Уебсайт: https://flutter.dev/

Това, което прави Flutter предпочитана програмна рамка, е неговата гъвкавост. Позволява лесно разработване дори на най-сложните потребителски интерфейси. За разлика от ReactNative, тази рамка не използва мост, за да комуникира с нативните компоненти на конкретна платформа. Това гарантира на Flutter приложенията много бърза реакция и отлична производителност. Приложенията, създадени с Flutter, имат потребителски интерфейс, който изглежда по един и същи начин на всички версии на дадена операционна система. Това означава, че дори потребителят да не е актуализирал устройството си, той ще получи същото изживяване като този, който е актуализирал своята операционна система.

Едно от основните предимства на Flutter е, че поддържа горещо презареждане на програмния код - *Hot Reload*. Това означава, че промените, направени в програмния код, незабавно се отразяват в приложението, без да е необходимо неговото рестартиране (*Live Reload*). По този начин лесно се проверява резултата след промяна на програмния код и се съкращава времето необходимо мобилното приложение да достигне до пазара. Програмната рамка Flutter може да се използва за разработка за всички най-разпространени популярни мобилни платформи, както и за разработка на Web приложения.

Предимства на Flutter:

- Бърз и надежден потребителски интерфейс.
- Горещо презареждане на програмни код.
- Съвместим с ОС Fuchsia на Google.
- Има много добра съвместимост с множество операционни системи.
- Предлага CLI и VI редактори.
- Лесна разработка чрез AndroidStudio.

Недостатъци на Flutter:

- Приложенията, създадени чрез Flutter са с около 40% по-големи по размер от нативния вариант на приложенията (минимум 4 MB).
- Необходимо е изучаване на програмния език Dart.
- Все още ограничена общност и малко експерти.
- Много висока цена на бизнес лицензa.

2.3 Ionic

Програмната рамка Ionic е официално пусната на пазара през 2013 г. Първоначално тя е базирана на AngularJS. След версия 5 може да използвате програмни рамки React и Vue за създаване на потребителския интерфейс. По-важните характеристики на тази програмна рамка са следните:

- Език за програмиране: AngularJS, JavaScript, TypeScript.
- Поддържани платформи: Android, iOS, Windows, Blackberry, Web.
- Време за разработка: кратко.
- Разходи за разработка: ниски.
- Популярни приложения: MarketWatch, Diesel, Pacifica, Sworkit.
- Уебсайт: https://ionicframework.com/

Изучаването на Ionic е сравнително лесно, тъй като се базира на JavaScript и AngularJS, които вече са достатъчно популярни. Освен това има много голяма общност, която използва Ionic. До момента с Ionic са създадени над 5 милиона приложения. Програмна рамка Ionic предлага множество инструменти за създаване на отлични програмни интерфейси, които работят безпроблемно на всяка платформа. Тя е сравнително бърза рамка за разработка на хибридни приложения. Позволява създаването и на прогресивни Web приложения – Progressive Web Apps (PWA). Прогресивните приложения са Web приложения, които могат да бъдат инсталирани на всяка платформа (която ги поддържа) и могат да работят и без наличие на мрежова свързаност (Offline-first Apps).

Ionic Command Line Interface (CLI) е инструмент чрез който се реализира разработката на Ionic проекти. Командите на Ionic CLI могат да бъдат стартирани от командния ред или от терминала на развойна среда като Microsoft Visual Code. Ionic CLI предоставя вграден сървър за разработка и тестване, както и инструменти за отстраняване на грешки.

Подобните на native компоненти за различни платформи позволяват лесно да се осигури страхотно потребителско изживяване. Предлага много добра вградена поддръжка за MaterialDesign за Android. Предоставя възможност за достъп до апаратната част чрез Cordova или Capacitor плъгини. Тези плъгини са лесни за включване и позволяват на

хибридното приложение да получи достъп до апаратни ресурси като камера, микрофон, файлова, батерия, Bluetooth и др.

Предимства на Ionic:

- Възможност за работа с AngularJS, React или Vue.
- Лесна актуализация.
- Множество компоненти за потребителския интерфейс.
- Предварително генерирани настройки на приложенията с креативни оформления
- Популярна програмна рамка.

Недостатъци на Ionic:

- Няма вградена функционалност за компилиране на програмния код.
- Не поддържа горещо презареждане.
- Твърде много зависи от множество плъгини.
- Не е подходяща рамка за създаване на сложни мобилни приложения.

2.4 Xamarin

Програмната рамка Xamarin е пусната на пазара през 2011 г. От 2016 г. насам тя е собственост на Microsoft. Базира се на принципа "WriteOnce, RunAnytime". Xamarin е част от екосистема .NET на Microsoft, която се използва от милиони разработчици в световен мащаб. Общността, която поддържа Xamarin, наброява над 1.4 милиона разработчици. По-важните характеристики на тази програмна рамка са следните:

- Език на програмиране: C# .NET.
- Поддържани платформи: Android, iOS, tvOS, watchOS, macOS и Windows.
- Време за разработка: средно.
- Разходи за разработка: средни.
- Популярни приложения: Siemens, Pinterest, Olo, Storyo, Insightly, Fox Sports, Световната банка.
- Уебсайт: https://dotnet.microsoft.com/apps/xamarin/

Чрез програмната рамка Xamarin се създават приложения с много добра производителност, като същевременно рамката дава възможност на разработчиците да използват повторно около 75% от кода си за различни платформи. Позволява много добра поддръжка на операционните системи на Apple (с изключение на iPadOS). Това е една от малкото хибридни програмни рамки които позволяват разработка на приложения за *интелигентни часовници*. Освен това, подкрепата на Microsoft гарантира наличието на голяма общност и много споделени ресурси. Рамката е сравнително лесна за изучаване. Като собственост на Microsoft, програмната рамка Xamarin дава възможност за лесна интеграция в облака. Това е особено полезно при създаване на интелигентни услуги.

Xamarin позволява разработването на различни интерфейси за различни платформи, които използват един и същ back-end код на C#. Това означава, че Xamarin приложението може лесно да получи дизайн на потребителския интерфейс подобен на този на нативните приложения за съответната платформа.

Предимства на Xamarin:

- Много добра подкрепа от Microsoft – C#.NET, VisualStudio, Azure.
- Безплатна платформа за индивидуални разработчици.
- Собствена екосистема (XamarinStudio, XamarinSDK, XamarinTestCloud и др.).
- По-голямата част от кода (над 90%) може да се използва многократно при разработване на приложения за Android и iOS.
- Приложенията се свързват лесно с апаратните компоненти чрез приставки, без да пречат на работата на устройството.
- Много добра производителност.

Недостатъци на Xamarin:

- Висока цена при разработка на комерсиални приложения (лицензи за MS Visual Studio и Visual Studio Enterprise).
- Ограничен достъп до библиотеки с отворен код.
- Голям размер на приложенията при сравнение с нативните им варианти.
- Не е подходяща при тежък потребителски интерфейс.
- Сравнително малка общност при сравнение с другите хибридни мобилни рамки..

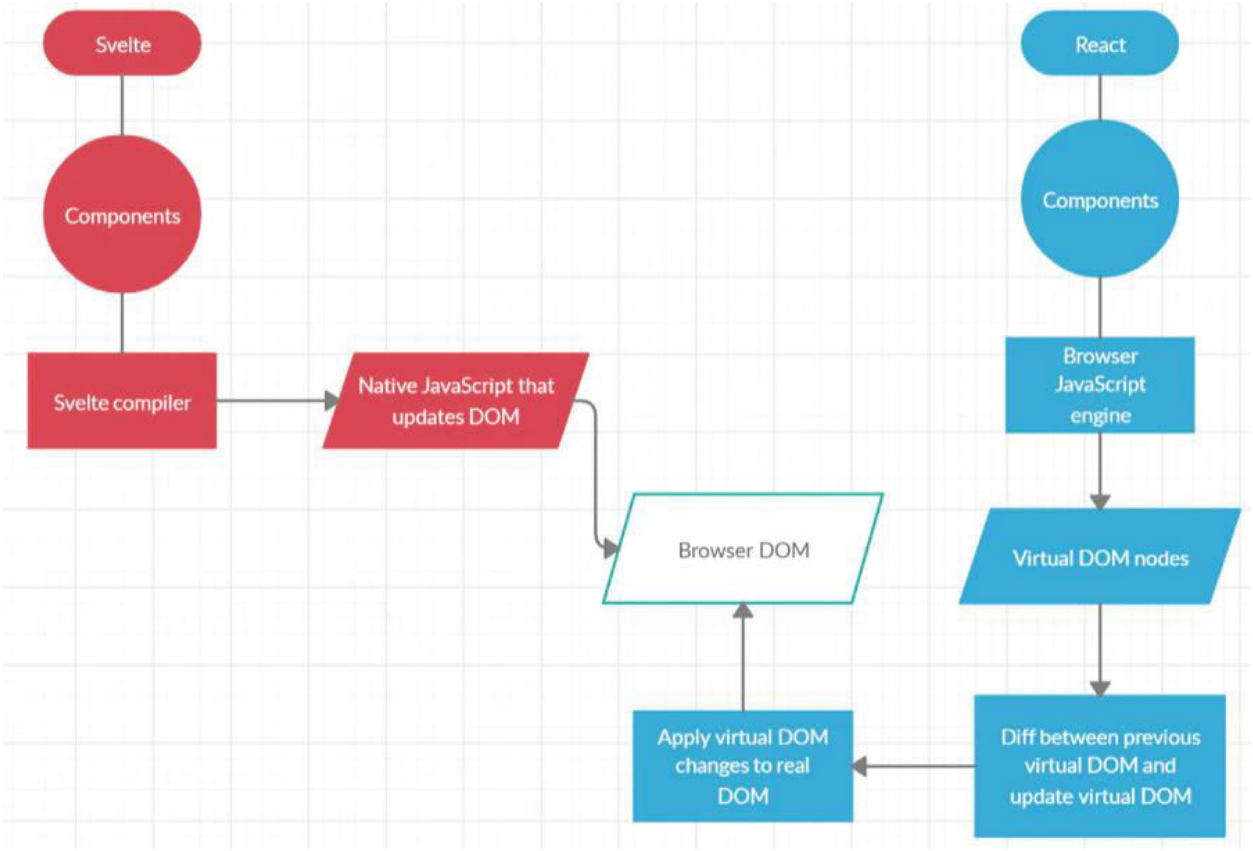
2.5 Framework7

Тази рамка е разработена през 2014 г. от руското студио iDangero. В основата на разработката е Владимир Харлампиди. Framework7 е безплатна и с отворен код рамка за разработка на хибридни приложения, която се използва за създаване на настолни, мобилни и Web базирани приложения с нативен потребителски интерфейс. Първоначално Framework7 се е състоял от набор от компоненти на потребителския интерфейс в стила на наскоро излязлата тогава iOS 7, откъдето получава името си. По-късно е добавена тема с цел симулиране на Material Design на Android. Поддръжката на Vue в Framework7 съществува още от версия 1, а от версия 3 се поддържа и React. Това прави рамката конкурентна с новите версии на Ionic.

При Framework7 v5.4.0+ може да се използва рамката Svelte - Framework7 Svelte. Рамката за компоненти Svelte е разработена от Рич Харис през 2016 г. Framework7 Svelte съчетава мощта и простотата на рамката Svelte с гъвкавостта и потребителския интерфейс на Framework7 с цел изграждане на мобилни приложения по още по-лесен и бърз начин. Тази нова рамка предлага радикално нов подход към изграждането на потребителския интерфейс. Докато рамки като React и Vue извършват по-голямата част от работата си в браузъра по време на работа на приложението, Svelte прехвърля тази работа на етап компилиране, която се извършва само при изграждане на приложението. Създава се силно оптимизиран чист JavaScript код (“vanilla” JavaScript). Вместо да използва техники като *виртуален* Domain Object Model (VDOM), Svelte генерира код, който актуализира DOM при промяна на състоянието на вашето приложение (виж Фиг. 1-1). По този начин приложенията стават по-бързи, с по-маък по размер и по-ясен код и с вградено управление на състоянието и реална реактивност. Резултатът от този подход е по-малък размер на програмните пакети, както и по-добра производителност. Тъй като е компилатор, Svelte може да разширява HTML, CSS и JavaScript, като генерира оптимален код на JavaScript без излишни разходи по време на изпълнение. За да постигне това, Svelte разширява Web технологиите по следните начини:

- Разширява HTML, като позволява използването на изрази на JavaScript и предоставя директиви за използване на условия и цикли по начин, подобен на този на работата с шаблони.

- Разширява CSS, като добавя механизъм за определяне на обхвата, позволяващ на всеки компонент да определя свои собствени стилове без риск от колизия със стиловете на други компоненти.
- Разширява JavaScript, като интерпретира специфични директиви на езика, за да постигне истинска реактивност и да улесни управлението на състоянието на компонентите.



Фиг. 1-1 Сравнение между Svelte DOM и React DOM

Компилаторът на Svelte се намесва само в много специфични ситуации и само в контекста на компонентите. Разширенията на езика JavaScript са минимални и внимателно подбрани, за да не нарушават синтаксиса на JavaScript и да не отблъскват разработчиците.

Framework7 Svelte помага да се съхраняват данните и съдържанието по по-структуриран начин, както и да се опрости разликата в оформлението за iOS и Material Design темите. Рамката Framework7 Svelte предоставя набор от почти всички елементи и компоненти на Framework7 с интеграция на Framework7 Router за визуализиране на страници по начина характерен за Svelte.

Предимства на Framework7:

- Лесна за научаване и внедряване програмна рамка.
- Съдържа много джаджи и компоненти.
- Вградени помощни библиотеки.

Недостатъци на Framework7:

- По-малко поддръжка от страна на онлайн общността.

- Средна по качество документация.

2.6 PhoneGap

Програмната рамка PhoneGap е комерсиалната версия на Apache Cordova. Apache Cordova е рамка за разработка на мобилни приложения, създадена от Nitobi през 2009 г. Adobe Systems купува Nitobi през 2011 г. и променя името й на PhoneGap. По-късно пуска версия на софтуера с отворен код, наречена Apache Cordova. Върху рамката Cordova са изградени и много други рамки, включително Ionic, Onsen UI, VoltBuilder, Framework7, Quasar Framework, както и Telerik Icenium. В проекта Apache Cordova участват Adobe, BlackBerry, Google, IBM, Intel, Microsoft, Mozilla и др. Програмната рамка PhoneGap е една от добрите рамки за писане на хибридни приложения, но и една от най-старите. Използват Web технологии - JavaScript, HTML5 и CSS3. PhoneGap осигурява надеждна междуплатформена съвместимост. Тя е добър избор, ако трябва да се разработват приложения с ограничен бюджет. Достъпът до системните ресурси на мобилното устройство се реализира чрез нативни плъгини. Те могат да бъдат викани от JavaScript код, като се осъществява директна комуникация между нативния слой и JavaScript. Тези плъгини позволяват достъп до камерата, акселерометъра, компаса, GPS сензора, файловата система, микрофона и др.

В края на 2020 г. фирма Adobe се отказа от поддръжката на PhoneGap, включително и на облачната платформа за компилиране PhoneGap Build. Най-добрата алтернатива на облачната платформа на Adobe е облачната платформа Monaca чрез която не се налага да инсталирате каквото е да е програмно осигуряване, за да може да пишете, тествате, компилирате и публикувате в магазините за приложения приложения за Android, iOS и Windows.

Предимства на PnoneGap:

- Съвместима с ОС Android, iOS и Windows.
- Сравнително бърза рамка за разработка, която може да създава приложения за много кратко време, лесна за изучаване.
- Лесно тестване на приложенията.
- Поддържа се от облачната платформа Monaca.

Недостатъци на PnoneGap:

- Липсват на джаджи за потребителския интерфейс.
- Не е достатъчно бърза в сравнение с други рамки.
- Няма подкрепа от страна на големите фирми.
- Много подходяща за прототипи на приложения.

III. Избор на програмна рамка

Много е трудно да се избере най-добрата програмна рамка за разработка на мобилни приложения. Причината за това е, че всяка рамка има както множество предимства, така и множество недостатъци. Пазарния дял на петте най-използвани хибридни програмни рамки за 2021 г. е следния:

- Flutter – 39%.

- ReactNative – 38%.
- Cordova – 16%.
- Ionic – 16%.
- Xamarin – 11%.

Решението какви технологии ще използва колектива, който ще разработва приложението, се взема основно от главния софтуерен дизайнер на фирмата. Тъй като използването на някои технологии изисква финансиране (закупуване и лицензни такси) в решението трябва да участват и главния техническия директор. Те заедно трябва да анализират комплексно проблема, като отчетат фирмените бизнес цели и възможни технологии, които ще доведат до постигане на тези цели при оптимизиране на зададена функция. Тази функция може да включва параметри като: време за разработване на приложението; финансова рамка; експертно ниво на колектива от програмисти; специфични изисквания на заявителя. Следва анализ на основните параметри от които зависи избора на технологичен стек чрез който ще бъде разработено приложението.

3.1 Време за разработка и финансова рамка

Едно от най-важните неща, които определят решението за избор на технологичен стек са времето и разходите за разработване на приложение с определен набор от технологии. Разработката на хибридни приложения обикновено е по-евтина и по-бърза от разработката на нативни приложения, независимо от това коя конкретна програмна рамка ще бъде избрана. При вземане на решение задължително трябва да се отчете и съгласуваното време за разработване на проекта. Ако срокове за разработка са кратки (няколко месеца), е най-добре да се разработи хибридно приложения с програмна рамка като React Native или Ionic.

3.2 Производителност

При необходимост от висока производителност трябва да се насочите към разработка на нативно приложение. Ако прецените, че колективът от програмисти няма да може да се справи с тази задача за зададения времеви срок, то трябва да се насочите към използване на хибридна програмна рамка като Flutter. В зависимост от сложността на вашето приложение трябва да прецените дали конкретна хибридна рамка ще може да гарантира необходимата функционалност. Ако производителността е най-важната функционалност за вашето приложение, то трябва да се ориентирате към разработване чрез нативни технологии.

3.3 Бизнес цели

Краткосрочните и дългосрочните бизнес цели на фирмата имат огромно влияние върху решението за избор на технологичен стек. Главният технически директор трябва да прецени каква да бъде бизнес рамката в зависимост от това какво и как точно се разработва. Ако трябва да се потвърди избраната концепцията може да се разработи бета версия или да се премине през създаване на минимален жизнеспособен продукт - Minimum Viable Product (MVP).

3.4 Интеграции и достъпност

В зависимост от вашата концепция може да се наложи приложението да използва функции като достъп до вградените сензори, интерфейси, състоянието на батерията и др. Достъпът до тях при използване на хибридните приложения обикновено е по-труден за реализация, отколкото ако използвате нативни приложения. Ако трябва да включите

подобна функционалност е по-добре е да използвате нативна разработка, защото тя ще гарантира безпроблемно използване на всички нативните функциите на устройството.

3.5 Потребителско изживяване

В повечето случаи нативните приложения осигуряват по-добро потребителско изживяване от хибридните. Това обаче не важи при използване на хибридна програмна рамка, която е специално разработена да гарантира нативен изглед и високо бързодействие на потребителския интерфейс, например Flutter и Framework7 Svelte. Оценката на потребителското изживяване се оценява най-често чрез анкетиране на избрана група от потребители.

3.6 Екип

Екипът, с който фирмата разполага, е нещо което трябва да се вземе предвид, когато се избира технологичния стек. Ако фирмата има опит и има главен софтуерен дизайнер и главен технически директор то се предполага, че проблеми с избора на подходящия стек няма да има. При по-малките фирми, това може да не е така. Ако все още фирмата няма експерти с опит в областта на дизайна на програмни проекти (опит над 5 години) е добре да се направи консултация с външна фирма. Тази инвестиция ще се изплати, тъй като няма да имате проблем с необходимостта от смяна на технологиите след като е започнала разработката на приложението.

Заключение

В тази глава се запознахте с различните видове мобилни приложения - нативни и хибридни - и с техните предимства и недостатъци. Разгледани бяха основните програмни рамки чрез които е възможно разработването и изграждането на хибридни мобилни приложения. Анализирани бяха основните критерии от които зависи правилния избор на програмна рамка.

Глава 2

Разработване на мобилни приложения

I. Въведение

Разработване на всеки програмен продукт е сложен процес, който включва не само изборът на технологичния стек и екип чрез които ще се реализира продукта, но и икономическите и социални рамки, които се задават от глобалния и локален пазар, възложителя на проекта и потенциалните потребители на продукта. Отчитането на всички финансови, програмни и социални ограничения е важно за създаването на програмен продукт, който достига до пазара възможно за най-кратко време при висок потенциал за възстановяване на вложените в него финансови ресурси в рамките на дефинирания предварително за това времеви интервал.

II. Фази на разработване на програмен продукт

Ефективният процес на разработване на приложения обхваща шест ключови фази. Независимо от размера и обхвата на вашия проект, следването на този процес на разработка ще направи вашата разработка успешна.

2.1 Дефиниране на вашата стратегия

Първият етап от процеса на разработване на мобилни приложения е определянето на стратегията за превръщането на вашата идея в успешно приложение.

В тази фаза трябва да:

- Идентифицирате потребителите на приложението.
- Проучите конкуренцията.
- Определите целите и задачите на приложението.
- Изберете подходящата технологична рамка.

2.2 Анализ и планиране

На този етап идеята ви за приложение започва да придобива форма и се превръща в реален проект. Анализът и планирането започват с дефиниране на случаи на употреба и улавяне на подробни функционални изисквания. След като сте определили изискванията за вашето приложение, подгответе *продуктова пътна карта*. Това включва подреждане по значимост на изискванията към мобилното приложение и групирането им в етапи на изпълнение. Ако времето, ресурсите, или разходите са проблем, тогава създайте „минимален жизнеспособен продукт”.

Част от фазата на планиране включва определяне на уменията на колектива, необходими за разработване на приложението. За целта се изисква избор на технологичния стек с който ще се работи.

На този етап трябва да изберете името на вашето приложение. Името трябва да е уникално за магазина за приложения от който то може да бъде инсталирано. Проучете всеки магазин за приложения, за да сте сигурни, че името на приложение вече не се използва. Изборът на име е важно и за класифицирането на приложението както и за оптимизацията на приложението в рамките на всеки магазин за приложения.

2.3 Дизайн на потребителския интерфейс

Успехът на едно мобилно приложение се определя въз основа на това колко добре потребителите приемат и се възползват от всички негови функции. Целта на дизайна на потребителския интерфейс на мобилни приложения е създаването на отлични потребителски изживявания, които да направят вашето приложение интерактивно, интуитивно и удобно за ползване. Не на последно място потребителите трябва да останат удовлетворени след използване на приложението. Макар че усъвършенстваният дизайн на потребителския интерфейс ще помогне за ранното приемане, вашето приложение трябва да има интуитивен потребителски интерфейс, за да задържи вниманието на потребителите.

Първата стъпка от процеса на проектиране на мобилно приложение е да определите данните, които мобилното приложение ще показва на потребителите, както и данните, които то ще събира, взаимодействията на потребителите с крайния продукт и *потребителските пътувания* в рамките на приложението (последователността през която потребителя преминава до достигане на желания резултат). Ако се налага потребителите да имат различни роли и привилегии е важно да включите тези правила като част от информационната архитектура на приложение. Диаграмите на работните потоци помагат да се определи всяко възможно взаимодействие на потребителя с приложението и навигационната структура на приложението.

2.3.1. Ръководство за стилово форматиране

Ръководството за стилово форматиране трябва да документира стандартите за дизайн на приложението - от правилата за брандиране на компанията до иконите за навигация. Ръководствата за стил включват и информация за използваните шрифтове, както и каква ще бъде основната цветова схема.

2.3.2. Получаване на прототип и минимален жизнеспособен продукт

Прототипът и *минималния жизнеспособен продукт* (MVP) са олекотена ранна версия на вашия продукт. Можете да визуализирате на хартия или цифрово чрез приложение за създаване и обработка на графика (Adobe Photoshop) как очаквате да изглежда потребителския интерфейс на приложението. Цифровата форма на скиците се наричат Wireframes. При създаването на wireframes трябва да вземете предвид специфичния за устройството дизайн.Тази задача се реализира най-често от графичен дизайнер. Основното предимство на прототипа е, че ще помогне за комуникацията с екипа, който трябва да реализира приложението. Колкото по-подробен е прототипа, толкова по-малко въпроси ще имат членовете на колектива в процеса на разработка. Прототипите са изключително полезни за симулиране на потребителското изживяване и работните процеси на приложението, които се очакват от крайния продукт. Макар че разработването на прототипи може да отнеме много време, усилията си заслужават, тъй като те предлагат тестване на ранен етап на дизайна и функционалността на приложението. За да съкратите времето за създаване на качествен прототип може да използвате специализирани продукти като *Invision* и *Figma*.

Следващата стъпка е да създадете така наречения „минимален жизнеспособен продукт”. Това е опростена версия на вашия продукт, която обаче решава проблема на вашите клиенти. Основната цел на MVP е да се разработи работоспособен продукт, който да достигне до пазара *бързо* и с *минимални разходи*. Чрез MVP екипът може да събере с най-малко усилия максимално количество знания за клиентите на приложението.

Минималният жизнеспособен продукт трябва да докаже приложимостта на избраната концепция. По същество това е начин за тестване на бизнес модела на приложението при минимална функционалност и минимално количество програмен код. MVP е минималистична форма на вашия продукт, която пазара трябва да тества. Тази стратегия за разработване позволява на екипа да потвърди (или отхвърли) предположенията за продукта и да научи как целевите ви потребители реагират и преживяват основната функционалност на продукта ви. Този подход ще ви даде представа за правилното разпределение на бюджета, за да задоволите общите си бизнес цели. Изграждането на MVP е итеративен процес, предназначен да идентифицира подходящата функционалност на продукта, която същевременно отговаря на нуждите на вашите потребители. Разработването на MVP следва принципа „изгради, измери, научи" (build-measure-learn), който позволява да се създаде продукт, който може да бъде итеративно подобряван на базата на потвърждаване или отхвърляне на предположения.

Основните стъпки през които се преминава при създаване на MVP са следните:

A. Идентифициране и разбиране на нуждите на вашия бизнес и пазар

Първата стъпка е да установите дали има нужда от вашия продукт на пазара. Това може да бъде нужда на фирма или нужда на група клиенти, която е насочена към текущ дефицит на пазара. Също така е важно да анализирате какво правят вашите конкуренти и да установите как можете да направите така, че вашият продукт да се откроява.

След като сте установили, че пазарът има нужда от вашия продукт, трябва да дефинирате вашата *дългосрочна бизнес цел* - какво планирате да постигнете? След това определете какво ще определи успеха на вашия продукт.

B. Отчитане на мнението на потребителите

Дефинирайте точно и еднозначно кои са вашите потенциални потребители. Възможно е да имате повече от една категория потребители. Добър начин да се уверите, че потребителите ви ще имат добро преживяване с първата итерация на вашето приложение, е като анализирате как те използват потребителския интерфейс - колко често се налага да достигат до помощна информация, колко време им е необходимо, за да постигнат поставена задача. Това ще ви позволи да погледнете на продукта си от гледна точка на потребителя, като започнете от отварянето на приложението и стигнете до крайната цел, например извършване на покупка. Това се нарича дизайн във фокуса на който е потребителя (user centered design). По този начин приложението ще бъде проектирано по начин, който е удобен за потребителите. Това със сигурност ще гарантира техната висока удовлетвореност.

C. Създаване на карта на „болката и печалбата"

След като сте разработили потребителския поток, ще трябва да създадете карта на „болката и печалбата" за всяко действие. Тази карта ви позволява да идентифицирате всички болезнени точки на потребителя и печалбите, които той получава, когато всяка от тях бъде преодоляна. Тази тактика ще ви позволи да определите къде имате най-голям потенциал да добавите стойност към вашия проект.

D. Решете какви функции да включите

На този етап ще можете да прецените кои функции да включите във вашия MVP, както и кои функции да включите в пътната карта на продукта. Задаването на въпроса какво иска потребителя спрямо това, от което се нуждае, може да помогне за идентифициране и

приоритизиране на функциите. Имайте предвид, че внедряването на твърде много функции, поискани от потребителите, твърде скоро може да навреди на потребителското изживяване и да се отклоните от основната цел на приложението. Единствените функции, които трябва да включите, трябва да са свързани с основната цел на продукта.

2.4 Програмна реализация

Преди да започнат действителните усилия, свързани с програмиране, трябва:

- Да се избере архитектурата на приложението.
- Да се избере подходящия технологичен стек.
- Да се дефинират основните етапи на разработката.

По-голяма част от мобилните услуги се състоят от три неразделни части: front-end, back-end и приложни програмни интерфейси (API). Front-end частта е самото мобилното приложение, което потребителят ще използва, за да получи желаната функционалност. Желателно е приложението да може да функционира и без наличие на мрежова свързаност за определен интервал от време (Offline-first App). За целта приложението трябва да използва локална база данни, която се синхронизира с базата данни от back-end частта при възстановяване на мрежовата свързаност. За реализацията на бизнес логиката и управление на базите данни отговаря back-end частта. Възможно е създаването и на услуга, която не изисква back-end код - *serverless* приложения. В този случай базата данни най-често е хоствана в някаква облачна инфраструктура, достъпът до която се реализира чрез интерфейс HTTPS. Комуникацията между мобилното приложение и back-end програмния код се реализира чрез приложни програмни интефейси.

След като разработката на програмния код завърши се преминава към тестване на приложението.

2.5 Тестване

Извършването на задълбочени тестове за осигуряване на качеството (QA) по време на процеса на разработване на мобилни приложения прави приложенията стабилни, използваеми и сигурни. За да осигурите изчерпателно тестване на осигуряването на качеството на вашето приложение, първо трябва да подготвите тестови скриптове, които обхващат всички аспекти на тестването на приложението.

Всяко приложение трябва да бъде подложено на следните методи за тестване, за да се гарантира качеството на продукта:

2.5.1. Тестване на потребителското изживяване

Критична стъпка в тестването на мобилни приложения е да се гарантира, че крайното изпълнение съответства на потребителското изживяване, желано от екипа по проектиране на приложението. Визуализацията, работният процес и интерактивността на вашето приложение са тези, които ще дадат на крайните потребители първото впечатление за вашето приложение. Ако приложението ви отговаря на първоначалните насоки за дизайн, то ще има пряко въздействие върху приемането му от потребителите.

2.5.2. Функционално тестване

Точността на функционалността на вашето мобилно приложение е от решаващо значение за неговия успех. Трудно е да се предвиди поведението и сценарият на използване за всеки краен потребител. Функционалността на вашето приложение трябва

да бъде тествана от колкото се може повече потребители, за да се обхванат възможно най-много потенциални сценарии на използване. Целта на функционалното тестване е да гарантира, че потребителите могат да използват функционалността на вашето приложение без проблеми. То може да се раздели още на системно тестване (приложението работи като цяло) и тестване на единици (отделните функции на приложението работят правилно). Ако създавате приложение за различни мобилните ОС, функционалното тестване трябва да включва и сравнение на функциите на всички варианти на мобилното приложение.

2.5.3. Тестване на производителността

Съществуват много количествени критерии, които можете да използвате за измерване на производителността на едно приложение:

- Колко точно и бързо приложението отговаря на заявките на потребителите?
- Колко бързо се зареждат екраните на приложението?
- Има ли изтичане на памет?
- За колко време приложението изчерпва капацитета на батерията на телефона?

Дори когато приложението ви отговаря на основните критерии за производителност, тествайте като симулирате максимален брой едновременно работещи потребители. Приложението ви трябва да може да се справи с натоварването и да работи добре дори при рязко увеличаване на неговото използване.

2.5.4. Тестване на сигурността

Сигурността е от изключителна важност за корпоративните мобилни приложения. Всяка потенциална уязвимост може да доведе до хакерска атака. Много компании наемат външни фирми, които да извършат задълбочено тестване на сигурността на техните приложения.

Ако приложението ви изисква от потребителите да влизат в системата след авторизация, тези сесии за влизане трябва да бъдат проследявани на устройството и в бекенда. Потребителските сесии трябва да се прекратяват от системата, когато потребителят е останал неактивен продължително време (обикновено десет минути или по-малко за мобилно приложение). Ако приложението ви съхранява идентификационните данни на потребителя на устройството, за да му е удобно да влезе отново, трябва да осигурите използването на надеждна услуга. Например, платформата за разработване на приложения за iOS предоставя функцията Keychain, която може да се използва за съхраняване на данните за профила на потребителя.

2.5.4. Тестване на реални устройства

Средно на всеки 12 месеца на пазара излизат нови мобилни устройства с нов хардуер, фирмен софтуер и дизайн. Мобилните операционни системи се актуализират на всеки няколко месеца. Това налага перманентно тестване на приложението на реални устройства с цел навременно разпознаване на проблеми с новия фирмен софтуер и версия на операционната система, за да се гарантира безпроблемната работа на приложението ви за всички потребители.

2.6 Внедряване и поддръжка

След като сте идентифицирали нуждите на вашия бизнес и потребители можете да се съсредоточите върху пускането на пазара на вашия „минимален жизнеспособен продукт”.

След пускането на приложението на пазара е задължително да съберете обратна връзка от потребителите с цел *пазарна валидация*. Това ще ви помогне да генерирате нови идеи, обосновани с проучване на поведението на потребителите, които ще оформят следващите версии на приложението. Подобряването на приложението продължава до получаване на неговата финална версия.

Пускането на едно мобилно приложение изисква изпращане на приложението в магазините за приложения - App Store при iOS и Google Play за Android. Трябва да имате предвид, че ще ви е необходима регистация за разработчик в Apple App Store и Google Play Store. Пускането на приложението в магазина за приложения изисква подготовка на множество метаданни за него, например:

- Заглавие на вашето приложение.
- Описание.
- Категория.
- Ключови думи.
- Икона.
- Снимки за магазина за приложения.

След като бъдат подадени в Apple App Store, приложенията за iOS преминават през процес на преглед, който може да отнеме от няколко дни до няколко седмици в зависимост от качеството на вашето приложение и от това доколко то следва насоките за разработване на iOS. Ако приложението ви изисква от потребителите да влизат в системата, ще трябва да предоставите на Apple тестови потребителски акаунт като част от процеса на пускане. При приложенията за Android няма процес на преглед и те стават достъпни в магазина за приложения в рамките на няколко часа след подаването им. За разлика от Web приложенията, при които чрез отстраняването на грешки чрез пачове могат да бъдат достъпни за потребителите на приложението незабавно, актуализациите на мобилните приложения трябва да преминат през същия процес на подаване и преглед, както първоначалното.

След като приложението ви стане достъпно в магазините за приложения, наблюдавайте използването му чрез платформи за мобилни анализи и следете ключовите показатели за ефективност за измерване на успеха на приложението. Често проверявайте докладите за сривове или други проблеми, докладвани от потребителите.

Заключение

В тази глава се запознахте с основните фази през които се преминава до получаване на работоспособно мобилно приложение. Описано бе какво е „минимален жизнеспособен продукт” и каква е неговата ключова роля когато приложението трябва да достигне до потребителите си бързо и с минимални разходи. Описана бе ролята на тестването на програмните продукти, както и различните видове тестове: потребителско изживяване, функционално тестване, производителност и сигурност.

Глава 3

Web технологии

I. Въведение

Създаването на Web услуги предполага използването на множество технологии. Една част от тях се използват с цел изграждане на потребителския интерфейс и генериране на заявки към мрежови или облачни услуги (front-end програмиране), а друга част се използват с цел обслужване на тези заявки и реализиране на бизнес логиката на услугата (back-end програмиране). Основните технологии, чрез които се реализира front-end частта от услугата, са HTML, CSS и JavaScript. Тези технологии функционират под управлението на Web браузър. Тяхното предназначение е както следва:

1. Hyper Text Markup Language (HTML). HTML е език чрез който се указва на браузъра какво да визуализира в изгледа на браузъра. Всеки HTML компонент се описва с Document Object Model (DOM) обект. Необходимият HTML код е записан в един или множество файлове с разширение html. Тези файлове съдържат код, който е съставен от етикети (tags). На настоящият етап се работи с версия 5 на HTML – HTML5.
2. Cascading Style Sheets (CSS). Основната цел на CSS кода е задаване на стилово форматиране на DOM обектите. Стилово форматиране може да се реализира и чрез HTML код, но това не трябва да се използва. За да се реализира по-лесно стиловото форматиране и за да може лесно да се модифицира, то трябва да се реализира само чрез CSS код, който е съсредоточен в един или множество файлове с разширение css. Свързването на CSS код с HTML код се реализира чрез един или няколко HTML етикети с име link. На настоящият етап се работи с версия 3 на CSS – CSS3.
3. JavaScript. Това е скрипт език, който по своята същност е хибриден – поддържа се обектно-ориентирано и функционално програмиране. Неговата основна цел е изпълнение на програмен код (логика) от ниво браузър. За да е възможно това е необходимо всеки браузър да има вграден интерпретатор на JavaScript код. Чрез JavaScript код имате достъп до свойствата и методите на DOM обектите и по този начин можете да ги променяте. Можете да генерирате различни формати заявки към Web- или облачно-базирани услуги (XML и JSON заявки), както и да обработвате отговорите, получени от тези услуги. Необходимият код е добре да бъде съсредоточен в един или множество файлове с разширение js. Той се зарежда в паметта и свързва с необходимия HTML код чрез етикет с име script. На настоящият етап се работи с основно с версии 7 и 8 на JavaScript.

Нека да обобщим: чрез HTML кода се задава какво трябва да се визуализира в изгледа на браузъра; чрез CSS се реализира стилово форматиране на това, което се вижда, а чрез JavaScript се интегрира логика, която се изпълнява от страна на клиента.

II. Hyper Text Markup Language (HTML)

Всеки HTML документ се състои от етикети. Етикетите имат имена и трябва да са в ъглови скоби < >. Освен име, всеки етикет може да има един или няколко атрибута.

Атрибутите следват след името на етикета и имат име и стойност. Връзката на името и стойността на етикета е символ =. Стойността на атрибутите са в кавички “ ”. Съществуват два вида етикети в зависимост от това дали в тях може да има други етикети или не. Етикетите, които могат да съдържат други етикети, са етикети с тяло. Началото на тялото е етикет, който започва с името му, а краят – етикет, който започва със символ / след който следва името на етикета. Етикетите без тяло завършват със символ /. На Фиг. 3-1 е показана примерна структура на HTML5 документ.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Структура на HTML документ</title>
         <!-Задаване на необходимата кодова таблица за кирилица -->
        <meta charset="utf-8">
        <!—Зареждане на необходимия JavaScript код -->
        <script type="text/javascript" src="myCode.js"></script>
        <!—Зареждане на необходимия CSS код -->
         <link rel="stylesheet" type="text/css" href="myStyle.css"/>
    </head>

    <body>
        Тяло на етикет body – съдържа HTML етикети, които задават какво се
         визуализира в изгледа на браузъра (потребителски интерфейс).
    </body>
</html>
```

Фиг. 3-1 Структура на HTML документ

Всеки HTML5 документ започва с етикет <!DOCTYPE>. Чрез него се задава, че документът съдържа HTML код. В тялото на етикет <html> се съдържат всички останали етикети. Структурата на документа се състои от две основни секции – *заглавен блок* и *тяло*. Заглавният блок се задава чрез началото и края на етикет <head>. В него може да има множество други етикети, по-важните от които са следните:

- <title> - име на страницата. Визуализира се в title bar на браузъра.
- <meta> - служи за предаване на мета информация към браузъра, например: Каква кодова таблица да се използва при визуализиране на текста от HTML страниците? Това се задава чрез атрибут charset. В конкретният пример е зададена кодова таблица UTF-8. Това ще ни позволи визуализиране на текст не само на латиница, но и на кирилица. Чрез meta етикети може да предавате информация не само към браузъра, но и към търсещите машини, които имат за задача да индексират вашите страници.
- <script> - служи за зарежда не на JavaScript код от файл, който се задава чрез атрибут src.
- <link> - служи за зареждане на CSS код от файл. Името на файла е стойността на атрибут href.

Етикет <body> формира тялото на HTML документа. В него се съдържат HTML етикети чрез които се реализира потребителския интерфейс. Компонентите, които могат да изградят един потребителски интерфейс, са разнообразни, например: текст, картинки, вградено видео, бутони, радио-бутони, check-боксове, плъзгачи и много други.

Структурирането и самото съдържание в тялото на документа е много важно за индексирането на вашия сайт от *търсещите машини*. Текстовата информация от тялото на документа трябва да съответства семантично на описанието на сайта чрез мета данните от заглавния блок на документа (description, keywords). Търсещите машини

„обичат” информацията в тялото да е добре структурирана. За целта е добре да използвате етикети header, footer, main и section (виж Фиг. 3-2)

```
<html>
    <head>
        <title>Заглавие на страницата</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <meta name="description" content="Описание на страницата ..."/>
        <meta name="keywords" content="Ключови думи, описващи страницата"/>
        <meta name="robots" content="index, follow"/>
        <link rel="stylesheet" type="text/css" href="css/style.css"/>
        <script src="libs/jquery-3.1.1.min.js"></script>
        <script src="js/code.js"></script>
    </head>
    <body>
        <header>Информация за началото на блока с информация</header>
        <main>
            <section id="sec-1">
                Информация от Секция 1
            </section>
            <section id="sec-2">
                Информация от Секция 2
            </section>
            <section id="sec-3">
                Информация от Секция 3
            </section>
        </main>
        <footer>Информация за края на блока с информация</footer>
    </body>
</html>
```

Фиг. 3-2 Примерна структура на тялото на HTML документ

Този начин на структуриране на информацията ще ви позволи и създаване на Single Page Apps (SPA), без да е необходимо за това да използвате специална библиотека. При този тип Web приложения се работи само с една HTML страница, а съдържанието от тялото й се променя динамично чрез JavaScript код. При конкретната структура на документа в даден момент от време може да показвате само една от всички секции, а останалите да са скрити.

III. Cascading Style Sheets (CSS)

При Web програмирането CSS се използва с цел стилово форматиране на страниците. CSS е стандарт (W3C CSS Working Group) и това гарантира, че услугата би трябвало да изглежда стилово еднакво, независимо от използвания браузър. CSS подобрява скоростта на интерпретиране на HTML документите, тъй като отпада необходимостта стиловото форматиране да се задава чрез атрибути на таговете. Чрез CSS сравнително лесно се постига responsive дизайн на услугите – потребителският интерфейс да изглежда еднакво добре на устройства с различна разделителна способност на екрана. Версия 1 на CSS е разработена през декември 1996 с цел замяна на стиловото форматиране, което предоставя HTML чрез атрибути към таговете. През май 1998 се анонсира CSS2. Тази версия добавя нова функционалност към CSS1, например: позициониране на елементите; WEB шрифтове; стилово форматиране на информация от

таблици; специфични за дадена медия стилово форматиране и др. На настоящият етап се използва основно CSS3, чиято спецификация се публикува през 1999 г. Последната версия CSS4 (W3.CSS - Pro) която се анонсира през март 2017 г.

Стиловото форматиране на DOM обектите може да се реализира чрез: 1) Специфични атрибути към различните HTML етикети; 2) Чрез етикет <style>, който се интегрира в HTML документа и 3) Чрез CSS код. Единственият препоръчителен начин за стилово форматиране е чрез използване на CSS код. Това позволява съсредоточаване на форматирането в един или няколко CSS файла. Всеки файл съдържа поредица от CSS правила. Всяко правило има синтаксис, показан на Фиг. 3-3.

```
селектор {
    дефиниция 1;
    дефиниция 2;
    свойство: стойност;
    свойство: стойност;
}
p {
    font-size: 14px;        // размер на шрифта 14 пиксела
    font-weight: bold;      // удебелени символи
    margin: 0px;            // Без отстъп спрямо останалите съседни обекти
    padding-top: 10px;      // Горен отстъп на текста от рамката на елемента
    text-align: center;     // Центриране на текста
    color: #00FF00;         // Задаване на цвят на текста (100% зелено)
}
```

Фиг. 3-3 Синтаксис на CSS

3.1 Синтаксис

Стиловото форматиране се задава чрез CSS *правила*. Всяко правило (rule) се състои от три елемента:

- Селектор (selector) – задава за кого се отнася CSS правилото: всички етикети от даден вид, група от няколко вида етикети, група от еднотипни етикети или конкретен етикет.
- Свойство (property) – задава какво точно се форматира стилово: цветове, позициониране, рамки на обектите и много др.
- Стойност на свойство (value) – задава стойността на дадено свойство, например кода на избрания цвят.

Синтаксисът на CSS правилата е следния:

```
селектор {
    свойство: стойност;
}
```

Всяко свойство и неговата стойност завършват със символа ; и се нарича декларация. Едно CSS правило може да има една или повече декларации.

3.1.1 Видове CSS селектори

Вида на селекторите зависи от версията на CSS спецификацията. Следва описание на най-често използваните видове селектори:

- Универсален селектор. Този селектор се използва за създаване на правило, което се отнася за всички HTML етикети. За целта се използва символ *.

```
* {
    color: red;
}
```

- Селектор отнасящ се за определен HTML таг. В този случай името на селектора съвпада с името на конкретен таг (етикет). Например, ако трябва да зададем червен цвят на символите за всички параграфи може да използваме следното правило:

```
p {
    color: red;
}
```

- Селектори за даден клас. В този случай се задава еднакво стилово форматиране за група HTML етикети. Етикетите могат да бъдат от един и същи тип (например всички параграфи), но могат да бъдат и различни (всички параграфи и div контейнери). Групата се формира чрез задаване една и съща стойност на атрибут клас. Преди името на класа трябва да има символ точка (.). Например, следващото правило задава червен цвят на символите за всички HTML етикети със стойност red за атрибут class.

```
.red {
    color: red;
}
```

Ако трябва да групираме стилово няколко етикета от един и същи тип, например параграф, то името на етикета (p) трябва да предхожда символа точка:

```
p.red {
    color: red;
}
```

Самите параграфи трябва да имат еднаква стойност за атрибут клас:

```
<p class="red"> текст 1 </p>
<p class="red"> текст 2 </p>
```

- ID селектори. В този случай се задава уникално правило за конкретен етикет или група етикети. Връзката между CSS селектора на правилото и HTML етикетите се задава чрез атрибут id. Правилото важи за етикета или етикетите с конкретно id. За да се зададе, че селектора е id тип се използва символ #:

```
#red {
    color: red;
}
```

Ако трябва да групираме стилово няколко етикета с еднакви id, например параграф, то името на етикета (p) трябва да предхожда символа #:

```
p#red {
    color: red;
}
```

- Селектори за вложени етикети (descendant). В този случай CSS правилото се отнася за етикет, който е в тялото на друг етикет. Разделителя между двата етикета е символа интервал. Следното правило се отнася за всички параграфи, който са в тялото на div етикет:

```
div p {
    color: red; }
```

- Селектори за дъщерни етикети (child). Дъщерни са тези етикети, които са *непосредствено* в тялото на родителския етикет. Връзката между двата етикета е символ >. Следното правило се отнася за всички параграфи, който са дъщерни на div:

```
div > p {
    color: red; }
```

- Групиране на селектори. Ако е необходимо може да се зададе едно и също CSS правило за няколко CSS селектора. Целта е да не се повтарят CSS правила с еднакви декларации. Групирането се реализира чрез използване на символ запетая (,) между имената на селекторите:

```
p, .red  {
    color: red;
}
```

- Селектори за етикети, които следват други етикети. Ако трябва да зададем CSS правило за етикет, който непосредствено е след друг етикет се използва символ плюс (+) между двата етикета. Следното правило задава червен цвят на символите за параграфа, който непосредствено следва div контейнер:

```
div + p  {
    color: red;
}
```

- Селектори за конкретен атрибут на етикет. Този тип селектори позволява задаване на CSS правило за конкретен атрибут или конкретна стойност на атрибут. Следното правило се отнася за етикет input, който има стойност *password* за атрибут *type*:

```
input[type=″password″] {
    color: red;
}
```

  Има множество варианти за този тип селектор, например:
  - p[lang] – правилото се отнася за всички параграфи за който се използва атрибут lang. Този атрибут се използва за задаване на езика на който е текста от параграфа. Използва се основно с цел да се помогне на търсещите машини.
  - p[lang="bg"] – правилото се отнася за всички параграфи за които е зададен български език (стойност bg за атрибут lang).
  - p[lang~="bg"] – правилото се отнася за всички параграфи за които има bg в стойността на атрибут lang.
  - p[lang|="bg"] – правилото се отнася за всички параграфи за които стойността на атрибут lang е bg или започва с bg.

3.2 Мерни единици

При задаване на размерите на обектите, както и отстъпите спрямо другите обекти, може се използват различни мерни единици (CSS units). Всяка мерна единица може да се причисли към една от следните две групи:

- абсолютни единици;
- относителни единици.

Абсолютните единици не зависят от физическите размери на екрана на устройството на което е стартиран браузъра. Това ги прави предпочитани, само ако тези размери са фиксирани или известни предварително, както и подготовка за печат на документ на принтер. Относителните единици са пропорционални на размера на друга мерна единица или размерите на екрана. Това ги прави подходящи при реализация на страници с responsive дизайн. Следва описание на по-важните абсолютни мерни единици.

Мерна единица	Описание
cm	сантиметри
mm	милиметри
in	Инчове (1 инч = 2.54cm = 96px)
px	Пиксели (1 пиксел = 1/96 инча)
pt	Точки (1 точка = 1/72 инча)

pc	Пики (1 пика = 12pt)

Следва описание на по-важните относителни мерни единици.

Мерна единица	Описание
%	Размерът е относителен (в проценти) спрямо родителския елемент.
em	Размерът е относителен спрямо размера на символите от текущо активния шрифт. Например, 2em задава размер 2 пъти по-голям от височината на символите от текущо активния шрифт.
ch	Размерът е относителен спрямо ширината на цифрата нула.
vw	Размерът е относителен (в проценти) спрямо ширината на прозореца на браузъра (viewport).
vh	Размерът е относителен (в проценти) спрямо височината на прозореца на браузъра (viewport).
vmin	Размерът е относителен (в проценти) спрямо минимума между височината и ширината на прозореца на браузъра (viewport).
vmax	Размерът е относителен (в проценти) спрямо максимума между височината и ширината на прозореца на браузъра (viewport).

3.3 Цветове

Кодирането на цветовете при CSS е на базата на името на цвета или кода на цвета. При CSS3 се използва две пространства от цветове (цветови схеми): RGB и HSL. При цветова схема **RGB** цветът се задава чрез стойностите (1 байт) са три основни цвята: червено (R), зелено (G) и синьо (B). Стойностите могат да бъдат в десетична бройна система (0-255) или шестнадесетична бройна система (00-FF). За да използвате десетични числа трябва да работите с функция rgb(). Когато задавате цвета в шестнадесетична бройна система, кодът му се предхожда от знак #.

```
p {
   color: #FF0000;
   background-color: rgb(255,0,0);
}
```

Ако желаете да зададете прозрачност на цвета (alpha) може да използвате функция rgba(). В този случай, стойността на параметър alpha е между 0 (напълно прозрачен) и 1 (напълно непрозрачен). Например `rgba(255,255,0,0.8)` задава жълт цвят, който е 80% прозрачен. Стойността на параметър alpha може да зададете и в шестнадесетична бройна система. В този случай той се задава като четвърти параметър, например `#FFFF00CC`.

При цветова схема **HSL** цветът се задава чрез три параметъра: нюанс (hue), наситеност (saturation) и субективна яркост на цвета (lightness). За да изберете цвят, използвайте колело на цветовете (color wheel). Нюансът се измерва в градуси (от 0 до 360). Например, 0° или 360° е 100% червено; 120° е 100% зелено, а 240° е 100% синьо. Наситеността се задава е проценти - 0% означава сянка на сивото, а 100% е чистия цвят. Субективната яркост също се задава в проценти - 0% е черно, а 100% - бяло. За задаване на цвета може да се използват функции hsl() и hsla(). Например, следното CSS правило задава 100% жълт цвят на фона и тъмно пастелно зелено за цвета на символите от всички параграфи.

```
p {
   color: hsl (120,60%,40%);
   background-color: hsl(60,100%,50%);
}
```

Цветът може да бъде зададен и чрез неговото **име**. В CSS има предварително дефинирани 140 имена на цветове, например: red, gray, brown, coral и др.

3.4 Шрифтове

При CSS се поддържат два типа шрифтове. Едните са предварително интегрирани на хоста на клиента (браузъра), а другите се свалят от специализиран за целта сървър. В повечето случаи се работи с предварително инсталираните шрифтове. В този случай не е 100% сигурно, че избраният шрифт е наличен на всеки хост, който ще използва услугата ви. Ако искате това да е гарантирано и да използвате специфичен за вашата услуга шрифт трябва да използвате Web шрифтове. На практика всеки шрифт се описва в специфичен формат. Най-често използваните формати за шрифтове са следните;

- TrueType Fonts (TTF). Разработени са през 1980 г. от Apple и Microsoft.
- OpenType Fonts (OTF). Разработени са от Microsoft на базата на TTF с цел предоставяне на по-добра мащабируемост.
- Web Open Font Format (WOFF). Разработени са през 2009 г. и са стандартизирани като W3C препоръка. На практика това са компресирани TTF или OTF с цел оптимизиране на мрежовия трафик.
- Embedded OpenType Fonts (EOT). Разработени са от Microsoft. Имат по-компактно описани при сравнение с OTF.
- Scalable Vector Graphics (SVG). Векторен графичен формат чрез който може да се интегрират шрифтове в SVG документи.

При CSS се използват две основни семейства шрифтове:

- Generic family – група от шрифтове, които изглеждат подобно.
- Font family – специфичен шрифт.

Следва таблица в която са описани три основни семейства шрифтове (serif, sans-serif и monospace) и техни конкретни представители.

Generic family	Font family
Serif	Times New Roman Georgia
Sans-serif	Arial Verdana
Monospace	Courier New Lucida Console

Разликата между sans-serif и serif е в начина на завършване на щриха на всеки символ:

Трябва да се има предвид, че на компютърния екран по=лесно се четат sans-serif шрифтовете. Ако се нуждаете от шрифт при който всеки символ има еднаква ширина, то трябва да използвате monospace шрифт. За да зададете шрифт може да използвате свойство font или font-family:

```
p {
    font-family: 'Open Sans', sans-serif;
    font: 1.25vmax 'Open Sans', sans-serif;
}
```

Други свойства, свързани с работа с шрифтове, са следните:

- font-style
- font-size
- font-weight

Стилът на шрифта (font-style) може да приема три стойности: normal, italic и oblique (наклонен, но не е сигурно дали се поддържа). За *размер* на шрифта (font-size) може да използвате всички видове мерни единици, но е препоръчително да работите с относителни единици. *Теглото* на шрифта (font-weight) приема стойности като normal и bold (удебелен) или число. Колкото по-голямо е това число, толкова по-голям е размера на шрифта.

3.5 Web шрифтове

За да сте сигурни, че избраният от вас шрифт (шрифтове) ще са налични за всички клиенти на услугата е добре да използвате шрифтове, които се свалят от хранилище за шрифтове. Най-често използваното хранилище на шрифтове е това на Google (Google fonts). За целта е възможно шрифтове да се интегрират към HTML документа чрез етикет <script>. Изборът на шрифт се реализира от сайта https://fonts.google.com/?subset=cyrillic.

Зареждането на шрифта може да се реализира чрез HTML кода или чрез CSS кода. При първият вариант използвайте етикет link от тялото на етикет head:

```
<link rel="stylesheet"  href="https://fonts.googleapis.com/css?family=Jura">
```

При вторият вариант се използва CSS правило @import. То трябва да бъде в *началото* на CSS файла, например:

```
@import url('https://fonts.googleapis.com/css2?family=Bad+Script&display=swap');
```

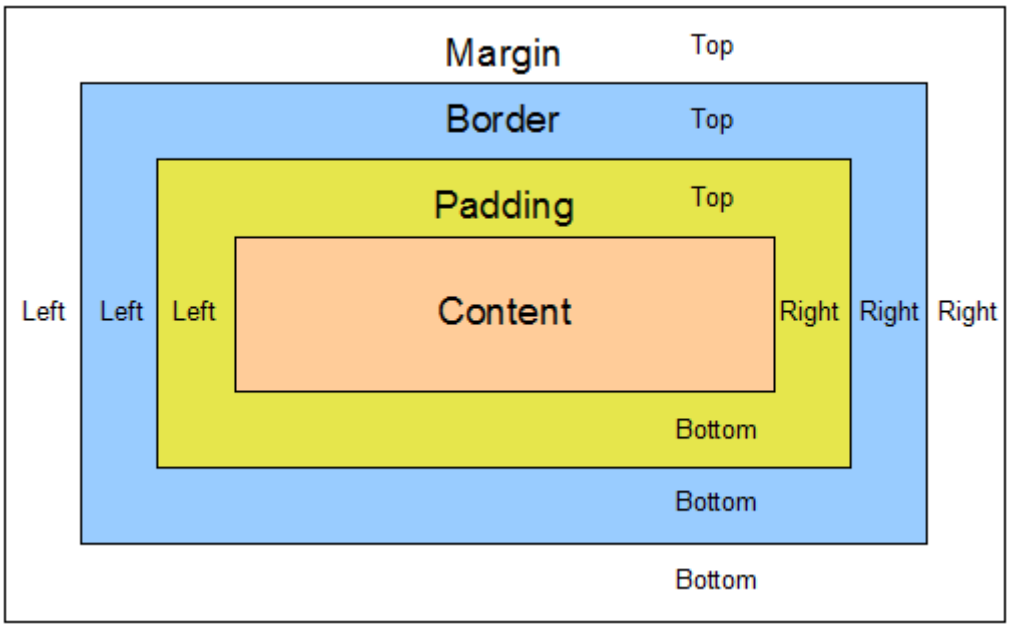
Друг начин за интегриране на собствен шрифт е да използвате правило @font-face:

```
@font-face {
    font-family: myFont;
    src: url(path/to/myFontFile.woff);
}
```

3.6 Отстъпи

Всеки обект, който се визуализира в изгледа на браузъра, има рамка. Тя е невидима по подразбиране. Чрез CSS правила може да зададем типа и свойствата на рамката, както и външните (**margin**) и вътрешни (**padding**) отстъпи (виж Фиг. 3-4). Външните отстъпи задават отстоянието на всеки обект спрямо съседните му обекти. Могат да се задават външни отстъпи за всяка страна на обекта: ляво (left), дясно (right), горе (top) и долу (bot-

tom). По подобен начин могат да се задават стойности и за вътрешните отстъпи (от рамката до съдържанието на обекта).



Фиг. 3-4 Отстъпи в CSS

Свойствата, които се използват за контрол на външните отстъпи са следните:

- margin
- margin-top
- margin-bottom
- margin-left
- margin-right

Свойствата, които се използват за контрол на вътрешните отстъпи са следните:

- padding
- padding-top
- padding-bottom
- padding-left
- padding-right

За стойност на всяко от тези свойства могат да се използват различни мерни единици, за свойства margin и padding – стойност auto. В този случай, отстъпите се изчисляват от браузъра. При следващия пример за вътрешните отстъпи са зададени стойности в пиксели:

```
p {
    padding-top: 20px;
    padding-right: 10px;
    padding-bottom: 20px;
    padding-left: 30px;
}
```

Това правило може да се запише и много по-компактно:

```
p {
    padding: 20px 10px 25px 30px;
    /* top  right bottom left  */
}
```

Има и синтаксис с три параметъра:

```
p {
    padding: 20px 10px 30px;
    /*  top  left,right bottom   */
}
```

Ако стойностите за отстъпите отляво и отдясно, както и отгоре и отдолу са еднакви се използва синтаксис с две стойности:

```
p {
    padding:  20px  25px;
    /*  top,bottom left,right   */
}
```

Ако всички отстъпи са еднакви се използва синтаксис с една стойност:

```
p {
    padding: 20px;
    /* top,bottom,left,right   */
}
```

3.7 Изображения

Стиловото форматиране на изображенията се свежда до тяхното позициониране, трансформиране, филтриране и др. Мащабирането на изображенията и позиционирането им в зависимост от размера на екрана е в основата на гарантиране на responsive дизайна на услугите.

За да позиционирате центрирано едно изображение е необходимо да зададете блоков тип визуализиране и автоматично изчисляване на външните отстъпи:

```
img {
    display: block;
    margin-left: auto;
    margin-right: auto;
}
```

Когато е необходимо изображението да запълва екрана в ширина, но без да се увеличава реалната ширина на изображението се използва свойство max-width със стойност 100%:

```
img {
    max-width: 100%;
    height: auto;
}
```

IV. JavaScript

За рождена дата на JavaScript се приема 1995 година, когато Netscape Corp. анонсира Mocha (по-късно LiveScript) като скрипт език за браузъра Netscape Navigator 2.0. По-късно (декември 1995) LiveScript се преименува на JavaScript. В основата на тази разработка е Брендън Айк, който е основател на проекта Mozilla, фондация Mozilla (създадена да поддържа проекта Mozilla) и Mozilla Corp., която разработва и разпространява браузъра Mozilla Firefox.

Програмният език JavaScript е интерпретаторен език от високо ниво. Както повечето скрипт езици и JavaScript поддържа ***динамично типизиране*** – типа на данните се определя в момента на инициализацията им (задава им се стойност). Той се базира на ECMAScript (ES), който през 1997 е стандартизиран (ECMA-262) като език за създаване на Web приложения от страна на клиента. В момента се работи основно с ES6+. *JavaScript е хибриден език, прототипно-базиран, обектно ориентиран и поддържа функционално програмиране*. Има вградени приложни програмни интерфейси (API) за работа с текст, масиви, списъци, регулярни изрази и др.

Програмният език JavaScript е създаден за разработване на Web приложения. Чрез него се пише основно *front-end* код – програмен код, който се интерпретира от Web браузър. Поради тази причина всеки браузър има вграден JavaScript интерпретатор. Това е основният начин за изпълнение на някаква логика под управлението на браузър. На съвременният етап JavaScript намира много по-широко приложение, например:

1. Създаване на *back-end* код – програмен код, който се изпълнява от страна на Web сървър. Чрез този код се реализира логиката на проекта, включително и достъпа до бази от данни. Типичен пример за такъв код е $Node.js$ -мулти-платформена среда (Microsoft Windows, Linux, OS X) за създаване на сървърни и мрежови приложения на JavaScript. В основата на $Node.js$ е V8 engine на Google. V8 компилира JavaScript кода директно до машинен код, който след оптимизация по бързодействие се изпълнява от машината на която е стартиран браузъра. Част от най-използваните сървъри за управление на $NoSQL$ бази данни като $MongoDB$ и $CouchDB$ използват https интерфейс за генериране на заявки към базата данни. Основна причина за използване на JavaScript като back-end код е неговата платформена независимост и възможности за писане на бърз асинхронен код.
2. Крос-платформи за създаване на *мобилни приложения* за различни мобилни операционни системи (Android, iOS, Windows 8, WebOS и др.) на базата на HTML5 и JavaScript. Съществува голямо разнообразие от подобни платформи, най-широко използваните от които са следните: $Sencha$, $PhoneGap$ ($Apache\ Cordova$), $Unity\ 3D$, $Titanium$, $Intel\ XDK$ и др.
3. Мобилни операционни системи, базирани на HTML5 и JavaScript. Ако трябва да се създават мобилни приложения на JavaScript за мобилни устройства е по-добре тази функционалност да е част от самата мобилна операционна система, отколкото да се използва някоя от крос-платформените програмни рамки (frameworks). Подобна операционна система е $Tizen$ на Samsung.
4. Създаване на *desktop* платформено-независими приложения. За целта, най-често се използват библиотеки $Electron.js$ и $NW.js$. Поддържаните операционни системи са Windows, OS X и Linux.
5. Създаване на програмния код за *вградени* ($embedded$) системи. Все повече микроконтролери и „компютри на чип" позволяват писането на програмния код да се реализира директно на JavaScript. Това се реализира чрез специализирани библиотеки или програмни рамки. Апаратните модули, които позволяват използване на JavaScript, са $Arduino$, $Raspberry\ Pi$, $Tessel$, $Samsung\ ARTIK$ и др.

Когато дадено приложение се разработва изцяло на JavaScript се казва, че се използва концепция „$Full-stack\ JavaScript$". Стекът от технологии включва:

- Потребителски интерфейс и логика от страна на клиента: HTML5, CSS3, JavaScript, JavaScript front-end библиотеки ($Angular$, $React$, Vue и др.).
- Бизнес логика на приложението: изцяло JavaScript програмни обекти.
- Слой данни: $MongoDB$, $MariaDB$, $Redis$ и др.
- Сървър: засега най-често $Node.js$.
- Програмни рамки за реализиране на $RESTful$ интерфейс със сървъра: $Express.js$, $Socket.io$ и $Total.js$.

Програмният код на JavaScript се интегрира директно в HTML страниците чрез етикет $script$ или чрез текстов файл с разширение js. Следователно, за въвеждането на изходния JavaScript код може да използвате кой да е *текстов* редактор. На практика се работи с on-line или off-line развойни среди (IDE). Основният недостатък на on-line развойните среди е невъзможността да ги използвате, ако нямате достъп до Интернет или мрежовата връзка в даден момент е много бавна. Алтернативата е да използвате off-line развойна среда, която трябва да инсталирате на вашия компютър. Най-често използваните развойни среди, които можете да използвате за тестване на HTML, CSS и JavaScript код, са: $NetBeans$, $Eclipse$ и $Microsoft\ Visual\ Studio$.

4.1. Ключови думи в JavaScript

Ключовите думи в един програмен език се използват за служебни цели и не могат да се използват с цел именуване на променливи, константи, обекти и класове. Трябва да се прави разлика между *ключови* и *резервирани* думи. Последните са ключови думи, които ще се използват при някоя от бъдещите версии на езика. Ключовите и резервни думи при ES (JavaScript) са описани в Табл. 3-1.

Табл. 3-1 Ключови и резервни думи в ES (JavaScript)

abstract	arguments	await*	boolean
break	byte	case	catch
char	class*	const	continue
debugger	default	delete	do
double	else	enum	eval
export*	extends*	false	final
finally	float	for	function
goto	if	implements	import*
in	instanceof	int	interface
let*	long	native	new
null	package	private	protected
public	return	short	static
super*	switch	synchronized	this
throw	throws	transient	true
try	typeof	var	void
volatile	while	with	yield

* Тези ключови думи могат да се използват само при ES6+.

4.2. Типове данни

В JavaScript всички данни се декларират чрез ключови думи $\boldsymbol{var}$ или $\boldsymbol{let}$ (ES6+), без да се задава техния тип явно. Вътрешно, след инициализация на данните, те се причисляват към един от следните типове с които JavaScript работи:

- *Числа*;
- *низове* (*текст*);
- *булеви данни*;
- *функции*;
- *обекти*.

Програмно можем да определим типа на данните чрез използване на ключова дума (оператор) $typeof$. Синтаксисът е „$typeof$ *данна*”. Този израз връща вътрешния тип на *данна*.

4.2.1. Низове

Типът на низовете е $string$. Те са текст на някакъв естествен език. При JavaScript низовете се дефинират чрез заграждане на текста в *кавички*, *апостроф* или *наклонен апостроф*:

```
var низ1 = "Технически Университет - Габрово";
var низ2 = 'ул. "Х. Димитър" 4';
var низ3 = `ул. \"Х. Димитър\" 4`;
```

Когато в даден низ се налага да има кавички, например име на улица, той трябва да започва с апостроф (*низ*2) или за вътрешните кавички да се укаже, че това е кода на символа кавичка (\"), а не начало и край на низ (*низ*3). При JavaScript е възможно обединяване на низове в един низ чрез оператор $+$. Когато към един низ се добави число, JavaScript не връща грешка, а приема, че числото трябва да се преобразува до низ и след това да се добави до другия низ. В JavaScript има множество вградени методи за работа с низове. По-често използваните от тях са описани в Табл. 3-2.

Табл. 3-2 Основни методи, които се използват при работа с низове

Метод	Предназначение	Синтаксис
toUpperCase	Преобразува символите на низ от малки до главни.	низ.toUpperCase()
toLowerCase	Преобразува символите на низ от главни до малки.	низ.toLowerCase()
charAt	Връща символа от низ, който е на указана позиция (от началото на низ) в него.	низ.charAt(позиция)
charCodeAt	Връща кода (Unicode) на символа в низ, който е на указана позиция (от началото на низ) в него.	низ.charCodeAt(позиция)
indexOf	Търси *първият* съвпадащ низ в низ от *началото* към *края*. Връща индекса от който започва намерения низ и -1, ако низа не е намерен. Може да зададете позицията от която да започне търсенето като втори параметър.	низ.indexOf(„търсен низ”, позиция)
lastIndexOf	Търси *последният* съвпадащ низ в низ от *началото* към *края*. Има аналогичен синтаксис на indexOf.	низ.lastIndexOf(„търсен низ”, позиция)
search	Търси *първият* съвпадащ низ в низ от *началото*	низ.search(„търсен низ”)

	към *края*. Има аналогичен синтаксис на indexOf. Не може да се задава позицията от която започва търсенето, но могат да се използват *регулярни изрази*.	

slice	Връща част от низа при зададени начален и краен индекс. Ако се използва само първият параметър – връща низ от указаната позиция до края на низа. Ако индексът е отрицателно число - адресирането е от края на низа.	низ.slice(начален_индекс, краен_индекс)
substring	Връща част от низа при зададени начален и краен индекс. Не поддържа отрицателни индекси.	низ.substring(начален_индекс, краен_индекс)
substr	Връща част от низа при зададени начален индекс и брой символи (дължина на низа).	низ.substr(начален_индекс, дължина)
replace	Заменя низ (*низ*1) от даден низ с друг низ (*низ*2). Изходния низ не се променя. Поддържа използването на регулярни изрази (първи параметър). По подразбиране се заменя само първият срещнат низ.	низ.replace(„низ1”, „низ2”)
concat	Слепване на низ към низ. Изпълнява функциите на оператор +. Може да слепва масиви от низове. Не променя изходния низ.	низ.concat(нов_низ);
split	Преобразува низ до масив от низове. Изисква задаване на разделител, чрез който определя кои да са елементите на масива. Разделителят може да е кой да е символ или низ.	низ.split(разделител);

4.2.2. Масиви

Масивите са данни от тип $object$. В JavaScript масивът е *обект*, който може да съдържа множество (един или повече) елементи. Елементите могат да бъдат от един и същи тип, (числа, низове, обекти) или от различни типове. Масивите се декларират подобно на променливите, но след оператор = следват стойностите на масива, заградени в квадратни скоби [ ], например:

```
var масив1 = [];                          // празен масив
var масив2 = [1, 2, 3, 4, 5, 6];          // масив от числа
var масив3 = ["а", "б", "в"];             // масив от низове
var масив4 = [1, "а", 2, "б", "в", 3];    // масив от числа и низове
var масив5 = [масив2, масив3, масив4];    // масив от обекти (масиви)
```

За да адресирате даден елемент от масив трябва да използвате следния синтаксис:

```
имеНаМасива[индекс_на_елемента]
```

Първият елемент от даден масив има индекс 0, а последният – броят на елементите – 1. Например *масив*2[3] ще върне стойност 4. Можете да получите броя на елементите в един масив, без да ги броите, чрез свойство $length$, например:

```
var n = масив2.length;
```

Списък на по-често използваните методи при работа с масиви и тяхното предназначение е описано в Табл. 3-3.

Табл. 3-3 Основни методи, които се използват при работа с масиви

Метод	Предназначение	Синтаксис
push	*Добавя* нов елемент в *края* на масива	масив.push(елемент)
pop	*Изтрива* елемента в *края* на масива	масив.pop()
unshift	*Добавя* нов елемент в *началото* на масива	масив.unshift(елемент)
shift	*Изтрива* елемента в *началото* на масива	масив.shift()
concat	Слепва два масива (добавя масив в края на масив)	масив.concat(нов_масив)
toString	*Преобразува* масив в *низ*. Елементите на масива са разделени със запетая.	масив.toString()
join	*Преобразува* масив в *низ*. Чрез параметър към метода може да се зададе какъв да бъде *разделителя* между елементите на масива. Без параметри методът функционира като $toString$	масив.join() масив.join(" ") масив.join("\n")
splice	*Вмъкване*, *заместване* или *изтриване* на елемент(и). Могат да се предават до 3 параметъра: 1) позиция, за която се отнася действието; 2) колко елемента, след тази позиция да бъдат премахнати; 3) ако има вмъкване – стойността на новия елемент.	*Вмъкване*: масив.splice(3,0, елемент) *Заместване*: масив.splice(3,1, елемент) *Изтриване*: масив.splice(3,1)
slice	Извлича масив от масив. Елементите на новия масив зависят от параметрите, които се предават: 1) позиция на началният елемент; 2) позиция на крайния елемент+1.	var нов_масив = масив.slice(3,5)
indexOf	Проверка за наличие на елемент в масив. Ако елементът съществува, методът връща неговата позиция. В противен случай връща -1.	масив.indexOf(елемент)
reduce	Вика функция, която се предава на метода толкова пъти, колкото са елементите в масива	масив.reduce(функция)

4.2.3. Функции

Функциите са данни от тип $\boldsymbol{function}$. Въпреки, че типът на функциите е $function$, в JavaScript те са *обекти*, по-точно обекти от тип $function$. За всеки обект-функция се поддържат вътрешни свойства и методи. Например, свойство $arguments$ е масив, който съдържа предаваните на функцията параметри. След ключова дума $function$ следват името на функцията (*име*) и скоби (). Интерпретаторът разпознава функциите именно по тези скоби. Имената на функциите могат да съдържат букви, цифри и символи като _ и $. Програмният код от тялото на функцията трябва да бъде във фигурни скоби { }.

На кода от тялото на функцията могат да се предават или да не се предават параметри (аргументи). Ако се предават параметри, техните имена се описват в скобите, разделени

със запетая един от друг. Няма ограничение за броя на предаваните параметри. Стойностите на параметрите са необходими за формиране на логиката на функцията. При определени случаи, на функцията трябва да е възможно предаването на различен брой параметри при различните й извиквания. При JavaScript това е възможно чрез използване на оператор … (***spread operator***).

В JavaScript обектите (включително масивите) се предават по *референция*. Това означава, че ако промените стойностите на елементите на масив или свойство на обект от тялото на функцията, *това ще се отрази на оригиналните обекти*. На Фиг. 3-5а е показан програмния код на функция, която променя елемент от обект-масив. Ако трябва оригиналният обект да не се променя, можете да използвате копие на обекта от тялото на функцията (обект-клонинг). В JavaScript няма специални методи за *клониране*, но за масивите можете да използвате метод *slice* без да му предавате параметри. В този случай, този метод ще върне нов обект - копие на оригиналния масив. Метод *slice* реализира така нареченото ***повърхностно клониране*** (*shallow cloning*). Ако се налага ***дълбоко клониране*** (*deep cloning*) трябва да създадете нов масив и в цикъл да копирате всеки елемент на оригиналния масив и да го запишете в съответния елемент на новия масив.

а)
```
1  var масив = [1, 4, 5];
2  function моятафункция(масив) {
3      масив[2] = 500;
4      console.log(масив);
5  }
6  моятафункция(масив);
7  console.log(масив);
```

б)
```
▶ (3) [1, 4, 500]      code.js:4
▶ (3) [1, 4, 500]      code.js:7
```

Фиг. 3-5. Пример за предаване на параметри по референция:
а) програмен код, б) резултат

На Фиг. 3-6а е показан програмния код, който решава задачата чрез повърхностно клониране на масива (ред 3).

а)
```
1  var масив = [1, 4, 5];
2  function моятафункция(масив) {
3      масив = масив.slice();
4      масив[2] = 500;
5      console.log(масив);
6  }
7  моятафункция(масив);
8  console.log(масив);
```

б)
```
▶ (3) [1, 4, 500]      code.js:5
▶ (3) [1, 4, 5]        code.js:8
```

Фиг. 3-6 Пример за използване на повърхностно клониране на масив:
а) програмен код, б) резултат

Кодът от тялото на функцията се изпълнява при всяко „викане" на функцията. При JavaScript е възможно множество начини за викане на функции, в зависимост от начина им на дефиниране:

- Викане на функцията **по име**. Това е „стандартен" начин за викане на функция.

- Викане на функция чрез **името на променлива**. При JavaScript, тъй като функциите са тип данни, е възможно за стойност на променлива да се зададе функция. Функциите, които се дефинират без име се наричат ***анонимни***.
- Автоматично извикване на функция, веднага след нейното дефиниране. Този тип функции се наричат ***самоизвикващи се***.

Изпълнението на кода от функцията спира при достигане до неговия край - оператор $return$. Последният се използва да укаже какво връща функцията като резултат. Ако след $return$ няма израз, функцията не връща стойност. В този случай функциалността ѝ се състои само в изпълнение на кода от тялото на функцията. Една JavaScript функция може да върне като резултат *число*, *стойността на променлива* или *функция*.

Функционално програмиране

Всеки програмен език използва функции. При *императивните* езици функциите са подпрограми (последователност от команди), които реализират някаква функционалност, която може да промени логиката на цялата програма. При *функционалното* програмиране функцията симулира действието на математична функция, която връща винаги един и същ резултат за еднакви по стойност аргументи (кодът ѝ не дава страничен ефект). Този тип програмиране позволява писането на капсулиран код, който не дава странични ефекти, които са характерни за императивните функции. Вече знаем, че на една функция може да се предава като параметър друга функция, както и една функция да връща като резултата друга функция. Тази функционалност не е присъща за всички програмни езици, а само за тези, които поддържат ***функции първа класа*** ($First-Class\ Functions$). Тези функции са в основата на *функционалното програмиране*. Едно от предимствата на тези функции е възможността за създаване на ***функции от по-висок ред*** ($Higher-Order\ Functions$ - HOF). Функциите от по-висок ред е прието да се наричат ***функционали***. Основното предимство на функционалите е възможността за фокусиране на цялата логика, която ни е необходима, на едно място. *Функционалното програмиране се базира на третиране на функциите като данни.*

Повечето функционални езици се базират на *ламбда* пресмятане. При ES6+ е възможно използването на ***ламбда*** (λ) функции, известни още като ***функции-стрелка***. Те са като анонимните функции – нямат име, но разликата е в това, че *ламбда* функциите се третират като данни. Синтаксисът на тези функции (виж Фиг. 3-7) изисква да се използва *дебела стрелка* ($=>$) чрез която се замества ключова дума $function$.

```
1  () => {
2      // тяло на функцията
3  }
```

```
1  (пар1[,пар2]) => {
2      // тяло на функцията
3  }
```

Фиг. 3-7 Синтаксис на *ламбда* функциите в JavaScript

Самоизвикващи се функции

При ES6+ е възможно дефиниране на функции, които се викат веднага след края на декларирането им. Те се наричат $Immediately\ Invoked\ Function\ Expressions$ (IIFE), за по-кратко - ***самоизвикващи се функции***, въпреки, че това наименование не е достатъчно точно. Най-често използваният стил на дефиниране на IIFE е даден от Дъглас Крокфорд (Douglas Crockford):

```
;(function() {          ;(() => {                ;((пар1, …,парN) => {
  …                       …                        …
```

```
})();                    })();                    })(стойност1,…,стойностN);
```

Следва пример за самоизвикваща се функция, която показва съобщение чрез функция $alert$:

```
1  (() => {
2      alert("Самоизвикваща се функция.");
3  })();
```

4.2.4. Обекти

Програмният език JavaScript е обектно ориентиран. Той работи с два вида обекти:

- Обекти-литерали.
- Обекти, създадени по шаблон от клас.

Обекти-литерали

В JavaScript се поддържа програмни ***обекти-литерали***. Те са асоциативни масиви, по-точно колекции от двойки "ключ-стойност". Всеки ключ може да се използва еднократно в колекцията и предоставя име за *свойство* на обект. Обектите са специфична комбинация от *свойства* и техните *стойности*. Свойствата могат да бъдат както *данни*, така и *функции*. Следователно, обектът е комбинация от специфични променливи (описват характеристиките на обекта) и функции (описват действията, които обектът може да реализира). Обектът, създаден чрез литерал, е прието да се нарича ***сингълтън*** ($singleton$) или накратко ***сек***. Секът е вид шаблон за дизайн, който се използва при обектно-ориентираното програмиране. Този шаблон се прилага обикновено при моделиране на обекти, които трябва да бъдат глобално достъпни за останалите обекти на приложението. Тази концепция е печеливша когато съществува само един обект или когато е ограничено представянето на определен брой обекти. Синтаксисът на дефиниране на обекти-литерали в JavaScript е следния:

```
var имеНаОбекта = {
   имеНаСвойство_1: стойност,
   имеНаСвойство_N: стойност
};
```

Нека да създадем обект, който описва даден човек чрез неговото име, години и града в който живее.

```
var личност = {
   име: "Иван",
   години: 20,
   град: "Габрово"
};
```

Обектът *личност* е съставен от 3 свойства, всички от тип данни (едно число и два низа). Въпреки, че *личност* се декларира като променлива (var) тя е нещо по-сложно, защото самата тя е изградена от променливи. Достъпът до свойствата на един обект може да се реализира по два начина:

- Точкова нотация: *имеНаОбекта.имеНаСвойство*;
- Скоба-нотация: *имеНаОбекта*['*имеНаСвойство*'];

Тъй като функциите в JavaScript са данни, то някои от свойствата на един обект-литерал могат да бъдат функции:

```
var личност = {
    име: "Иван",
    възраст: 20,
    град: "Габрово",
    върниИнформация: function() {
        var информация = this.име+" е на "+
            this.възраст+" години.";
        return информация;
    }
};
```

Фиг. 3-8 Пример за обект-литерал

Обекти, получени чрез използване на програмен клас

JavaScript поддържа както функционално програмиране, така и обектно-ориентирано програмиране. Има два подхода за деклариране на класове:

- Чрез използване на функции и
- Чрез ключова дума $class$.

Ще разгледаме само втория подход, който е по-нов и е препоръчително да се използва. За деклариране на клас се използва ключова дума $class$, която е налична при ES6+. Синтаксисът за деклариране на програмен клас е показан на Фиг. 3-9.

```
class ИмеНаКласа extends ИмеНаКласКойтоСеНаследява {
    constructor(параметри) {
        // Инициализация на
        // свойствата на класа:
        this.именаСвойство = стойност;
    }
    имеНаМетод() {
        // код от тялото на метода
    }
    // Други методи от класа
}
```

Фиг. 3-9 Синтаксис на програмен клас

Всеки клас може да има специален метод, наречен **конструктор**. Веднага след създаване на обект от класа се вика неговия конструктор. Името на конструктора е $constructor$ (ред 2). Може да декларирате ***само един конструктор***, за разлика от обектно-ориентираните езици при които могат да се декларират множество конструктори с име, което съвпада с името на класа. Програмният език JavaScript позволява ***наследяване*** на класове. То се реализира както при Java – използва се ключова дума $extends$. От тялото на конструктора можете да изпълните метод $\boldsymbol{super}$ чрез който да извикате конструктора на родителския клас. Метод $super$ трябва да е *първият* метод, който се изпълнява от тялото на конструктора.

4.3. Обработка на изключения

Обработката на изключения при JavaScript е подобно по функционалност на това при обектно-ориентирани езици като Java и C#. За целта се използват ключови думи try, $catch$ и $finally$. В тялото на клауза try трябва да е програмния код, който може да генерира грешка(и). В тялото на клауза $catch$ трябва да е програмния код, който

обработка евентуална грешка(и). Ако трябва да се изпълни програмен код, независимо дали е имало или не грешка, той трябва да е в тялото на клауза $finally$. Синтаксисът е следния:

```
try {
    // Код, който може да генерира грешки.
}
catch(error) {
    // Код, който обработва възникналите грешки. Информация за грешката
    // се получава от обект error.
}
finally {
    // код, който се изпълнява независимо дали има или не грешки.
}
```

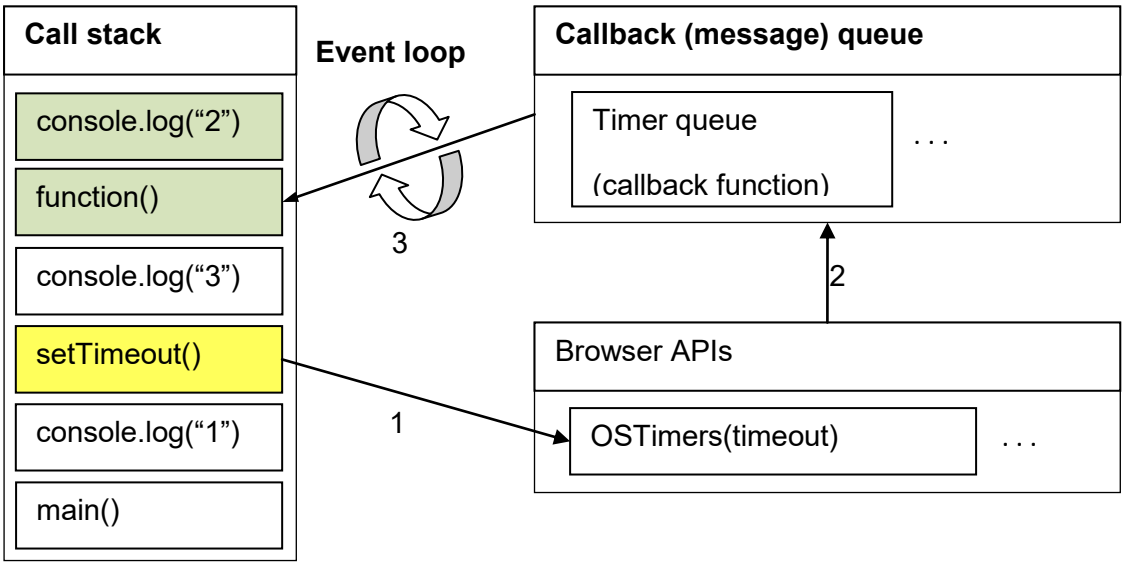
4.4. Асинхронен JavaScript код

JavaScript модулите се изпълняват в рамките на един и същ контекст (програмна нишка). Това означава, че не е възможно конкурентно изпълнение на техния код. Трябва да се прави разлика между конкурентен и асинхронен програмен код. Конкурентен е кодът, който позволява отделни части от приложението да се изпълняват в един и същ момент от време (паралелно или в режим на деление на ресурсите на микропроцесора). Асинхронен е кодът, който позволява отделни части от приложението да се изпълняват непоследователно във времето – не се изисква изчакване на един модул да завърши, за да се подготви за изпълнение друг модул. Конкурентният код по своята същност е асинхронен, но обратното не винаги е вярно.

При езиците от високо ниво, например Java, C++ и C#, конкурентността на кода се реализира най-често чрез използване на програмни нишки (threads). Програмната нишка е функция, която се „вижда" от операционната система като отделен процес и следователно тя има свой собствен контекст. Това позволява нишката да се изпълнява конкурентно на останалите процеси. Всяка нишка има свой собствен стек и програмен приоритет. При JavaScript, програмният код се изпълнява в рамките на една програмна нишка. Това е причината този език да не поддържа конкурентен код. Разбира се, трябва да има основателни причини JavaScript да е еднонишков (single-threaded) програмен език. При многонишковото програмиране са необходими много добри познания за съответния език, за да се пише безопасен (thread-safe) програмен код. В многонишкова среда трябва да се внимава с използването на споделените ресурси (променливи, обекти и функции, които се достъпват от множество нишки едновременно), както и възможността за получаване на така наречената мъртва хватка (dead lock) – взаимно блокиране на няколко нишки, работещи с общи споделени ресурси.

Всички тези проблеми не съществуват в JavaScript, тъй като JavaScript използва само една нишка. Така лесно се гарантира писането на неблокиращ програмен код и използване на по-малко ресурси. JavaScript поддържа асинхронно изпълнение на програмния код, по точно асинхронни функции. Това най-често са функции, които трябва да се извикат при възникване на определено събитие, например: изминал е зададен интервал от време, натиснат или отпуснат е бутон от клавиатурата или мишката, заредена в паметта е картинка или HTML страница, начало на въвеждане на информация в текстово поле, получаване или губене на фокус и много други. Функциите, които се стартират по условие, се наричат callback функции. Тези функции се изпълняват асинхронно, но не конкурентно на кода от останалите функции. Типичен пример за

асинхронни функции в JavaScript са тези, които използват таймери – setTimeout и setInterval.



Фиг. 3-10 Пример за работа на JavaScript при изпълнение на асинхронен код

„Конкурентният" модел на JavaScript се базира на call stack, message queue и event loop (виж Фиг. 3-10):

- **Call stack** - това е област от паметта с достъп Last Input First Output (LIFO), която JavaScript използва с цел да знае коя е текущо изпълняваната функция и коя ще е следващата. В стека се записват съобщения, които описват коя функция трябва да се стартира. За всяка функция, която се вика, първо се записва съобщение за нея във върха на стека. Един запис в стека се нарича рамка (frame). Интерпретаторът на JavaScript започва да изпълнява тази функция, която е записана в стека последна.
- **Message queue** - в тази структура в паметта от тип опашка с достъп First Input First Output (FIFO) се записват само съобщения (messages), които са свързани с изпълнението на callback функции. Всяко ново съобщение се записва в края на опашката. Записът се реализира при детектиране на следено събитие, а изтриването – при възможност за изпълнение на callback функцията. Тази програмна опашка е прието да се нарича message queue или callback queue. За да функционира тази идея е необходим и механизъм, наречен event loop.
- **Event loop** - основната задача на event loop е да изпълнява callback функции в точният момент от време. Той проверява дали call stack е празен (не съдържа функции за изпълнение). Ако стекът не е празен се изчаква. Ако е празен – започва изпълнение на callback функциите, които са в опашката.

В една асинхронна среда е важно да имаме средства за синхронизиране на асинхронно изпълняван програмен код. Основният начин за това са callback функциите. Те обаче имат редица недостатъци:

- Трудно се синхронизира работата на повече от две асинхронни функции.
- Не е предвидена възможност за обработка на грешки, а те са много вероятни особено при реализация на мрежови комуникации.

Тези недостатъци са отстранени при така наречените обещания (promises).

4.4.1 JavaScript обещания

Програмният език JavaScript предлага елегантен начин за създаване на асинхронно работещи модули, както и възможност за синхронизиране на съвместната работа на множество такива модули. За целта се използва интерфейс Promise. Чрез обект, създаден чрез този интерфейс, е възможно да дефинираме асинхронно работещ код, както и да получим информация в момента, когато кодът завърши без или с грешка.

На Фиг. 3-11 е показан синтаксисът на използване на интерфейс Promise.

```
1  var promise = new Promise(function(resolve, reject) {
2      // асинхронен код, изпълнението на който
3      // завършва с уставяване на флаг статус
4      if (статус === "ок") {
5          resolve(обект);
6      }
7      else {
8          reject(обект);
9      }
10 });
```

Фиг. 3-11. Създаване на обект $promise$

Конструкторът (ред 1) изисква един аргумент – анонимна функция с два аргумента - $resolve$ и $reject$. Това са функции, които се викат при изпълнение на асинхронния код без грешка ($resolve$) или с грешка ($reject$). Тези функции изпълняват ролята на $callback$ функции. Обектите, които те предават, са достъпни чрез функция *then*. Тя изисква два атрибута. Първият атрибут е функция, която се вика при изпълнение на кода без грешка. Вторият атрибут е функция, която се вика при изпълнение на кода с грешка:

```
1  promise.then(function(обект) {
2    // успешно изпълнен код (resolve)
3  }, function(обект) {
4    // неуспешно изпълнен код (reject)
5  });
```

Обработката при грешки може да се реализира и чрез функция $catch$. Логиката е същата, както при предходния пример, но този синтаксис е по-разбираем:

```
1  promise.then(function(обект) {
2    // успешно изпълнен код (resolve)
3  }).catch(function(обект) {
4    // неуспешно изпълнен код
5  })
```

Когато трябва да се синхронизира работата на няколко блока (функции) от асинхронен код, за всеки от тях се създава обект $promise$. След това се използва функция all. Към така създадения чрез функция all promise обект може да се приложат функции $then$ и/или $catch$.

```
Promise.all(array_of_promises).then(function() {
  // всички модули са завършили успешно
}, function() {
  // един или повече модули не са
  // завършили успешно
});
```

4.4.2 Web Workers

При JavaScript се предоставя възможност за изпълнение на програмен код във фонов режим под формата на модули, наричани *web workers*. Те частично симулират действието на програмните нишки. Всички по-нови версии на браузърите поддържат технология web workers. Програмният код на web worker е отделен JavaScript файл. Той функционира в *свой глобален контекст*, различен от този на текущия обект $window$. Всеки web worker код има свой *собствен* стек, хийп и message queue. Съществуват два вида web workers според контекста – *несподелени* ($dedicated$) и *споделени* ($shared$). Контекстът на несподелените web worker се описва чрез обект $DedicatedWorkerGlobalScope$, а контекстът на споделените web worker - чрез обект $SharedWorkerGlobalScope$. В първият случай се създава web worker, функциалността на който е достъпна от един скрипт файл. Споделеният web worker е достъпен от множество скриптове, посредством обект с име $port$. Част от браузърите поддържат *subworkers*. Те се получават като от тялото на web worker се създава един или няколко обекта чрез конструктора $Worker$. Задължително трябва да проверите дали можете да създадете този обект, защото браузъри като Chrome и Safari не поддържат subworkers.

За да се изпълнява кода от web workers *паралелно* е необходимо процесорът да е многоядрен. Броят на ядрата (логически) можете да получите чрез следния код:

$$var\ \text{брой}\text{Ядра} = window.navigator.hardwareConcurrency;$$

Следва описание на начина на използване на *несподелени* web workers. Програмно може да установите дали браузърът поддържа web workers като проверите дали се поддържа тип $Worker$:

```
if ( typeof (Worker) === "function") {
    // поддържа се
}
else {
    // не се поддържа
}
```

Обект web worker се създава чрез програмния интерфейс $Worker$. На конструктора се предава пътя и името на файла с кода на web worker:

```
var worker = new Worker("js/worker.js");
```

Кодът на web worker е изолиран от останалия JavaScript код и се изпълнява подобно на отделна програмна нишка със свой собствен контекст. Достъпът до кода на web worker се реализира чрез разширяване на стандартните възможностите на браузърите за генериране и обработка на събития. За да получавате информация от web worker е необходимо да се прехване събитие $message$ и да се зададе име на $callback$ функцията, която ще се активира при наличие на данни от web worker:

```
worker.addEventListener("message", информацияОтWorker, false);
```

В този случай, всеки път когато web worker изпрати данни, ще се извика функция информацияОт$Worker$:

```
1  function информацияОтWorker(данни) {
2      var data = данни.data;
3      // извличане на данни ...
4      // логика ...
5  }
```

Тази функция получава данни – обект с име данни. Достъпът до тези данни (най-често JSON обект) се реализира на ред 2.

Можете да изпратите информация към web worker чрез метод $postMessage$ от интерфейс $Worker$. Методът предава тази информация към контекста на web worker като обект, който се *клонира*. Използва се *структурно клониране* - обектът се сериализира преди предаване и десериализира при приемане. Това гарантира изолация на кода на web worker от кода на останалите JavaScript модули. Можете да създадете свой собствен команден език и формат на отговорите. Следва пример за изпращане на команда $start$ с един параметър $time$. Тази информацията се изпраща като JSON обект:

```
worker.postMessage({command: "start", time: 100});
```

Командата може да се интерпретира по следния начин – активирай кода на web worker за време от 100 ms.

За да може web worker да получава информация, той трябва по аналогичен начин да прехване събитие $message$:

```
1  this.addEventListener("message", function(данни) {
2      var data = данни.data;
3      switch (data.command) {
4          case "start":
5              var time = data.time;
6              // логика ...
7              break;
8          // други команди ...
9      }
10 }
```

Наименованието и броят на командите, като и останалите параметри от JSON обекта, зависят изцяло от логиката на работа на web worker и е задача, която се решава от програмиста.

Когато web worker изпълни логиката, която му е заложена, неговия програмен код трябва да бъде премахнат от паметта. Това може да се реализира по два основни начина:

1. Изпращане чрез $postMessage$ на команда за унищожаване. При разпознаване на тази команда web worker трябва да се самоунищожи чрез следния код:

```
self.close();
```

2. Унищожаване на web worker с код извън неговия контекст. Това се реализира чрез метод $terminate$ от интерфейс $Worker$:

```
worker.terminate();
```

Кой метод ще използвате зависи от конкретната разработка и логика на функциониране на web worker, но за предпочитане е първият.

От тялото на web worker *нямате достъп до всички обекти*, които са създадени от браузъра с цел сигурност на кода. Имате достъп до следните по-важни обекти: $Navigator$, $Array$, $Date$, $String$, $Math$, $XMLHttpRequest$, $WindowTimers$ (методи $setTimeout$, $setInterval$, $clearTimeout$ и $clearInterval$), $requestAnimationFrame$.

Както вече знаем кодът на един web worker трябва да е в отделен JavaScript файл. Ако Вашата цел е кодът на web worker и на модула, който го създава, да бъдат в един файл е необходимо да използвате *inline web worker*. Той се създава като на конструктора $Worker$ се подава $Blob$ (Binary Large Object) обект. Чрез него се създава URL манипулатор към кода на web worker:

```
1  var blob = new Blob(["onmessage = function(e) { "+
2      "console.log(e.data);"+
3      "postMessage('Информация от web worker'); }"]);
4
5  var blobHandle = window.URL.createObjectURL(blob);
6  var worker = new Worker(blobHandle);
7  worker.onmessage = function(event) {
8      console.log(event.data);
9  };
10 worker.postMessage("Информация за web worker");
```

Заключение

Тази глава има за цел да запознае читателите, които нямат или имат ограничени познания за Web технологии с предназначението и синтаксиса на HTML5, CSS3 и JavaScript. Тези знания ще ви бъдат необходими, за да може да разбирате програмния код от готовите проекти към тази книга, както и да разработвате собствени проекти.

Глава 4

Развойна среда Monaca

I. Въведение

Платформата Monaca е разработка на японската компания Asial Corporation. Тя е колекция от софтуерни инструменти и услуги с цел изграждане и внедряване на мобилни хибридни приложения. Изградена с помощта на програмната рамка Apache Cordova, тя предоставя всички ресурси, необходими за разработване на мобилни приложения:

- Облачно базирана среда за разработка – Monaca Cloud IDE.
- Локални инструменти за разработка – Monaca Localkit и Monaca CLI.
- Мобилно приложение – дебъгер, което работи синхронно с Monaca Cloud IDE и Monaca CLI.
- Back-end поддръжка.

Базираният в облака IDE на Monaca изгражда хибридни мобилни приложения за Android, iOS, Windows и Web с помощта на HTML5, CSS3 и JavaScript. Monaca е фреймуърк-агностична платформа и осигурява интеграция с програмни рамки като Onsen UI, Ionic, Framework7, React и Vue с цел изграждане на потребителския интерфейс. Облачната платформа Monaca позволява и импортиране на вече съществуващи PhoneGap проекти. Използвайки Monaca не се налага да инсталирате каквото и да е програмно осигуряване, освен дебъгера.

Имате възможност да компилирате разработваното от вас приложение за ОС Android, iOS и Windows без да се налага да имате достъп до специфична апаратна част или да инсталирате необходимите SDK за конкретна ОС. Може да създавате и прогресивни Web приложения (PWA), както и да ги хостване автоматично на няколко палтформи, например Google Firebase. Ако сте регистриран разработчик за iOS или Android може лесно да подготвите необходимите файлове за качване на приложението в съответния магазин за приложения.

Използването на платформата за комерсиална дейност предполага заплащане на месечни такси, но те са много по-малки при сравнение с други подобни услуги. Имате и възможност да използвате платформата безплатно, но при множество ограничения, например: не можете да импортирате и експортирате проекти; не може да използвате плъгини на трети страни; не може да компилирате приложението повече от 3 пъти за 24 часа и други.

II. Развойната среда

За да използвате Monaca трябва да се регистрирате с валиден адрес за електронна поща. Автоматично ще получите 14 дневен период през който може да използвате *пълните функционални възможности* на платформата *безплатно*. След този период трябва да потвърдите, че преминавате на безплатен план или на избран платен план. Развойната среда Monaca предоставя четири различни среди за разработка: Monaca Cloud IDE, Monaca Localkit, Monaca CLI и Monaca Debbuger.

2.1 Monaca Cloud IDE

Monaca Cloud IDE е среда за разработка, която работи под управлението на браузър. Това позволява разработка на Cordova и PhoneGap проекти без никакви настройки. Тази развойна среда интегрира всички необходими програмни инструменти с цел отдалечено изграждане на хибридни мобилни приложения. Заедно с Monaca Debugger и Live Preview (вградена функция в Monaca Cloud IDE) можете да следите лесно напредъка на вашето приложение по време на разработка.

2.2 Monaca Localkit

Monaca Localkit е инструмент за поддръжка на разработката на приложения локално – на машината на разработчика. Можете да постигнете по-добра сигурност и спокойствие от собствената си среда за разработка на приложения, като комбинирате съществуващите инструменти, като редактори на код и система за управление на версии, с универсалните възможности за поддръжка на разработката на Monaca. С помощта на Monaca Localkit потребителите на Monaca могат да настроят предпочитаната от тях среда с цел разработка на локален компютър.

2.3 Monaca CLI

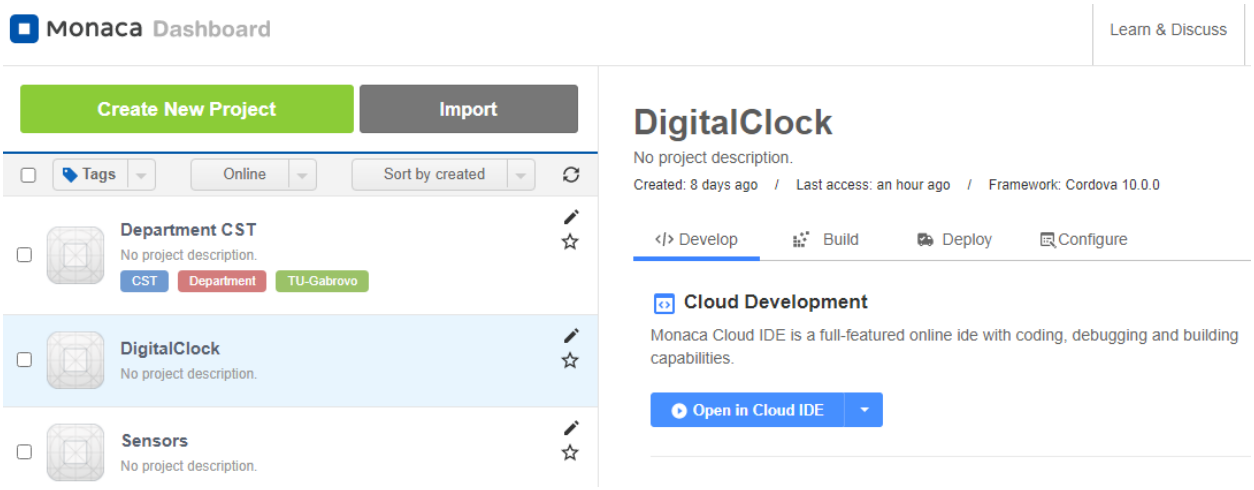
Monaca CLI предоставя интерфейс от командния ред за използване на Monaca Cloud IDE. Имате на разположение команди за създаване на проекти, свързване с Monaca Debbuger, както ии отдалечено изграждане на приложението. Можете също така да импортирате проекти, които съществуват в облака, когато искате да превключите към работа локално.

2.4 Monaca Debbuger

Освен гъвкавостта на средата за разработка, Monaca осигурява и висока ефективност на разработката чрез използването на Monaca Debugger. Това е мобилно приложение, което се използва за тестване и отстраняване на грешки. Работи върху реални устройства, без да се налага да изграждате приложенията по време на разработката. Приложението автоматично ще синхронизира всички ваши проекти (локални и в облака) и ще ги стартира без процеса на изграждане.

III. Работа с развойната среда

За да не се налага да правим инсталации на програмно осигуряване, ще работим с облачната платформа Monaca Cloud IDE. След проверка на автентичността на потребителя (възможно е и чрез GitHub акаунт) ще получите достъп до развойната среда, по-точно до нейния потребителския интерфейс (dashboard) – виж Фиг. 4-1.

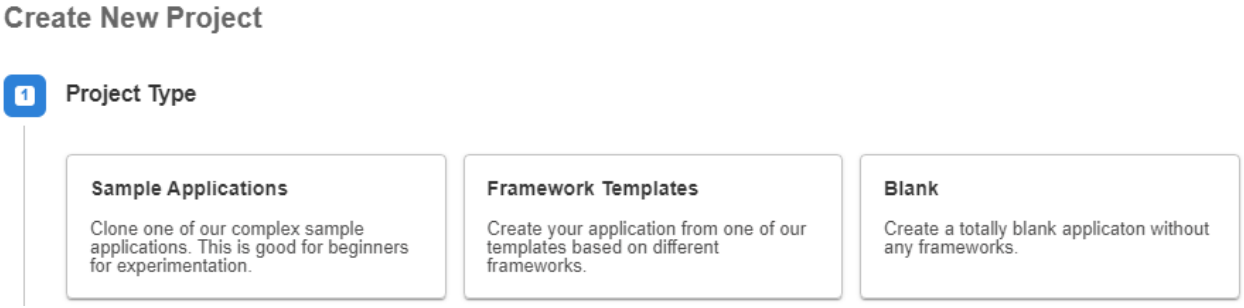


Фиг. 4-1 Потребителски интерфейс на Monaca Cloud IDE

3.1 Създаване на нов или импортиране на проект

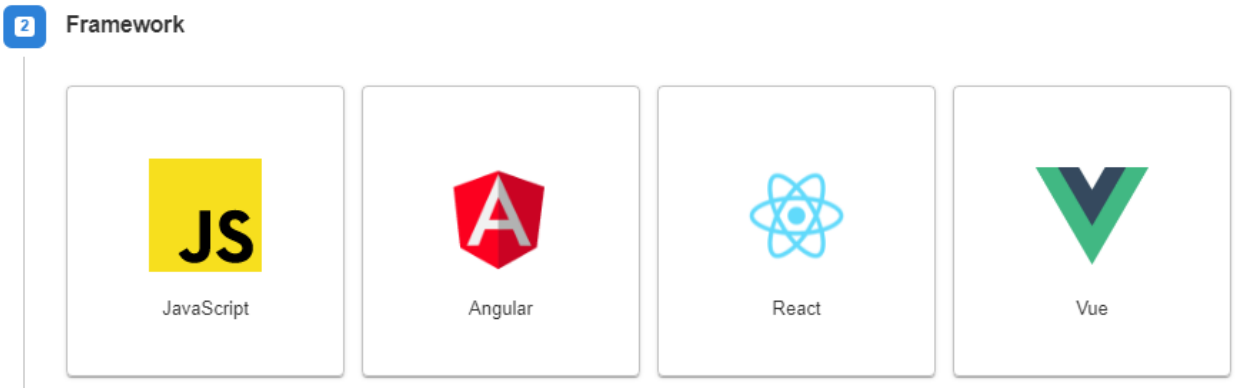
Може да започнете със създаване на нов проект (Create New Project) или да импортирате вече съществуващ проект (Import). Ако създавате нов проект имате следните възможности (виж Фиг. 4-2):

- Да клонирате вече готов проект – Sample Application.
- Да използвате шаблон за дадена хибридна мобилна рамка – Framework Template.
- Да започнете с празен проект – Blank.



Фиг. 4-2 Създаване на нов проект

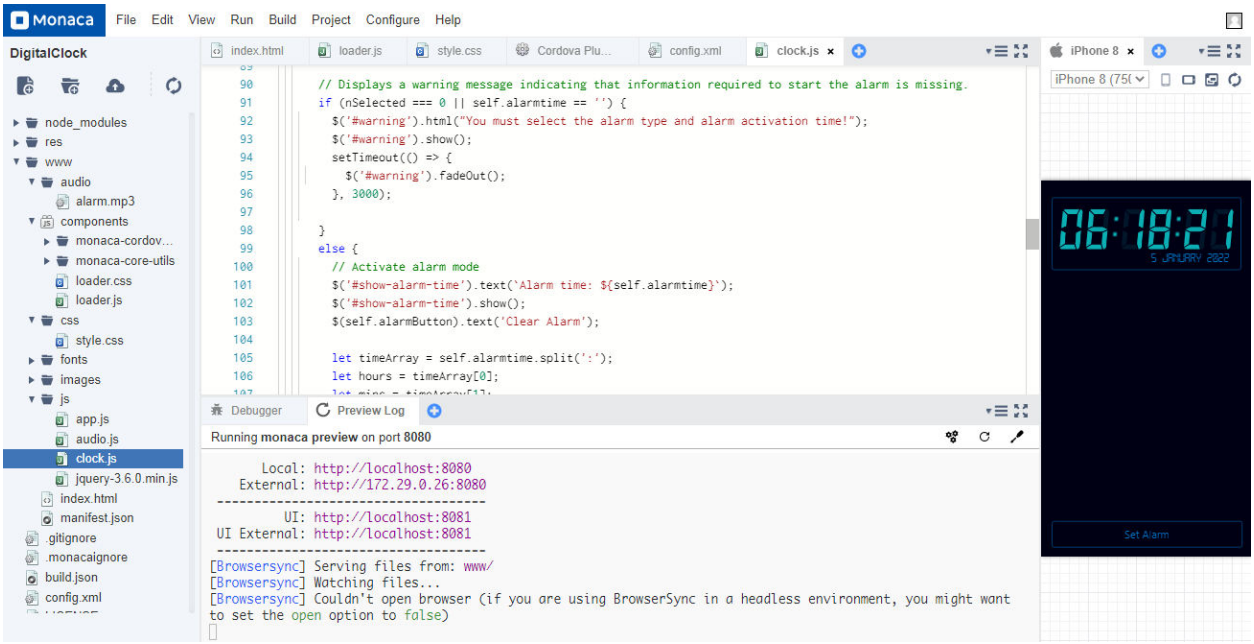
В повечето случаи е подходящо да изберете шаблон за желаната от вас хибридна мобилна рамка. Развойната среда Monaca по подразбиране работи с Onset UI, но можете да изберете Framework7, Framework7 Svelte, AngularJS (Ionic), Onset + React, Framework7 + React, Onset + Vue и Framework + Vue (виж Фиг. 4-3).



Фиг. 4-3 Избор на програмна рамка за изграждане на изгледа на приложението

3.2 Писане на програмен код

След като сте избрали програмен стек сте готови да пишете програмен код. За целта развойната среда Monaca Cloud IDE предоставя редактор за JavaScript, HTML5, CSS3, XML и JSON. Имате вградена помощна информация за синтаксиса при всеки един от тези езици. Лесно можете да форматирате кода (автоматично подреждане) чрез комбинацията от бутони Ctrl+Alt+F. Средата има и вграден емулатор чрез който може да тествате в реално време работата на приложението (виж Фиг. 4-4).



Фиг. 4-4 Графичен редактор и емулатор на Monaca Cloud IDE

При писане на CSS код имате на разположение вграден Color Picker чрез който лесно може да получите цифровото кодиране на желания от вас цвят (виж Фиг. 4-5).

Фиг. 4-5 Color Picker на Monaca Cloud IDE

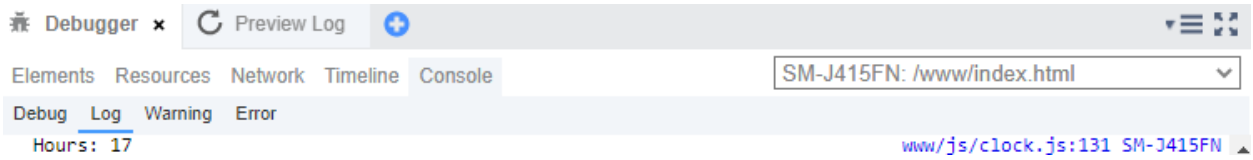
3.3 Откриване и отстраняване на грешки

Основният начин за отстраняване на грешки в приложенията с Monaca Cloud IDE е така нареченото конзолно отстраняване. Развойната среда има панел за отстраняване на грешки Debug. В този панел е вграден популярен инструмент за отстраняване на грешки в Web страници - WEb INspector REmote (Weinre). Този инструмент ви позволява да

отстранявате грешки във вашето приложение, като използвате конзолно отстраняване на грешки и проверка на DOM. Конзолният приложен програмен интерфейс на Monaca позволява да записвате и показвате съобщения в конзолата, като използвате JavaScript. Функциите, чрез които може да визуализирате текст на конзолата са следните:

- console.log() - извежда съобщение в конзолата.
- console.debug() - извежда съобщение на ниво за отстраняване на грешки (можете да видите съобщението в раздела за отстраняване на грешки).
- console.warn() - извежда съобщение с жълта икона за предупреждение.

За да виждате какво записват тези функции в конзолата на Dubug панела е необходимо да инсталирането безплатното мобилното приложение *Monaca Debugger*. Това е приложение за отстраняване на грешки, което може да провери функционалността на разработваното приложение на реално устройство, без да е необходимо то да се компилира (изгражда). Когато използвате Monaca Debugger, всички промени, направени в кода, веднага се отразяват на действителното работещо устройство. Използва се Live Preview – след запис на промените в кода, приложението автоматично се презарежда. По този начин се ускорява процесът на разработка и тестване. Всичко, което показвате на конзолата чрез функции log, debug и warn се вижда в конзолата на Dubug панела на Monaca Cloud IDE, както и на екрана на Monaca Debugger (виж Фиг. 4-6).

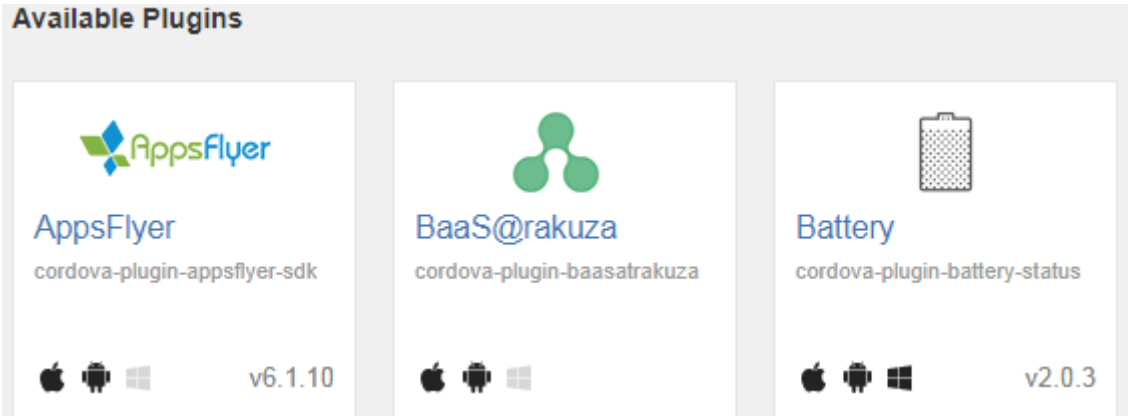


Фиг. 4-6 Използване на режим Debugger

Трябва да имате предвид, че Monaca Debugger не поддържа пълната функционалност на реалното приложение – някои функции не могат да се емулират. Освен това приложението работи по-бавно на Monaca Debugger, отколкото ако е инсталирано след изграждане.

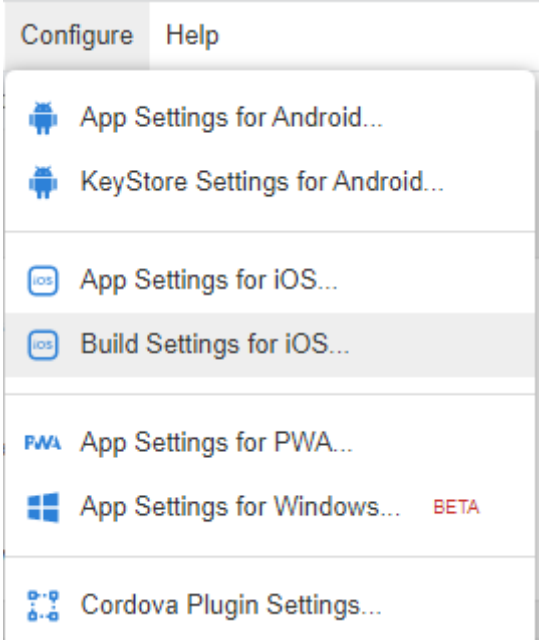
3.4 Изграждане на приложението

След като сте отстранили всички синтактични и логически грешки трябва да компилирате (изградите) приложението до програмен пакет за инсталация за съответната платформа. Преди това трябва да изберете плъгините с които искате да работите. Развойната среда работи с два типа плъгини: едните са част от платформата, а други са писани от трети страни. Последните се интегрират към проекта, но само ако сте на платен план. На Фиг. 4-7 са показани част от плъгините, които са част от развойната среда.



Фиг. 4-7 Част от интегрираните в Monaca IDE плъгини

Накрая, трябва да въведете необходимата за изграждането информация за приложението – App Settings (виж Фиг. 4-8).

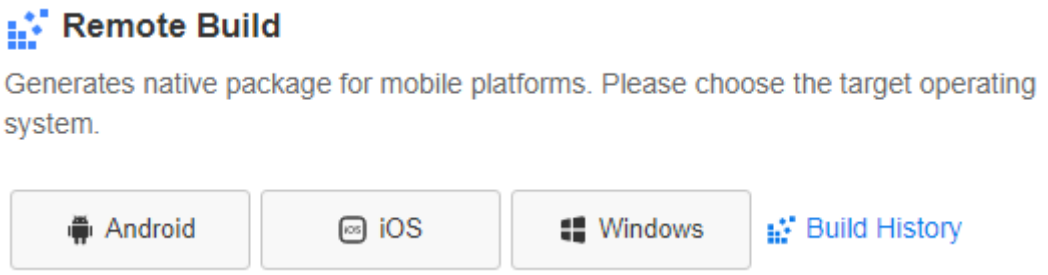


Фиг. 4-8 Избор на задаване на конфигурационна информация за приложението

Информацията, която трябва да въведете, е специфична за всяка платформа, например:

- Име на приложението.
- Име на програмния пакет (package name) – името трябва да е уникално и да спазва синтаксиса на домейн, но в обратен ред, например: bg.tugab.kst.digitalclock.
- Версия на приложението, например 1.0.5.
- Версия на SDK с която искате да бъде изградено приложението. Трябва да имате предвид, че някои функционални възможности са достъпни само за определени версии на SDK. Освен това, всеки магазин за приложение има специфични изисквания за минималната версия на SDK.
- Икони на приложението (icons).
- Картинка за началния екран на приложението (splash screen).
- Да работи ли приложението на фонов режим или не.
- Начална ориентация на екрана (portrait, landscape).

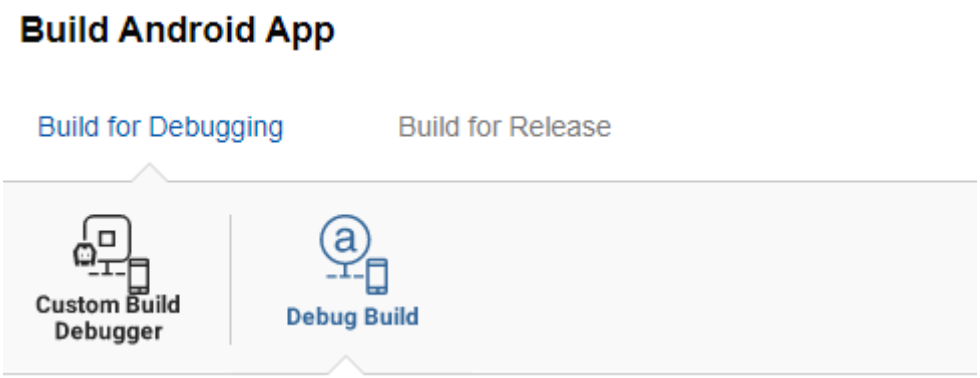
Тази информация е достатъчна само за изграждане на приложението с цел тестване. На този етап Monaca позволява изграждане на приложения за Android, iOS и Windows (виж Фиг. 4-9).



Фиг. 4-9 Изграждане на приложението чрез Monaca Cloud IDE

Изграждането е автоматизирано и не изисква да инсталирате какъвто и да е SDK или да имате специфична платформа (компютър и ОС), необходима например за изграждане на приложения за iOS. Може да избирате между няколко вида изграждане (виж Фиг. 4-10):

- Изграждане с цел тестване на приложението на реално устройство – Debug Build.
- Изграждане с цел тестване на приложението на реално устройство и работа с външни плъгини – Custom Build Debugger.
- Изграждане с цел публикуване в магазин за приложения – Build for Release.



Фиг. 4-10 Избор на тип изграждане на приложението

3.5 Качване на приложението в Google Play

Ако трябва да изградите приложението с цел пускане за сваляне от магазин за приложения (Build for Release) трябва да преминете през още стъпки, които са специфични за всяка мобилна ОС. Например, за ОС Android трябва да преминете през конфигуриране на Android KeyStore. KeyStore е двоичен файл, който съдържа набор от частни ключове. Частният ключ представлява субект, който трябва да бъде идентифициран с приложението, например частно лице или компания. KeyStore е криптиран с парола и не може да бъде възстановен, ако паролата бъде загубена. Когато KeyStore се загуби или презапише друг KeyStore, е невъзможно да се използва същият ключ за повторно подписване на програмния пакет. KeyStore е необходим за създаването на версия за пускане на вашето приложение за Android.

Чрез Monaca IDE можете да създадете нов KeyStore или да импортирате съществуващ такъв. За да създадете нов KeyStore от менюто изберете Configure → Android KeyStore Settings. Ще се отвори страницата Manage KeyStore and Alias. Попълнете следната информация:

- Alias name (псевдоним): името, представляващо частния ключ, който ще използвате по-късно при подписването на приложението. В един KeyStore могат да се съхраняват няколко псевдонима.
- Password (парола): паролата за частния ключ (псевдоним).
- Password for KeyStore (парола на хранилището на ключове): тази парола ще ви е необходима при импортирането на KeyStore.

След това, щракнете върху бутона <Generate KeyStore and Alias> (Генериране на хранилище на ключове и псевдоними), за да генерирате хранилището на ключове.

Когато хранилището на ключове е изгубено, е невъзможно да се използва същия ключ за повторно подписване на подписания пакет. Затова винаги създавайте резервно копие и

пазете KeyStore, който се използва за подписване на приложението(ята). Използвайте бутона <Export> (Експортиране), за да запазите локално своя KeyStore.

Възможно е да се наложи да изпълните няколко други задачи като част от процеса на подготовка за пускане на приложението. Например, ще трябва да получите частен ключ за подписване на пакета. Също така ще трябва да създадете икона за вашето приложение и може да искате да подготвите лицензионно споразумение с краен потребител (EULA), за да защитите своята личност, организация и интелектуална собственост. Когато приключите с подготовката на приложението си за пускане, ще имате подписан APK файл, който можете да разпространявате сред потребителите.

Ако искате да разпространявате приложенията си сред възможно най-широка аудитория, трябва да качите приложението в Google Play. Google Play е основният магазин за приложения за Android. Google Play е платформа за публикуване, която ви помага да популяризирате, продавате и разпространявате вашите приложения за Android сред потребителите по целия свят. Когато публикувате приложенията си чрез Google Play, имате достъп до набор от инструменти за разработчици, които ви позволяват да анализирате продажбите си, да определяте пазарните тенденции и да контролирате до кого се разпространяват приложенията ви. Имате достъп и до няколко функции за увеличаване на приходите, като например таксуване в приложението и лицензиране на приложения.

Системата за фактуриране в Google Play е услуга, която ви позволява да продавате цифрови продукти и съдържание в приложението си за Android. Можете да използвате системата за таксуване на Google Play, за да продавате следните видове цифрово съдържание:

1. Еднократни продукти

Еднократен продукт е съдържание, което потребителите могат да закупят с еднократна, неповтаряща се такса към формата на плащане на потребителя. Еднократните продукти могат да бъдат както потребителски, така и неконсумативни. Продукт, който може да се консумира, е продукт, който потребителят консумира, за да получи съдържание в приложението. Когато потребителят консумира продукта, приложението ви предоставя свързаното съдържание, а потребителят може след това да закупи продукта отново. Продукт, който не може да се консумира, е продукт, който се купува само веднъж, за да осигури постоянна полза.

2. Абонаменти

Абонамент е продукт, който осигурява достъп до съдържание на периодична основа. Абонаментите се подновяват автоматично, докато не бъдат отменени. Примерите за абонаменти включват достъп до онлайн списания и услуги за стрийминг на музика и видео съдържание. Конзолата на Google Play предлага голяма гъвкавост при създаването на абонаментни продукти. Като примери можете да зададете периода на фактуриране, да предложите безплатна пробна версия, да предложите въвеждаща цена, да осигурите гратисни периоди при неуспешно плащане и да позволите на потребителите да приведат абонаментите си на пауза като алтернатива на отмяната на абонамента.

Google Play предлага услуга за лицензиране, която ви позволява да прилагате политики за лицензиране на приложенията, които публикувате в Google Play. С Google Play Licensing вашето приложение може да направи запитване към Google Play по време на изпълнение, за да получи информация за състоянието на лицензиране на текущия

потребител, след което да разреши или забрани по-нататъшното използване, както е необходимо.

С помощта на услугата можете да прилагате гъвкава лицензионна политика за всяко отделно приложение - всяко приложение може да прилага лицензиране по най-подходящия за него начин. Ако е необходимо, дадено приложение може да прилага потребителски ограничения въз основа на състоянието на лицензиране, получено от Google Play. Например, дадено приложение може да провери състоянието на лицензиране и след това да приложи потребителски ограничения, които позволяват на потребителя да го стартира без лиценз за определен период на валидност. В допълнение към всички други ограничения приложението може да ограничи използването до конкретно устройство. Услугата за лицензиране е сигурно средство за контрол на достъпа до вашите приложения. Когато дадено приложение проверява състоянието на лицензиране, сървърът на Google Play подписва отговора за състоянието на лицензиране, като използва двойка ключове, която е уникално свързана с приложението. Въпреки че е възможно вашето приложение да съхранява публичния ключ в APK файла, много по-безопасно е да проверявате отговора за състоянието на лицензиране на сървър, на който имате доверие. Всяко приложение, което публикувате чрез Google Play, може да използва услугата Google Play Licensing. Не е необходим специален акаунт или регистрация.

Пускането на приложението ви в Google Play е процес, който включва няколко основни стъпки:

3.5.1. Регистрация като разработчик

Трябва да се регистрирате като разработчик на приложения за ОС Android. Използвайте конзолата на Goggle Play:

https://play.google.com/console/signup

Трябва да имате акаунт в Gmail, за да продължите с регистрацията. Трябва да извършите плащане на еднократна такса на стойност 25 долара. Приемете „Споразумението за програмисти“. След това трябва да въведете данните на вашата банкова карта, за да се извърши плащането. След регистрация ще получите достъп до контролния панел на Developer Console. Качването на приложението става чрез бутон <Upload new APK to Production>. Следва въвеждане на името на приложението и езика по подразбиране. Имате три възможности за качване на APK файла, което зависи от фазата, в която се намирате във вашия процес:

- Alpha testing.
- Beta testing.
- Production.

Алфа и бета са различни фази от жизнения цикъл на разработката, определени от напредъка на разработката на приложението. Те се характеризират с различни степени на стабилност, използваемост и завършеност на функциите. *Алфа* версия е първата работеща версия на вашето приложение, която невинаги разполага с всички заложени функции. Може да съдържа грешки. Разработчиците добавят всички липсващи функции и се фокусират върху намирането, проследяването и отстраняването на възможно най-много грешки. Когато приложението стане "функционално завършено" и не останат сериозни сривове или грешки, алфа фазата се счита за завършена и окончателната версия се превръща в първата бета версия.

Традиционно алфа тестовете се провеждат вътрешно във фирмата разработчик, без да се включва никой извън цикъла на разработка. Използването на външни тестери или дори крайни потребители за алфа фазата обаче не е нещо необичайно. Всъщност, напоследък то става сравнително по-популярно сред по-малките екипи за разработка, които не разполагат със специален екип за контрол на качеството. По време на алфа и бета фазите разработчиците могат да добавят нов код към приложението и могат да правят нови компилации, за да го тестват. С други думи, фазите алфа и бета се състоят от много итеративни алфа и бета изграждания.

Бета е версията, с която се отбелязва завършването на разработката на приложението. Традиционно на този етап не се добавят нови функции, а фокусът се поставя върху подобряване на качеството, използваемостта и цялостното изживяване на потребителите. Бета версията е първата налична за потребителите версия извън разработката и все още може да има грешки и проблеми със стабилността. Въпреки това, тя не трябва да има сериозни проблеми и се счита за напълно функционална, използваема версия. Програмите за бета тестване могат да бъдат затворени, като достъпът до тях е ограничен до поканени членове, или отворени, в които може да участва всеки. Така или иначе, приложението се тества в "реална среда" с "реални сценарии" от външни тестери и потребители. Поради по-голямото разнообразие на средите на устройствата и моделите на използване, това помага да се открият много грешки, които са останали през по-ранните фази на тестване. След това разработчиците ще използват подадените доклади за грешки и сривове, за да определят и отстранят всички докладвани дефекти и да повторят компилациите, докато приложението не бъде готово за пускане. Тъй като при бета тестването потребителите за първи път могат да изпробват приложението, това е първата възможност за валидиране.

На този етап се реализират така наречените *нощни изграждания*. Нощната версия се променя всяка вечер. Изграждането се прави в края на деня, като включва всички промени, направени през този ден. Те не са ограничени до нито един етап от жизнения цикъл на разработката и се използват през целия цикъл, т.е. по време на алфа и бета тестване. Нощните компилации обикновено не са достъпни за никого извън жизнения цикъл на разработката, но понякога се споделят с избрани външни тестери. Те се намират в "най-напредналата фаза" на разработката. Нощната сглобка към края на бета цикъла трябва да е стабилна.

Когато бета фазата на тестване е завършена се получава така наречената „сребърна” версия на приложението, която може да бъде "кандидат за пускане" На този етап разработчиците не правят никакви промени в приложението, освен дребни поправки на кода и подобрения в документацията. Когато тези последни щрихи са завършени, компилацията преминава в производство и се разпространява до крайните потребители. Това е известно още като „златен кандидат”.

3.5.2. Подготовка на рекламни материали

За да се възползвате напълно от възможностите за маркетинг и реклама на Google Play, трябва да създадете рекламни материали за приложението си, като например снимки на екрани, видеоклипове, графики и рекламен текст. Първо, трябва да добавим „кратко описание” на приложението с максимум 80 символа. Това описание се вижда от потребителите на Play Store. След това трябва да се попълни „пълното описание”. Важно е тук да се показва всичко, което потребителят трябва да знае за това. Може да използвате текст до 4000 символа.

Трябва да се качат екранни снимки, демонстриращи работата на приложението. Те са тези, които потребителите ще видят в магазина. Уверете се, че те са ясни и можете да видите интерфейса на приложението как работи. Освен това трябва да качим икона на приложението, която задължително трябва да бъде във формат PNG. Следва раздел „Категоризация” където трябва да обясним за какво е приложението ни.

Трябва да зададете дали искате приложението да е платено или безплатно. Важно е да знаете, че безплатните приложения са по-успешни в Play Store, тъй като бързо достигат до много потребители. Затова залагането на нещо безплатно, но с реклами или покупки вътре в него, ви дава повече шансове за финансов успех. Google Play ви позволява да насочите приложението си към огромна група от потребители. Чрез конфигуриране на различни настройки на Google Play можете да изберете държавите, които искате да достигнете, езиците на листване, които искате да използвате, и цената, която искате да начислявате във всяка държава. Можете също така да конфигурирате детайли на списъка, като например тип на приложението, категория и рейтинг на съдържанието.

3.5.3. Такси за обслужване

Програмистите подлежат на такса за обслужване в Google Play въз основа на процент от цената на покупката или дигиталните покупки в приложението им. Само 3% от програмистите в Google Play са предмет на такса за обслужване, останалите 97% могат да разпространяват приложенията си и да се възползват от всичко, което Google Play предлага, без парично заплащане. Таксата за обслужване отразява стойността, предоставена от Android и Google Play, и всички услуги за програмисти, които се предлагат, включително разпространение и откриване на приложения, платформата за търговия, инструменти за програмисти, анализ, обучение и др. От 1 януари 2022 г. таксата за обслужване се определя по следния начин:

- За програмистите, които са регистрирани за таксата за обслужване от 15%,, таксата е 15% за първия спечелен 1 милион USD всяка година; 30% за приходи над 1 милион USD всяка година. За програмистите, които не са регистрирани за таксата за обслужване от 15%, таксата е фиксирана на 30%.
- За продуктите с автоматично подновяващ се абонамент, таксата за обслужване е фиксирана на 15%.

Заключение

От тази глава научихте какво е предназначението и възможностите на развойна среда Monaca (Monaca IDE, Monaca Localkit, Monaca CLI, Monaca Debugger). Научихте каква е последователността на създаване на проект в тази среда, от конфигуриране до изграждане на проекта. Описана бе последователността от действия, която трябва да се реализира, за да може успешно да регистрирате мобилно приложение за ОС Android в магазина за приложения Google Play.

Глава 5

Първо приложение

I. Въведение

Ще използваме Monaca Cloud IDE с цел създаване на хибридно мобилно приложение за ОС Android. Знанията, които са необходими за разработката са свързани с Web технологии - HTML5, CSS3 и JavaScript. Този код се стартира под управление на клас WebView от страна на мобилното устройство. Чрез него се реализира рендиране и визуализиране на Web страници.

Един от основните недостатъци на хибридните приложения, които се базират на Web технологии, е свързан с тяхното бързодействие. При повечето разработки се налага динамично презареждане на съдържанието на страниците, вмъкване и изтриване на елементи от HTML кода или промяна на свойствата на тези елементи, за да се реализира желана интерактивност. За да е възможно всичко това е необходимо да се предостави *междуплатформен и независим от езика стандартизиран интерфейс* между HTML обектите и езика за програмиране - JavaScript. За целта при браузърите се използва *моделът на обекта на документа* – Document Object Model (DOM). За стандартизацията на DOM отговарят W3C и WHATWG. При зареждане на всяка HTML страница, браузърът създава обектно-ориентирано представяне на страницата – DOM. Това позволява създаването на динамични страници, тъй като в рамките на една страница JavaScript може:

- Да добавя и премахва обекти.
- Да променя стиловото форматиране на обектите.
- Да реагира на всички събития, генерирани от обектите.
- Да създава нови събития.

За да е възможно всичко това DOM представя всеки HTML документ като логическо дърво. Всяко разклонение на дървото завършва с възел, а всеки възел съдържа обекти. Методите на DOM позволяват програмен достъп до дървото. С тях можете да променяте структурата, стила или съдържанието на документа. Към възлите могат да бъдат прикрепени и функции, обработващи събития (callback функции). След като бъде задействано дадено събитие (например натискане на бутон), се изпълнява свързаната с него callback функция.

Тъй като DOM се описва чрез дървовидна структура от данни, промените и актуализациите на обектите от DOM са бързи. Но след промяната актуализираният обект и неговите дъщерни обекти трябва да се визуализират отново, за да се актуализира потребителският интерфейс на приложението. Повторното визуализиране на потребителския интерфейс е това, което прави промените в DOM бавни. Ето защо, колкото повече обекти на потребителския интерфейс имате, толкова по-бавни са актуализациите на DOM, тъй като те ще трябва да се рендерират отново при всяка актуализация на DOM.

За да се повиши бързодействието при работа с DOM екипът на React първи предлага решение за използване на така наречения *виртуален* DOM (VDOM). Виртуалният DOM е

само виртуално представяне на реалния DOM. Всеки път, когато състоянието на приложението се променя, виртуалният DOM се актуализира вместо реалния DOM. Това прави VDOM много по-бърз и ефективен. Програмна рамка Framework7 също поддържа VDOM от версия 3.1.0. Библиотеката VDOM се нарича Snabbdom, защото е изключително лека, бърза и се вписва чудесно в средата на Framework7.

При някои програмни рамки всяка част от потребителския интерфейс е *компонент* и всеки компонент има състояние. Компонентите осигуряват силна капсулация и позволяват възможност за многократна употреба. Основните технологии, използвани за създаването на компоненти, включват:

- API за създаване на нови HTML елементи.
- Shadow DOM.
- HTML документи-шаблони.

DOM в сянка (Shadow DOM) се използва с цел капсулиране на изпълнението. При Shadow DOM се работи с поддърво от реалния DOM, което съдържа променени компоненти. Шаблоните са фрагменти от HTML код, които не се визуализират, а се съхраняват, докато не се инстанцират чрез JavaScript код. Този код генерира динамично липсващия HTML код, веднага след зареждане на страницата.

Повечето рамки за изграждане на потребителския интерфейс използват *компонентен модел* (React, Vue, Ionic4+, Framework7). Компонентите са изградени с HTML, CSS и JavaScript и могат лесно да бъдат внедрени в устройства с различни операционни системи като Android, iOS, или в прогресивно Web приложение (PWA). Програмната рамка, базирана на компоненти, слуша за промени в тяхното състояние. Когато състоянието на компонент се промени, рамката актуализира дървото на виртуалния DOM. След като виртуалният DOM бъде актуализиран, рамката сравнява текущата версия на виртуалния DOM с предишната версия на виртуалния DOM. След като се разбере кои виртуални DOM обекти са се променили, се актуализират само тези обекти в реалния DOM. Този процес се нарича "различаване" (diffing). Това прави производителността много по-добра в сравнение с директното манипулиране на реалния DOM. Всичко, което трябва програмистът да направи, е да актуализира състоянията на желания компонент, а рамката ще се грижи кога тази промяна да се отрази в реалния DOM. За да се оптимизира бързодействието повечето рамки използват механизъм за *пакетно актуализиране* на реалния DOM. Това означава, че актуализациите на реалния DOM се изпращат на порции, вместо да се изпращат актуализации за всяка отделна промяна в състоянието.

II. Условие на задачата

Да се напише, тества и изгради хибридно мобилно приложение за ОС Android, което визуализира цифров часовник с възможност за аларма. Алармата да се активира в зададен час и минути и да позволява възпроизвеждане на аудио (mp3) файл и/или вибрация за определен интервал от време. Да се предвиди спиране на възпроизвеждането на аудио файла, преди да е завършило възпроизвеждането чрез докосване на екрана на мобилното устройство. Да не се използва програмна рамка за изграждане на потребителския интерфейс.

III. Реализация на задачата

Условието да не се използва програмна рамка за изграждане на потребителския интерфейс предполага директна манипулация на DOM, за да постигнем желаната динамична функционалност на страниците. За да реализираме достъпа до реалния DOM с по-малко и по-разбираем програмен код ще използваме библиотека **jQuery**, Библиотека jQuery е една от най-често използваните библиотеки при изграждане на Web приложения. Получаването на достъп до DOM обект е свързано с викане на функция $ на която се предава като аргумент валиден CSS селектор, който идентифицира обекта. По този начин може да изберете конкретен обект (използване на атрибут `id` за избрания HTML етикет) или група от обекти (използване на атрибут `class` за избрания HTML етикет). Функция $ връща обект чрез който може да манипулирате DOM обектите чрез наличните за това jQuery методи, например text, html, val, attr, click и др. За да използвате jQuery е необходимо да сте сигурни, че тя вече е заредена в паметта. За целта може да използвате метод `ready` (вариант 1) или по-кратките варианти на този код, които използват анонимна (вариант 2) или лабмда (вариант 3) функция - виж Фиг. 5-1.

```
// Вариант 1
$(document).ready(function() {
   // Програмен код, който използва jQuery
});
// Вариант 2
$(function() {
   // Програмен код, който използва jQuery
});
// Вариант 3
$(() => {
   // Програмен код, който използва jQuery
});
```

Фиг. 5-1 Прехващане на събитието „Библиотеката jQuery е заредена в паметта”

3.1 Потребителски интерфейс

Ще започнем с цифровата форма на скицата (wireframe), която представя потребителския интерфейс. Трябва да отчетете наличието на два типа изгледи при мобилните устройства - portrait и landscape. Потребителският интерфейс трябва да изглежда еднакво добре и в двата режима. Ще позиционираме цифровия часовник в горния край на дисплея и непосредствено под него ще покажем деня от месеца, името на месеца и годината. В долния край на екрана ще *фиксираме* обектите, чрез които да може да се активира и деактивира алармата. Задаването на часа за алармата и избора на аудио и/или вибрация да се реализира след натискане на бутон <Set Alarm>. Деактивирането на алармата да се реализира чрез бутон <Clear Alarm>. Тези бутони трябва да се визуализират алтернативно. Ще създадем форма за задаване на аларма. Тя ще съдържа поле за отметка [Audio], поле за отметка [Vibrate], поле за въвеждане на часа на алармата във формат hh:mm и бутон за активиране на алармата <Activate>. При активирана аларма под часа и датата да се показва часа, зададен за аларма, във формат “Alarm: hh:mm”.

Следва избор на стилово форматиране на обектите от потребителския интерфейс. Ще използваме синя цветова палитра. Нека фонът да е тъмно син. Часът, датата, бутоните и обектите от формата ще бъдат светло сини. Зададения час за аларма ще се визуализира в светло червено. За да може да покажем часа в цифров формат ще използвам по една картинка за всяка цифра от 0 до 9, както и картинка, която показва разделителя между часа, минутите и секундите. Тъй като няма стандартен шрифт за показване на цифри и текст в дигитален стил ще заредим потребителски шрифт. Чрез него ще визуализираме

датата. Скиците, които описват потребителския интерфейс на приложението са показани на Фиг. 5-2. След стартиране на приложението трябва да се визуализира екрана вляво.

Фиг. 5-2. Скициране на потребителския интерфейс

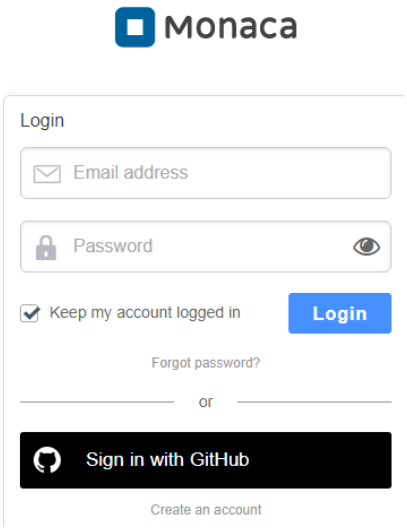
След натискане на бутон <Set Alarm> се визуализира средния екран, който съдържа форма за задаване на параметрите на алармата и за нейното активиране. При активиране на алармата се показва десния екран.

3.2 Структура на проекта

Въвеждането на програмния код, редактирането му и отстраняването на синтактични грешки ще реализираме чрез редакторите на HTML5, CSS3 и JavaScript код от Monaca Cloud IDE. Тъй като няма да използваме програмна рамка за изграждане на потребителския интерфейс ще създадем нов празен проект. Стартирайте развойната среда

https://monaca.mobi/en/dashboard

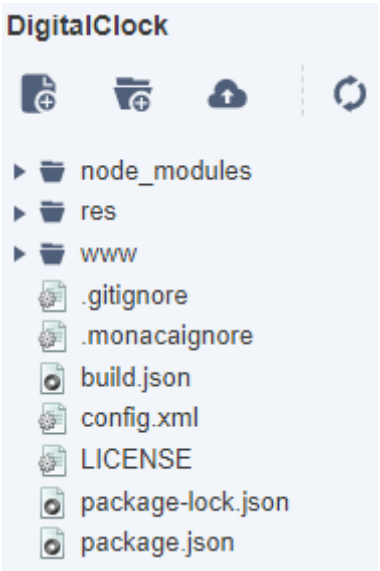
и се логнете чрез име на потребител и парола или чрез GitHub, ако имате активен профил за него (виж Фиг. 5-3).



Фиг. 5-3 Авторизация на достъпа до Monaca Cloud IDE

Ако нямате регистрация за Monaca Cloud IDE изберете “Create an account”. При регистрация се изисква въвеждане на валиден адрес на електронна поща и парола. След потвърждаване на валидността на електронната поща ще бъдете пренасочени към форма за избор на план за използване на платформата (изберете Pro). Имате право за 14 дни да тествате безплатно всеки един от наличните планове.

От Monaca dashboard натиснете бутона <Create New Project>. От Project Type изберете Blank. От Project Information задайте име на своя проект (това не е името на приложението) и кратко описание на проекта. Задайте за име DigitalClock. Натиснете бутон <Create Project>. Развойната среда ще генерира така наречения „празен проект”, който визуализира само текст на екрана. Можете и да импортирате приложението (project.zip) чрез бутон <Import Project>. Структурата на проекта е показана на Фиг. 5-4.

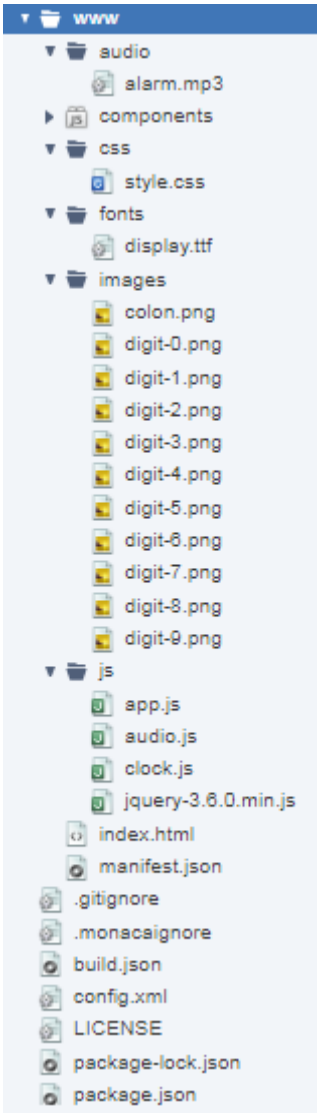


Фиг. 5-4 Начална структура на проект DigitalClock

В папка **node_modules** са всички модули за Node.js, необходими за функциониране и на приложението.

Папка **res** съдържа папки с ресурсите на приложението за всяка една от платформите за които може да изградите приложението, например Android, iOS, Winrt и Electron. В тези папки са графичните файлове с иконите (icons) и началните екрани на приложението (screen). Развойната среда ще генерира файлове по подразбиране, които впоследствие ще промените.

Папка **www** съдържа целия програмен код на приложението. В зависимост от типа на приложението, което изграждате, тази папка ще съдържа различни папки. При конкретният пример това са папки components, css и js. Папка **components** съдържа JavaScript код, необходим за зареждане на приложението. Не трябва да променяте кода от тази папка. В папка **css** ще бъдат необходимите CSS файлове (style.css). В папка **js** трябва да бъде вашия JavaScript код (app.js) и евентуално JavaScript библиотеките, които ще използвате. В папка www е индексния HTML документ - **index.htm**. Неговият код се зарежда първи след стартиране на приложението. В тази папка е и файл **config.xml**. Съдържанието му се генерира автоматично при всяка промяна на структурата на проекта. Не е желателно да правите промени директно в този файл. В папка www има още един важен файл – **package.json**. Съдържанието на този JSON документ се генерира автоматично и съдържа информация, необходима на Node.js.



Фиг. 5-5 Крайна структура на проект DigitalClock

Ще създадем следните допълнителни папки в папка www:

- Папка **images** – съдържа всички необходими графични ресурси.
- Папка **audio** – ще съдържа mp3 файл за алармата
- Папка **fonts** – ще съдържа потребителския шрифт, който ще използваме за датата.

За да създадете тези папки, маркирайте с мишката папка www и натиснете десния бутон на мишката. От менюто изберете “New Folder”. Следва вмъкване в папките на необходимите файлове. За целта ще прехвърлим всички файлове от локалния компютър. Разархивирайте проекта DigitalClock на локалния си компютър. Вмъкването на файлове във всяка от новосъздадените папки се реализира по следния начин. Маркирайте папката с мишката, натиснете десния бутон и изберете “Upload Files”. Изберете съответния файл(ове) от съответната локална папка и с Drag and Drop ги прехвърлете в папката в облака. След като „качите” всички файлове, структурата на проекта трябва да бъде като показаната структура на Фиг. 5-5.

Тъй като за функционирането на приложението е необходима библиотека jQuery свалете я от следния URL:

https://jquery.com/download/

Изберете за сваляне последната компресирана версия на библиотеката, например:

Download the compressed, production jQuery 3.6.0

Щракнете с десен бутон на мишката върху кода и изберете [Save as…].

3.3 Програмен код

Проектът се състои от един HTML файл (index.html), три JavaScript файла (app.js, audio.js и clock.js) и един файл с CSS код (style.css). Необходимите ресурси са 1 аудио файл (alarm.mp3), 11 графични файла (colon.png, digit-x.png, където x=0-9) и 1 файл съдържащ необходимия за датата шрифт (display.ttf).

```
<!DOCTYPE HTML>
<html>
<head>                                                                    1
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1,
            maximum-scale=1, user-scalable=no">
  <meta http-equiv="Content-Security-Policy" content="default-src *
            data: gap: content: https://ssl.gstatic.com; style-src * 'unsafe-inline';
            script-src * 'unsafe-inline' 'unsafe-eval'">
  <link rel="stylesheet" href="components/loader.css">
  <link rel="stylesheet" href="css/style.css">
</head>
<body>
  <div id="wrapper">                                                      2
    <div id="container">
      <div id="clock-and-date">
        <img src="images/digit-0.png" class="figure" />
        <img src="images/digit-0.png" class="figure" />
        <img src="images/colon.png" class="colon" />
        <img src="images/digit-0.png" class="figure" />
        <img src="images/digit-0.png" class="figure" />
        <img src="images/colon.png" class="colon" />
        <img src="images/digit-0.png" class="figure" />
        <img src="images/digit-0.png" class="figure" />
        <div id="date"></div>                                             2а
      </div>
      <div id="show-alarm-time"></div>
      <div id="warning"></div>
      <div id="set-alarm-container">
        <div id="set-alarm-ui">
          <label>Audio</label>
          <input type="checkbox" name="audio" value="Audio" checked="checked" />
          <label>Vibrate</label>
          <input type="checkbox" name="vibrate" value="Vibrate" checked="checked" />
          <input type="time" id="alarmtime" min="00:00" max="24:00" required />
          <button id="set">Activate</button>
        </div>
        <div id="set-alarm-button">Set Alarm</div>                        2б
      </div>
    </div>
  </div>
  <script type="text/javascript" src="components/loader.js"></script>     3
  <script type="text/javascript" src="js/jquery-3.6.0.min.js"></script>
  <script type="text/javascript" src="js/app.js"></script>
  <script type="text/javascript" src="js/clock.js"></script>
  <script type="text/javascript" src="js/audio.js"></script>
</body>
</html>
```

Фиг. 5-6 Съдържание на файл index.html

3.3.1. HTML код

Приложението е изградено чрез една HTML страница – index.html. Този файл се зарежда чрез конструктора на клас WebView на мобилната ОС веднага след стартиране на приложението. Неговата задача е да изгради интерфейса с потребителя и зареди необходимия JavaScript и CSS код. На Фиг. 5-6 е показано съдържание на файл index.html. Следва описание на програмни код по секции.

Секция 1 съдържа заглавния блок на документа. Основните етикети в тази секция са meta етикети чрез които се задава:

- Използваната от браузъра на кодова таблица (utf-8).
- Съдържание на полицата за сигурност (задава възможност за комуникации в un-safe режим).
- Как да се инициализира изгледа на приложението (viewport).
- Зареждане на всички файлове с CSS код.

Чрез елемент meta, атрибут *charset* се задава коя кодова таблица да използва браузъра при визуализиране на текст на екрана на мобилното устройство. За да можем да визуализизираме текст на кирилица сме задали кодова таблица UTF-8.

Чрез атрибут *viewport* на етикет meta се задава как да се използва екрана на устройството, как да се визуализира съдържание и с какъв мащаб:

- initial-scale – мащабиране в момента на зареждане на приложението (1 = 100%);
- user-scalable – разрешава ли се или не на потребителя да променя мащаба на информацията на екрана;
- minimum-scale, maximum-scale – минимален и максимален мащаб при промяна на мащаба на информация на екрана;
- width, height – размер (ширина и височина) на видимата област от екрана за съдържанието. Най-често width=device-width и height=device-height.
- target-densitydpi – брой пиксели на инч (dpi). Най-често target-densitydpi=device-dpi.

Секция 2 формира основния изглед на приложението (конкретното приложение има само един изглед, но на практика те може да са неограничен брой). Контейнерът със идентификатор “wrapper” ще се използва с цел абсолютно позициониране на съдържанието на екрана и ще заема 100% от неговата ширина и височина. Чрез div етикета с идентификатор “container” ще зададем отстъпите и центрирано позициониране на обектите в рамките на контейнера. Контейнерът с идентификатор “clock-and-date” (*Секция 2а*) съдържа информация за часа и датата. Цифровият час се симулира чрез изображения, които се зареждат чрез етикети img. Датата се позиционира в контейнер с идентификатор “date”.

Чрез кода в *Секция 2б* се реализира задаване, активиране или отмяна на алармата. При активирана реклама, часът и минутите на алармата се визуализират чрез контейнер “show-alarm-time”. Контейнер “warning” се използва с цел показване на съобщение при опит за неправилно задаване на аларма. Форматът за задаване на аларма се показва чрез контейнер “set-alarm-container”.

В контейнер “set-alarm-container” са разположени други два контейнера:

- Контейнер “set-alarm-ui” съдържа програмния код чрез който се визуализира форма задаване на аларма. Тази форма дава възможност за избор какво да се случи след като се активира алармата – възпроизвеждане на аудио файл и/или вибрация. По подразбиране и двете опции са активни (виж атрибут checked от етикети input с атрибут type=checkbox). Часа на активиране на алармата се задава чрез input етикет с type=time. Активирането на алармата се реализира чрез бутон <Activate>.
- Визуализирането на екрана на тази форма се реализира чрез натискане на бутон <Set Alarm>, който се симулира чрез div контейнер с идентификатор “set-alarm-button”. Този бутон се променя алтернативно до <Clear Alarm> чрез JavaScript код.

Секция 3 съдържа HTML кода чрез който се зареждат необходимите за работа на приложението JavaScript библиотеки. Първо се зарежда кода на файл loader.js, който има

за задача да зареди необходимите за функционирането на приложението библиотеки. В конкретния случай това е библиотека cordova.js. Това означава, че не е необходимо да включвате кода на тази библиотека във вашия проект – loader.js ще реализира това вместо вас. Следва зареждане на библиотека jQuery и файловете с JavaScript код на потребителя - app.js, audio.js и clock.js.

3.3.2. JavaScript код

Логиката на приложението се реализира чрез три JavaScript файла, които са разположени в папка js. Инициализацията на приложението се реализира с програмния код от файл app.js. Програмният код за възпроизвеждане на аудио е във файл audio.js. Визуализацията на часовника, задаването и отмяната на алармата се реализира от програмния код от файл clock.js.

Инициализация – app.js

Програмният код за инициализация на приложението е показан на Фиг. 5-7. Чрез обект `player` се получава достъп до програмния код за възпроизвеждане на аудио. Следва изчакване на зареждането на библиотека jQuery. Обект player се създава чрез викане на конструктора на клас MediaPlayer, част от файл audio.js. Създава се и обект `clock` чрез който се изгражда потребителския интерфейс. Активирането на базовата логика на приложението се реализира чрез метод `start`.

```
var player = null;
// Wait for jQuery library to load.
$(() => {
  // Make a player object
  player = new MediaPlayer();
  // Make a clock object
  var clock = new Clock();
  // Start a clock
  clock.start();
});
```

Фиг. 5-7 Инициализация на приложението – app.js

Възпроизвеждане на аудио – аудио.js

Възпроизвеждането на аудио от файл изисква разрешение за достъп до апаратната част на мобилното устройство – вградения говорител. За целта ще използваме плъгин. За програмна рамка Cordova този плъгин се нарича “cordova-media”. Достъпът до кода на този плъгин се реализира чрез клас Media. На Фиг. 5-8 е показан програмния код чрез който се реализира възпроизвеждане на аудио от локално разположен файл – клас MediaPlayer. Плъгинът позволява и възпроизвеждане на аудио файлове, който са хоствани на Web сайт или в хранилище на файлове.

```
class MediaPlayer {
  // Init variables
  constructor() {
    this.player = null;
    this.playStatus = 0;
  }
  // Returns player status
  getStatus() {
    return this.playStatus;
  }
  // Stop audio player
  stopAudioPlay() {
    if (this.playStatus === Media.MEDIA_STARTING ||
        this.playStatus === Media.MEDIA_RUNNING) {
      this.player.stop();
      this.player.release();
    }
  }
  // Play audio file
  playAudioFile(fimeName) {
    var url = "file:///android_asset/www/audio/" + fimeName;
    this.stopAudioPlay();
    this.player = new Media(url, this.succsess.bind(this),
          this.error.bind(this), this.statusChanged.bind(this));
    this.player.play();
  }
  // Seccess callback function
  succsess() {
  }
  // Error callback function
  error() {
    this.stopAudioPlay();
  }
  // Status changed callback function
  statusChanged(value) {
    this.playStatus = value;
    // If audio play stopped, then remove event 'click' (document)
    if (value === Media.MEDIA_STOPPED) {
      $(document).off('click');
    }
  }
}
```

Фиг. 5-8 Възпроизвеждане на аудио – audio.js

От конструктора на клас MediaPlayer декларираме обект player и променлива playStatus чрез която се разбира статуса на възпроизвеждане (няма възпроизвеждане, начало на възпроизвеждане, възпроизвежда се в момента, край на възпроизвеждане). Тези състояния се кодират чрез съответните константи от клас Media. Методите, които MediaPlayer предоставя са следните:

- playAudioFile(file) – стартира възпроизвеждане на файл с име file.
- stopAudioPlay() – спира принудително възпроизвеждането.
- getStatus() – връща текущото състояние на обекта player.Създаването на обект player се реализира чрез конструктора на клас Media.

Конструкторът изисква задаване на три атрибута:

- Път (локален или глобален) до аудио файла – url. В конкретния случай е зададен локалния път за устройства с ОС Android (android_asset/www).
- Функция (callback), която се вика при успешно създаване на player – success.
- Функция (callback), която се вика при неуспешно създаване на player – error.
- Функция (callback), която се вика при промяна на състоянието на player – statusChanged.

Да припомним, че всички променливи и обекти, част от JavaScript обект трябва да се достъпват чрез ключова дума `this` (референцията на текущия обект). Тъй като стойността на this за кода от тялото на клас MediaPlayer и стойността на `this` от тялото на callback функциите имат различни стойности, се налага да приложим метод `bind` към всяка callback функция. По този начин callback функциите получават коректната стойност за `this`.

Спирането на възпроизвеждането се реализира чрез метод stopAudioPlayer. Проверява се дали в момента се възпроизвежда аудио и ако това е вярно се спира възпроизвеждането (метод stop) и се освобождават заеманите ресурси (метод release).

По условие на задачата трябва да се преустанови възпроизвеждането, ако потребителят докосне екрана. За целта, преди да започне възпроизвеждането, прехващаме събитие “click” (файл clock.js). Ако потребителя не докосне екрана ще трябва да деактивираме подслушването на това събитие в момента когато възпроизвеждането завърши. Това се реализира чрез метод `off` (виж метод statusChanged).

Потребителски интерфейс – clock.js

Файл clock.js съдържа клас Clock чрез който се реализира логиката на приложението и потребителския интерфейс (виж Фиг. 5-9). От тялото на конструктора декларираме и инициализираме всички променливи и обекти, принадлежащи на класа. Например, стойността на променливата alarmStatus определя дали алармата е зададена (true) или не е зададена (false). Кой аудио файл трябва да се възпроизведе при активиране на алармата се задава чрез променлива audioFile.

```
class Clock {
  // Init vars and objects
  constructor() {
    this.alarmStatus = false;
    this.alarmItems = [];
    this.alarmTime = '';
    this.path = 'images/digit-';
    this.months = ['January', 'February', 'March', 'April', 'May', 'June',
                   'July', 'August', 'September', 'October', 'November', 'December'];
    this.alarmUi = '#set-alarm-ui';
    this.alarmButton = '#set-alarm-button';
    this.audioFile = 'alarm.mp3';
  }
  // Init and start the clock to be displayed every second.
  start() {
    this.init();
    this.showClock();
    this.timer = setInterval(this.showClock.bind(this), 1000);
  }
  // Stop a clock
  stop() {
    clearInterval(this.timer);
  }
  // set alarm time in format hh:mm
  setAlarmTime(hours, mins) {
    this.alarmTime.setHours(hours, mins, 0);
  }
  // Init clock ...
  init() {

  }
  // Show digital clock
  showClock() {

  }
}
```

Фиг. 5-9 Потребителски интерфейс – clock.js

Метод `start` активира логиката на приложението. Последователно се викат методи init(), showClock() и setInterval(…). Чрез метод setInterval се вика callback функция showClock през интервал от 1 секунда. Тази функция визуализира текущия час.

```
  init() {
    $('#warning').hide();
    $(this.alarmUi).hide();
    $(this.alarmButton).show();
    var self = this;
    // <Set alarm> button callback function
    $(this.alarmButton).on('click', () => {
      let text = $(self.alarmButton).text();
      if (text == 'Set Alarm') {
        $(self.alarmButton).hide();
        let now = new Date();
        let hours = now.getHours().toString().padStart(2, '0');
        let minutes = now.getMinutes().toString().padStart(2, '0');
        let time = `${hours}:${minutes}`;
        $('#alarmtime').val(time);
        $(self.alarmUi).show();
      }
      else if (text === 'Clear Alarm') {
        self.alarmStatus = false;
        self.alarmtime = '';
        $('#show-alarm-time').hide();
        $(self.alarmButton).text('Set Alarm');
      }
    });
    // <Activate> alarm button callback function
    $('#set').on('click', () => {
      $(self.alarmUi).hide();
      $(self.alarmButton).show();
      var nSelected = 0;
      var items = ['audio', 'vibrate'];
      self.alarmItems = [];

      for (var item of items) {
        var itemObject = $(`[name="${item}"]:checked`);
        var value = itemObject.length;
        nSelected += value;
        self.alarmItems.push(value);
      }
      self.alarmtime = $('#alarmtime').val();

      // Displays a warning message indicating that information required to start the
      // alarm is missing.
      if (nSelected === 0 || self.alarmtime == '') {
        $('#warning').html("You must select the alarm type and alarm activation time!");
        $('#warning').show();
        setTimeout(() => {
          $('#warning').fadeOut();
        }, 3000);
      }
      else {
        // Activate alarm mode
        $('#show-alarm-time').text(`Alarm time: ${self.alarmtime}`);
        $('#show-alarm-time').show();
        $(self.alarmButton).text('Clear Alarm');

        let timeArray = self.alarmtime.split(':');
        let hours = timeArray[0];
        let mins = timeArray[1];

        self.alarmTime = new Date();
        self.alarmTime.setHours(hours, mins, 0);
        self.alarmStatus = true;
      }
    });
  }
```

Фиг. 5-10 Метод init от файл clock.js

Програмният код на метод init() е показан на Фиг. 5-10. Започваме със показване само на бутон <Set Alarm> и скриване на останалите компонентите на потребителския интерфейс. За целта използваме методи show() и hide() от jQuery. След това се активира следене дали не се натиска този бутон. Това се реализира чрез jQuery метод `on` със събитие “click”, приложен към DOM обекта, описващ бутона. Този бутон се използва алтернативно. Първоначално е <Set Alarm>. След активиране на алармата, бутонът се променя до <Clear Alarm>. След изчистване на алармата отново се преминава към <Set Alarm>. За да разпознаем за какво служи бутона в даден момент прочитаме текстовото му описание чрез метод text(). Ако текстът е “Set Alarm” получаваме текущата дата и час чрез конструктора на клас Date. Извличаме часа и минутите във формат “hh:mm” – променлива time. Инициализираме с тази стойност времето за аларма - input елемента с type=time. По този начин полето е инициализирано, а не остава празно. Потребителят решава дали да промени времето за аларма или до го остави текущото. Следва визуализиране на формата за задаване на аларма. Ако текстът е “Clear Alarm” скриваме формата за задаване на аларма, задаваме стойност false за alarmStatus и показваме бутон <Set Alarm>.

Следва прехващане на събитието генерирано при натискане на бутон <Activate> за активиране на алармата. Това се реализира чрез метод `on`, събитие “click”. Методът се прилага към бутон <Activate>, който има идентификатор set. При активиране на callback функцията се скрива формата за задаване на аларма и показва бутон <Set Alarm>. Следва прочитане на състоянието на чек боксове audio и vibrate и прочитане на часа за активиране на алармата. Ако потребителят не е избрал време за аларма или не е маркирал audio и/или видео се показва съобщение за това в контейнер с идентификатор warning. Това съобщение се вижда на екрана за 3 секунди след което плавно се скрива. За да се постигне това се използва метод fadeOut().

Ако потребителят е задал всичко коректно се преминава към активиране на алармата. Първо се сменя името на бутона от <Set Alarm> на <Clear Alarm>. Извличат се часа и минутите за алармата (получени от формата за задаване на аларма). Това се получава чрез прочитане на стойността на променлива alarmtime. Следва получаване на времето за аларма във формат Date, за да може да бъде сравнявано с текущото време. Това се реализира чрез метод setHours().

На Фиг. 5-11 е показан програмния код на метод showClock. Този метод се вика през интервал от 1 секунда. Започва се с получаване на текущата дата и час. Ако алармата е зададена (променлива alarmStatus=true) се сравнява дали часа и минутите от алармата съвпадат с текущия час и минути. Ако това е вярно се скрива времето за аларма и показва бутон <Set Alarm>. Променлива alarmStatus приема стойност false. Ако е избрано да се възпроизвежда аудио файл (alarmItem[0]=1) се задейства прехващане на събитието „докоснат е екрана”. За целта е използван метода `one`, а не `on`. При метод one() ако събитието се генерира, прехващането се деактивира автоматично. Правим това прехващане, за да дадем възможност на потребителя да прекъсне възпроизвеждането чрез докосване на екрана. В противен случай възпроизвеждането ще спре след достигане до края на аудио файла. Следва активиране на възпроизвеждането на аудио файла (playAudioFile). Ако е зададена и вибрация (alarmItem[1]=1) се активира вибрация за интервал от 500ms. За целта се използва метод vibrate() от обект navigator. За да може да получите достъп до вибратора е необходимо да използвате плъгина “cordova-vibrate”.

```
  showClock() {
    var figures = $('.figure');
    var date = $('#date');

    var now = new Date();
    var day = now.getDate();
    var month = this.months[now.getMonth()];
    var year = now.getFullYear();
    var seconds = now.getSeconds();
    var minutes = now.getMinutes();
    var hours = now.getHours();

    if (this.alarmStatus === true) {
      let alarmHours = this.alarmTime.getHours();
      let alarmMinutes = this.alarmTime.getMinutes();
      if (alarmHours == hours && alarmMinutes == minutes) {
        $('#show-alarm-time').hide();
        $(this.alarmButton).text('Set Alarm');
        this.alarmStatus = false;
        // Play audio file ...
        if (this.alarmItems[0] == 1) {
          $(document).one('click', () => {
            let playerStatus = player.getStatus();
            if (playerStatus === Media.MEDIA_RUNNING) {
              player.stopAudioPlay();
            }
          });
          player.playAudioFile(this.audioFile);
        }
        // Vibrate ...
        if (this.alarmItems[1] == 1) {
          navigator.vibrate(500);
        }
      }
    }
    figures[0].src = this.path + this.getTens(hours) + '.png';
    figures[1].src = this.path + this.getOnes(hours) + '.png';
    figures[2].src = this.path + this.getTens(minutes) + '.png';
    figures[3].src = this.path + this.getOnes(minutes) + '.png';
    figures[4].src = this.path + this.getTens(seconds) + '.png';
    figures[5].src = this.path + this.getOnes(seconds) + '.png';
    date.html(`${day} ${month} ${year}`);
  }
```

Фиг. 5-11 Метод showClock от файл clock.js

И накрая, остава да обновим датата и часа. Това се реализира чрез промяна на атрибут src на всяка едно от изображенията, които описват часа, минутите и секундите. Обектът, чрез който ще реализираме достъп до тези изображения е figures. Той се получава в началото на метод showClock. Пътят до изображенията се описва с променлива path. Нейната стойност е “images/digit-“.

Остава да долепим до този път цифрата, която отговаря за часа, минутите или секундите. За целта за часовете, минутите и секундите трябва да получим десетиците и единиците. Това се реализира чрез два помощни метода – getTens и getOnes (виж Фиг. 5-12).

```
getOnes(i) {
    return ("" + i).slice(-1);
}
getTens(i) {
    var str = "0" + i;
    return str.slice(-2, str.length - 1);
}
```

Фиг. 5-12 Методи getOnes и getTens от файл clock.js

3.3.3. CSS код

Стиловото форматиране се реализира чрез CSS правила от файл style.css (виж Фиг. 5-13). Зареждането на шрифта display.ttf трябва да се реализира в *началото* на файла чрез правило @font-face. За всички символи, без полето за дата, се използва шрифт system-ui. За да фиксираме позицията на бутоните и формата за задаване на аларма в края на екрана, за елементи с идентификатори “set-alarm-ui” и “set-alarm-button” задаваме декларации position: absolute; и bottom: 15px;.

```
@font-face {
   font-family: display;
   src: url(../fonts/display.ttf);
}

body {
   font-family: system-ui;
   background-color: #000015;
   color: rgb(0, 119, 199);
   padding: 0;
   margin: 0;
   height: 100%;
}
#wrapper {
   position: absolute;
   left: 0px;
   right: 0px;
   width: 100%;
   height: 100%;
}
#container {
   width: 90%;
   margin: auto;
   padding-top : 30px;
   display : block;
   text-align: center;
}
#container img {
   display: block;
   float: left;
   width: 15%;
}
#container img.colon{
   display: block;
   width: 5%;
}
#clock-and-date {
   border: 1px solid rgb(0, 103, 172);
   border-radius: 4px;
   padding-top: 20px;
   padding-left: 10px;
   padding-right: 10px;
}
#show-alarm-time {
   margin: 10px 0;
   padding: 0;
   color: rgba(206, 0, 0, 0.932);
   font-size: 5vmin;
   text-align: right;
   border: none;
#date {
   font-family: display;
   font-size: 6vmin;
   padding-bottom: 10px;
   padding-right: 10px;
   text-align: right;
}
#set-alarm-ui, #set-alarm-button {
   border: 1px solid rgb(0, 103, 172);
   padding-top: 10px;
   padding-bottom: 10px;
   border-radius: 4px;
   position: absolute;
   width: 90%;
   bottom: 15px;
   font-size: 5vmin;
}
#set, #clear {
   display: in-line-block;
   margin: 10px;
   padding: 5px 10px;
   border: 1px solid rgb(0, 103, 172);
   border-radius: 4px;
   background-color: rgba(0,0,0,0);
   color: rgb(0, 103, 172);
   font-size: 4.5vmin;
}
#alarmtime {
   margin: 0 5px;
   padding: 0 5px;
   border: none;
   border-radius: 3px;
   background-color: rgb(0, 103, 172);
   color: #000030;
   font-size: 5vmin;
}
#warning {
   margin: 20px 0;
   padding: 10px 0;
   color: rgba(255, 0, 0, 0.932);
   font-size: 5.5vmin;
   background-color: rgba(0,100,100,0.3);
   border: none;
   border-radius: 6px;
}
```

Фиг. 5-13 Стилово форматиране – style.css

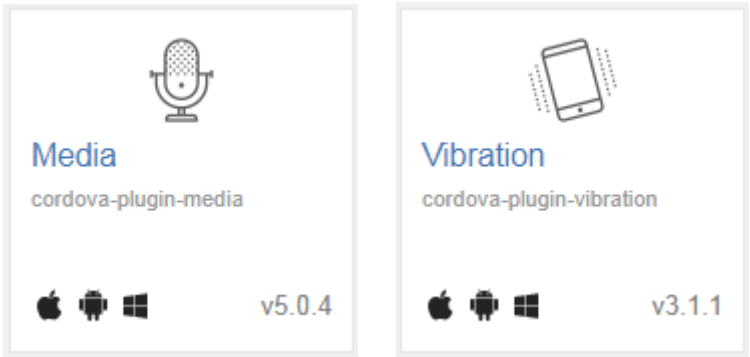
3.4 Тестване на приложението

Приложението може да бъде тествано по три начина:

- Използване на вградения в Monaca Cloud IDE симулатор. Екранът му е прикачен в десния край на редактора. Можете да избирате симулиране на различни мобилни устройства, както и режим portrait и landscape. При конкретният пример не може да се симулира работата на вибратора и възпроизвеждането на аудио.
- Използване на Monaca Debugger. В този случай няма да можете да тествате възпроизвеждането на аудио.
- Използване на реално устройство. В този случай трябва да изградите приложеното, за да получите инсталационен файл (APK за Android).

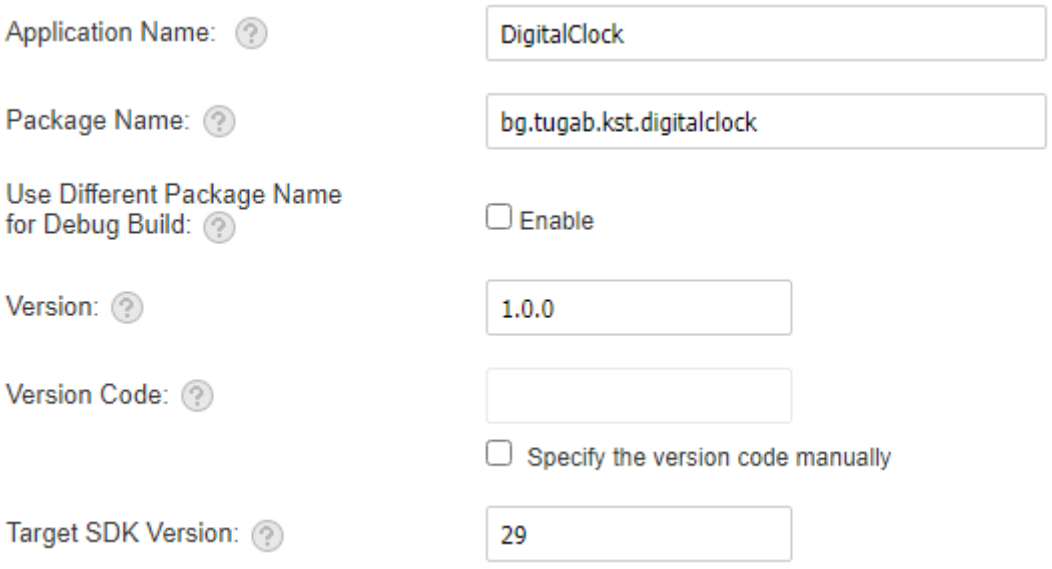
3.5 Конфигуриране на приложението

За да изградите приложението трябва да преминете през избор на необходимите плъгини и задаване на информация за приложението. От менюто на Monaca Cloud IDE изберете Configure, а след това “Cordova Plugin Settings…”. От “Available Plugins” изберете плъгините Media и Vibration (Фиг. 5-14).



Фиг. 5-14 Необходими плъгини

От менюто изберете Configure, а след това “App Settings for Android…”. Трябва да зададете редица конфигурационни параметри, както и икона и начален екран на приложението. Нека името на приложението да е DigitalClock, а името на пакета – bg.tugab.bg.digitalclock. Задайте версия 1.0.0, а за “Target SDK Version” – 29 (виж Фиг. 5-15).

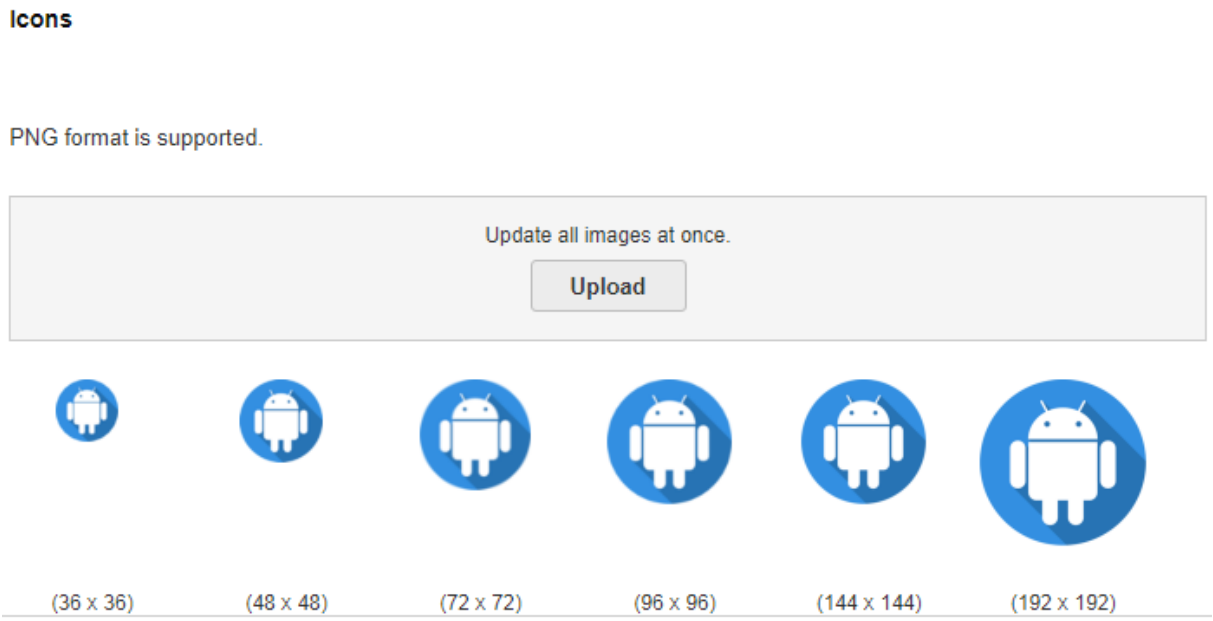


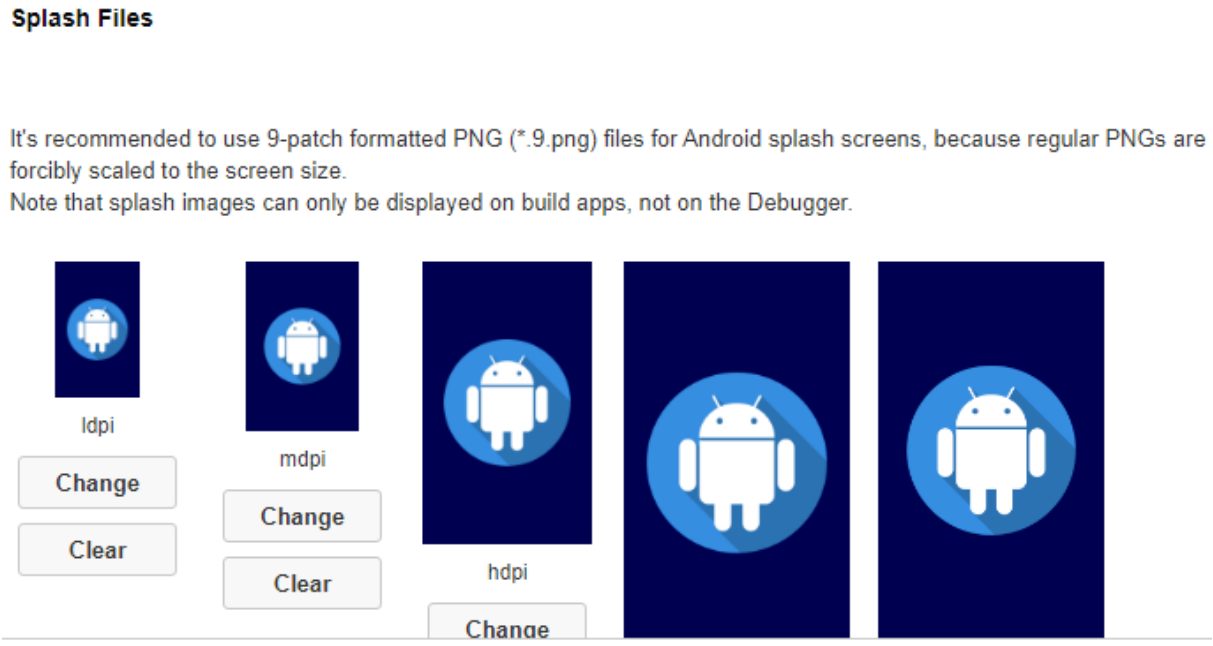
Фиг. 5-15 Конфигуриране на приложението

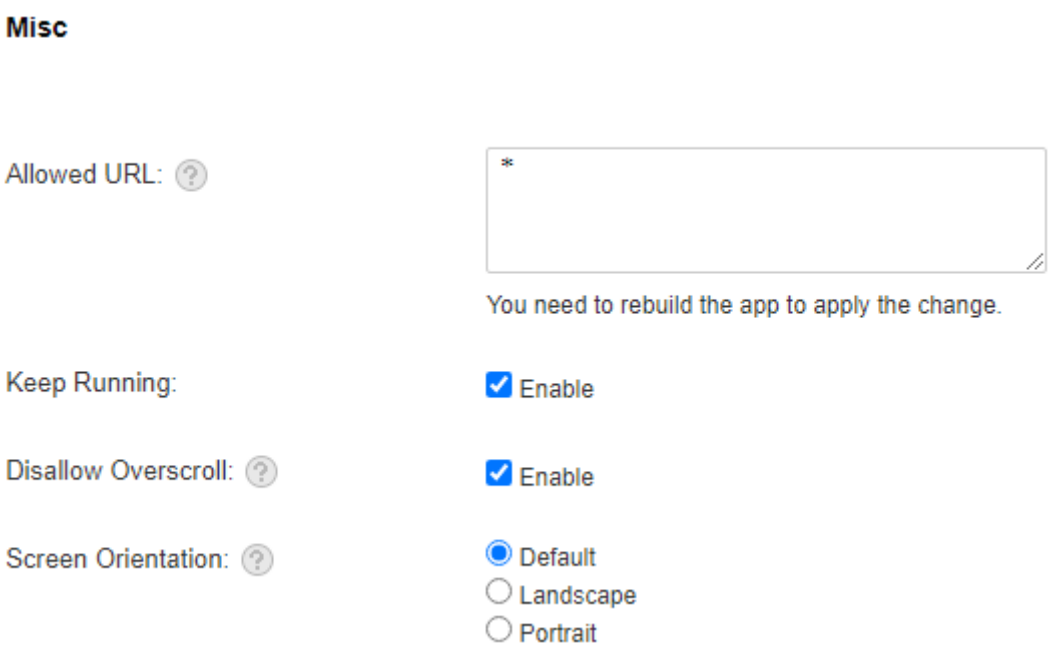
Трябва да генерирате графични файлове в PNG формат за началния екран (splash screen) и иконата на приложението. За да получите тези файлове максимално бързо използвайте online ресурс

https://pgicons.abiro.com/

Наименованията на иконите и картинките е в зависимост от операционната система и размера на екрана на устройството на което приложението ще се стартира. Задайте приложението да продължи да работи при загуба на фокус (Keep Running). Забранете функция overscroll. За да запазите въведената информация натиснете бутон <Save>.



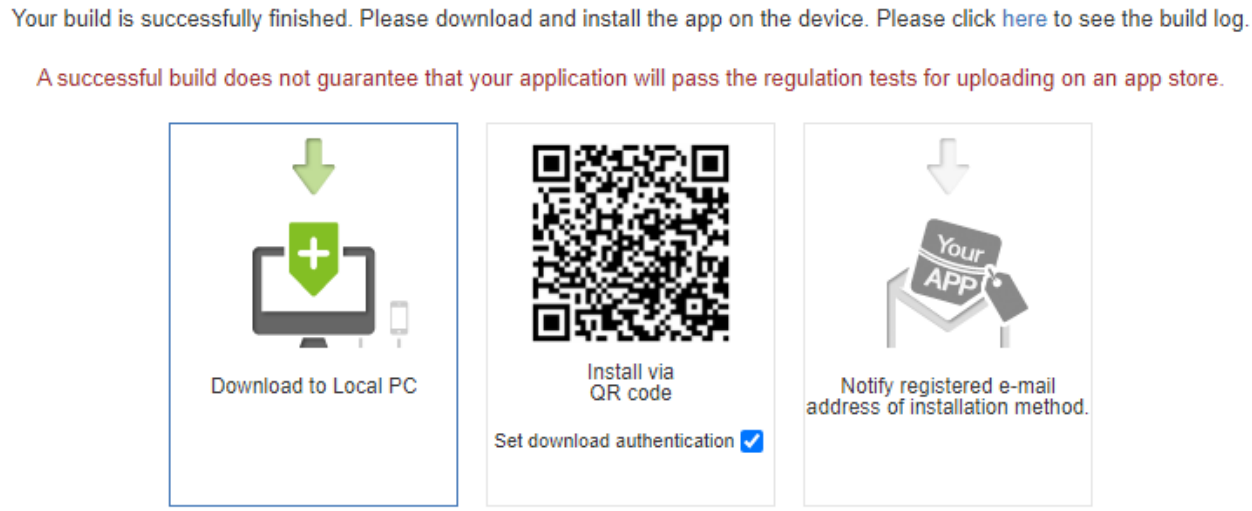




Фиг. 5-16 Завършване на конфигурирането на приложението

3.6 Изграждане на приложението

За да получите инсталационен файл за ОС Android изберете от менюто Build, а след това “Build App for Android…”. След това изберете “Build for Debugging” и натиснете бутон <Start Build>. След успешно изграждане ще получите възможност да инсталирате APK файла като го свалите на локалния диск, като сканирате генерирания QR код или да го изпратите на пощата си (виж Фиг. 5-17).



Фиг. 5-17 Получаване на инсталационния пакет (APK файл)

След инсталиране на приложението може да го тествате. На Фиг. 5-18 са показани екрани, получени при тестване на приложението на реално устройство (Android ОС).

Фиг. 5-18 Тестване на приложението на реално устройство

Заключение

В тази глава се научихте как да създадете мобилно хибридно приложение, без да използвате рамка за изграждане на потребителския интерфейс. Това предполага писане на повече програмен код, отколкото ако използвате програмна рамка, но целта на този проект е да придобиете опит с използването на HTML5, CSS3 и JavaScript. Научихте се каква е структурата на един Monaca проект, как да конфигурирате проекта и как да го изградите, за да получите инсталируем пакет за ОС Android.

Глава 6

Програмна рамка Framework7

I. Въведение

Framework7 е безплатна и с отворен код програма рамка за разработване на хибридни мобилни приложения за iOS, Android и Windows. Началото на анонсиране на Framework7 е 2014 г. Първоначално идеята е била да се предостави Web (HTML5 и JavaScript) базирана програмна рамка за разработване на мобилни приложения с нативен интерфейс за iOS 7 устройства (затова името на рамката е Framewor7). В последствие се добавя поддръжка и на устройства с операционна система Android, Windows и Web.

Основните предимства на Framework7 са следните:

- Отворен код, за безплатно използване.
- Сравнително малък размер на програмния код, добро бързодействие.
- Лесен за изучаване. Ако имате опит с програмна рамка jQuery лесно ще се адаптирате към синтаксиса на Framework7.
- Независима от други библиотеки и програмни рамки: работа със собствени шаблони (Template7) и със собствен DOM наречен DOM7.
- Възможност за динамично зареждане на страници.
- Възможна интеграция с програмни рамки React и Vue.
- Компонентен модел – наличие на множество графични компоненти, които позволяват потребителският интерфейс да изглежда така, както при нативните приложения за съответната ОС.

Основните недостатъци на Framework7 са следните:

- Не достатъчно подробна докумнетация.
- Ограничена поддръжка.

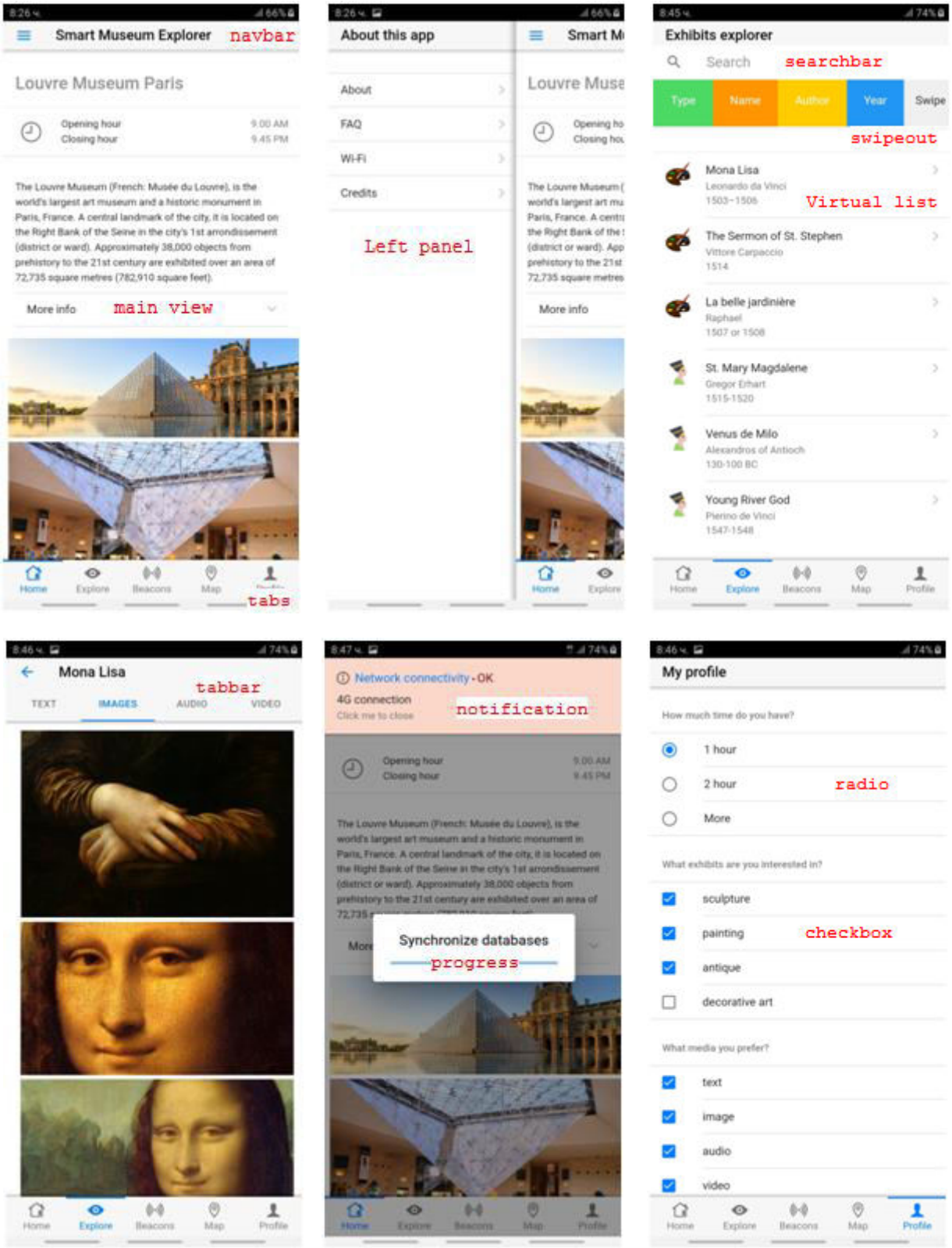
II. Основни Framework7 компоненти

Чрез Framework7 се създават хибридни мобилни приложения с нативен потребителски интерфейс. От архитектурна гледна точка Framework7 създава приложения с една HTML страница - Single Page Apps (SPA). Най-често използваните програмни рамки за реализиране на интерфейса с потребителя като React, Vue и Angular също използват SPA дизайн. Основната страница на подобни приложения по подразбиране е файл index.html. За да се визуализира ново съдържание се интегрира динамично HTML код в тялото на индексния файл. Новото съдържание можа да се получи от HTML файл, но може да се генерира и програмно.

SPA приложенията позволяват:

- Оптимизиране на трафика между клиентите и сървъра.
- По-бързо превключване на съдържанието.
- Намалява се размера на кода, тъй като допълнителните документи не трябва да съдържат заглавен блок.

Програмна рамка Framework7 предоставя над 65 графични компоненти чрез които може да изградите потребителския интерфейс. Може да използвате графичните компоненти на рамки React или Vue.



Фиг. 6-1 Основни компоненти от изгледа на едно Framework7 приложение

На Фиг. 6-1 са показани основните компоненти, които изграждат потребителския интерес на примерно Framework7 приложение. Изгледът на приложението се състои от следните основни компоненти:

- Навигационната лента (navbar) – може да съдържа заглавие (navbar title), ляво и дясно меню. При конкретният пример се използва само ляво меню.
- Съдържание (page content) на основния изглед на страницата (main view).
- Лента с инструменти (toolbar) – при конкретния пример приложението използва пет табулатора (tabs). Табулаторите позволяват бързо превключване на контекста, без да се налага да се търси съответния елемент от менюто на приложението.
- Панели. Конкретният пример работи с ляво позициониран панел (left panel). Най-често панелът се използва за реализация на менюто на приложението. При избор на елемент от менюто се визуализира съдържание, което може да бъде позиционирано както в панела, така и в самата страница (page content). Няма ограничение за елементите, които изграждат това съдържание. Панелът може да припокрива съдържанието на страницата или да не припокрива това съдържание (panel-reveal), както е показано на Фиг. 6-1.

Други компоненти, които често се използват при изграждане на потребителския интерфейс, са следните:

- Нотификации – прозорци със графично и текстове съдържание, които се визуализират основно при настъпване на някакво събитие. При конкретния пример събитието е „Възобновяване на мрежовата свързаност”.
- Информация за прогреса на дадена операция. При конкретния пример – напредъка по синхронизиране на съдържанието на локалната база данни с отдалечената база данни.
- Компоненти, които се използват за изграждане на форми за въвеждане на информация: радио бутони, отметки (checkbox), падащо меню (select-option) и др.
- Компоненти-списъци. При конкретния пример се използва компонент Virtual List с цел визуализиране на експонатите в музей.
- Лента за търсене на съдържание (search bar). Използва се с цел филтриране на съдържанието в Virtual List.
- Компоненти, които разкриват своята функционалност след изтегляне с пръст наляво и/или надясно – swipeout. При конкретния пример по този начин се достига до елементите на филтъра за Virtual List.

III. Структура на Framework7 проектите

Основните програмни модули, които изграждат всяко Framework7 приложение, са следните:

- **index.html** - основна HTML страници, която изгражда основния изглед на приложението (main view).
- **app.js** - файл, съдържащ JavaScript код за инициализация на приложението. В този файл се създават обекти за достъп до DOM7 и самата рамка Framework7. Ако използвате плъгини за Framework7, тук е мястото да ги импортирате. В този файл може да реализирате изграждането на обектите чрез които ще управлявате изгледите на вашето приложение. В този файл може да включите и прехващането на всички събития, които са необходими за за функционирането на приложението.
- **routes.js** - файл, описващ маршрутите за вашето приложение (routes). Маршрутите описват връзката между името на ресурс и реалното местоположение

на документа в проектната папка. Те най-често се описват като масив от JSON обекти, всеки от които описва един маршрут.

- **app.css** – код за стилово форматиране, което допълва стиловото форматиране, което се задава чрез Framework7. Можете да зададете специфично стилово форматиране на определени компоненти от потребителския интефейс.

Съдържанието на примерна началната страница (index.html) е показано на Фиг. 6-2. Ще разгледаме този програмен код по секции:

Секция 1. Съдържа HTML кода от *заглавния блок* на страницата. Чрез атрибут charset се задава кодовата таблица с която браузъра трябва да работи (utf-8). Трябва задължително да се зададе чрез мета етикет и viewport, тъй като той е необходим за създаване на responsive дизайн на мобилното приложение. Тук можете да укажете дали съдържанието от страницата да може да бъде мащабирано или да е фиксирано като мащаб и какъв да бъде този мащаб като минимална и максимална стойност. Задава се и основната цветова схема с която искате да работите theme-color. Може да разрешите или да забраните позвънявания към телефонни номера, които са част от текста, показан в изгледа на приложението – format-detection, telephone=yes/no. В тази секция са зарежда и необходимия CSS код чрез който се стилизира интерфейса с потребителите. Първо се зарежда CSS кода от Framework7, който реализира нативния изглед на приложението. Следва зареждане на стиловото форматиране за иконите. Последен трябва да заредите CSS кода, който съдържа специфично за приложението CSS форматиране (app.css).

Секция 2. Тази секция описва левият панел на приложението. Съдържанието на панела се визуализира при натискане на иконата за менюто на приложението. Тази икона има име menu и е стандартния тип „хамбургер” икона. Зададено е панелът да не припокрива съдържанието на страницата (panel-reveal). Това се използва когато екранът на устройството е достатъчно голям. В контейнера от клас page-content се вмъква HTML кода, който визуализира заглавието на приложението и описва желаните елементи от менюто на приложението (в случая About и Help).

Секция 3. Тази секция описва горната навигационна лента (navbar). В тази лента най-често са иконата за активиране на менюто на приложението и неговото заглавие. Иконите се задават чрез етикет `i` при стойност icon на атрибут class. Ако приложението трябва да компилира за няколко операционни системи, трябва да опишете иконата за всяка от тях.

Секция 4. Тази секция описва лентата с инструменти (toolbar). При конкретният пример в лентата с инструменти има описан един табулатор (Home). Чрез табулаторите се реализира бързо превключване между съдържанието на основните страници, които изграждат приложението. Използвани са икони за допълнително графично описание на табулаторите. Framework7 поддържа собствени икони в SVG формат (векторна графика). Може да получите информация за поддържаните икони от следния адрес:

https://v4.framework7.io/icons/

Секция 5. Програмният код от тази секция визуализира съдържанието на основната страница (page-content). Това съдържание най-често се поставя в един или повече div контейнери. Може да използвате всички налични в програмната рамка графични компоненти, за да създадете своя потребителски интерфейс.

```
<!DOCTYPE html>
<html>
  <head>
```

1

```
    <meta charset="utf-8">
    <meta http-equiv="Content-Security-Policy" content="default-src *
     data: gap: content: https://ssl.gstatic.com; style-src * 'unsafe-inline';
     script-src * 'unsafe-inline' 'unsafe-eval'">
    <meta name="viewport" content="width=device-width, initial-scale=1,
     maximum-scale=1, minimum-scale=1, user-scalable=no, minimal-ui,
     viewport-fit=cover">
    <meta name="apple-mobile-web-app-capable" content="yes">
    <meta name="apple-mobile-web-app-status-bar-style" content="default">
    <meta name="theme-color" content="#2196f3">
    <meta name="format-detection" content="telephone=no">
    <meta name="msapplication-tap-highlight" content="no">

    <link rel="stylesheet" href="framework7/css/framework7.min.css">
    <link rel="stylesheet" href="css/icons.css">
    <link rel="stylesheet" href="components/loader.css">
    <link rel="stylesheet" href="css/app.css">
```

```
  </head>
  <body>
    <div id="app">
      <!-- Status bar overlay for fullscreen mode-->
      <div class="statusbar"></div>
```

2

```
      <!-- Left panel with cover effect-->
      <div class="panel panel-left panel-reveal">
        <div class="view view-left">
          <div class="page">
            <div class="navbar">
              <div class="navbar-inner sliding">
                <div class="title">Меню</div>
              </div>
            </div>
            <div class="page-content">
              <div class="block-title">Заглавие на секция в менюто</div>
              <div class="list links-list">
                <ul>
                  <li><a href="/about/">About</a></li>
                  <li><a href="/help/">Help</a></li>
                </ul>
              </div>
            </div>
          </div>
        </div>
      </div>

      <!-- Right panel -->

      <!-- Your main view, should have "view-main" class -->
      <div class="view view-main safe-areas">
      <!-- Page, data-name contains page name which can be used in callbacks -->
      <div class="page" data-name="home">
```

3

```
        <!-- Top Navbar -->
        <div class="navbar">
          <div class="navbar-inner">
            <div class="left">
              <a href="#" class="link icon-only panel-open" data-panel="left">
                <i class="icon f7-icons ios-only">menu</i>
                <i class="icon material-icons md-only">menu</i>
              </a>
            </div>
            <div class="title sliding">Заглавие на приложението</div>
          </div>
        </div>
```

Фиг. 6-2 Съдържание на файл index.html (1 от 2)

```
    <!-- Toolbar-->                                                    4
    <div class="toolbar tabbar-labels toolbar-bottom view-link"
                                        id="main-toolbar">
      <div class="toolbar-inner">
        <a href="#view-home" class="tab-link tab-link-active">
          <i class="icon f7-icons ios-only">home</i>
          <i class="icon f7-icons ios-only icon-ios-fill">home</i>
          <i class="icon f7-icons md-only">home</i>
          <span class="tabbar-label">Home</span>
        </a>
        ...
      </div>
    </div>

    <!-- Scrollable page content-->                                    5
    <div class="page-content">
      <div class="block">
        …
      </div>
    </div>

  </div>   // <div class="page" data-name="home">
 </div>    // <div class="view view-main safe-areas">
</div>     // <<div id="app">

    <!-- Framework7 library -->                                        6
    <script src="framework7/js/framework7.min.js"></script>
    <!-- Monaca JS library -->
    <script src="components/loader.js"></script>
    <!-- App routes -->
    <script src="js/routes.js"></script>
    <!-- Your custom app scripts -->
    <script src="js/app.js"></script>

  </body>
</html>
```

Фиг. 6-2 Съдържание на файл index.html (2 от 2)

Секция 6. В тази секция, чрез етикет script, се зареждат всички JavaScript библиотеки, необходими за функциониране на приложението. Първо трябва да укажете пътя до кода на рамката Framework7. След това - кода на модул loader.js. Следва зареждане на файл routes.js и накрая се зарежда вашия JavaScript код – app.js. Разбира се, ако приложението е с по-сложна структура може да имате множество JavaScript файлове с потребителски код.

IV.Състояние на страниците

Всяка страница преминава през множество състояния по времето на своя жизнен цикъл:

- **page:mounted** - Събитието ще се задейства, когато нова страница бъде вмъкната в DOM или когато страница с маршрут keepAlive се монтира към DOM.
- **page:init** - Събитието ще се задейства, след като Framework7 инициализира необходимите компоненти на страницата и навигационната лента.
- **page:reinit** - Това събитие ще бъде задействано в случай на преминаване към страница, която вече е била инициализирана.
- **page:beforein** - Събитието ще бъде задействано, когато всичко е инициализирано и страницата е готова да бъде прехвърлена в изглед.
- **page:afterin** - Събитието ще бъде задействано, след като страницата премине в режим на показване.

- **page:beforeout** - Събитието ще се задейства точно преди страницата да бъде изведена от изглед.
- **page:afterout** - Събитието ще се задейства, след като страницата излезе от изглед.
- **page:beforeunmount** - Събитието ще бъде задействано, когато страницата с маршрут keepAlive ще бъде демонтирана от DOM.
- **page: beforeremove** - Събитието ще се задейства точно преди страницата да бъде премахната от DOM. Това събитие може да бъде много полезно, ако трябва да отделите някои събития или да унищожите някои плъгини, за да освободите памет. Това събитие няма да бъде задействано за keepAlive маршрути.
- **page:tabshow** - Събитието ще бъде задействано, когато родителското меню на страница-табулатор стане видимо.
- **page:tabhide** - Събитието ще се задейства, когато родителският изглед на страница-табулатор стане скрит.

V. Маршрутизация

Всяка програмна рамка за изграждане на хибридни мобилни приложения трябва да позволява навигация между изгледите на различните компоненти, които изграждат потребителския интерфейс. Дефинирането на маршрути позволява бързо да се намери пътя (URL) до даден ресурс по зададено име. Ако за името на ресурс няма запис, приложението се насочва да покаже страница за липсващ ресурс (Page not found – код 404).

Както вече знаем маршрутите се описват като масив от JSON обекти и се задават като стойност на свойство routes при инициализация на Framework7 (виж Фиг. 6-3). Основните свойства, които може да зададете за всеки път, са следните:

- **name** - име на пътя, например home.
- **path** – псевдоним на ресурс, който е препратка (link) или изглед (view).
- **url** – статичен или динамичен ресурс, който съответства на указан път (path).
- **templateUrl** - статичен или динамичен ресурс-шаблон, който съответства на указан път.
- **async** – асинхронна функция, която се активира с цел динамично маршрутизиране. Задава се вместо свойство url.

В **Секция 1** се задава кой ресурс да се зареди при „сляпа” GET заявка (path: ‘/’) и при препратка с име ‘/settings/’. При „сляпа” заявка се зарежда index.html, а при заявка ‘/setting/’ – файл settings.html, който се намира в папка pages, която е в root папката на проекта.

В **Секция 2** е показано как се зарежда документ, който съдържа шаблон. Шаблоните се използват с цел динамично изграждане на HTML страници. Това се реализира чрез специален синтаксис, който позволява вграждане в статичната част от страницата на динамично генериран HTML код. Този динамично генериран код може да се получи или чрез вмъкване на стойностите на променливи или чрез JavaScript код. Вместо свойство url трябва да се използва свойство templateUrl. Програмна рамка Framework7 използва двигателя Template7 (до версия 5.x), чийто синтаксис е подобен на синтаксиса на Handle-bars. Основните предимства на Template7 са неговото високо бързодействие (над 3 тъпи

по-бърз от Handlebars) и малък размер (около 1KB в gzip формат). Пътят до страницата 404 трябва да бъде описан последен.

```
routes = [
  {
    path: '/',
    url: './index.html',
  },
  {
    path: '/settings/',
    url: './pages/settings.html',
  },
  {
    path: '/page-template-1/',
    templateUrl: './pages/page-template-1.html',
  },
  {
    path: '/show/news/:newsId/
    async: function (to, from, resolve, reject) {
      let router = this;
      let app = router.app;
      let newsId = to.params.newsId;
      let newsInfo = app.data.newsdata[newsId];
      resolve(
        {
          componentUrl: './pages/page-template-2.html',
        },
        {
          context: {news: newsInfo}
        }
      );
    }
  },
  {
    path: '/profile/',
    url: 'profile.html',
    beforeEnter: function ({ resolve, reject }) {
      if ( /* Проверка дали е позволен достъпа */) {
        // Има достъп
        resolve();
      } else {
        // Няма достъп
        reject();
      }
    },
  },
  // Default route (404 page). MUST BE THE LAST
  {
    path: '(.*)',
    url: './pages/404.html',
  },
];
```

1 2 3 4 5

Фиг. 6-3 Примерно съдържание на файл routes.js

В **Секция 3** е показан вариант на маршрутизация при който се изпълнява JavaScript асинхронна функция преди да се реализира прехода към желания ресурс. Този синтаксис се използва с цел проверка дали преходът към ресурса е възможен в този момент или с цел получаване на данните, необходими за компонента-шаблон. При конкретният пример

към ресурса /show/news/ се предава параметър newsId. Тъй като в зависимост от стойността на newsId трябва да се предадат различни данни на шаблона, те се извличат от тялото на JavaScript функция. Тя се дефинира като стойност на свойство async. Функцията получава 4 аргумента:

- to – обект чрез който се описва заявения ресурс.
- from – обект чрез който се описва източника на заявката.
- resolve – функция, която позволява извикването на ресурса.
- reject – функция, която забранява извикването на ресурса.

В **Секция 4** е описано как може да блокираме достъпа до даден ресурс. За целта може да приложим JavaScript функция към избрано от нас събитие, свързано с дадена страница. При конкретния пример се прехваща събитието beforeEnter. То се генерира непосредствено преди рендиране на страницата.

В **Секция 5** се задава кой документ да се зареди, ако се адресира ресурс, който маршрутезатора не разпознава. В конкретният случай това е страницата 404.html. Свойството path трябва да има стойност '(.*)'. Това е регулярен израз, който описва кой да е път, който не е описан до този момент. Ето защо пътят за страница 404 трябва да се декларира последен.

VI. Инициализация на приложението

Инициализацията на приложението най-често се реализира чрез програмния код от файл app.js. Тя се свежда основно до създаване на следните обекти:

- Получаване на инстанцията на обект с цел достъп до DOM7. Framework7 не зависи от каквито и да външни библиотеки, включително и от DOM. DOM7 предоставя голямо разнообразие от бързи методи за DOM манипулиране. DOM7 е достъпен чрез глобалната променлива Dom7, но по-добрия вариант е да се дефинира локална променлива за достъп до DOM7. В конкретния случай това е променливата с име $$. Не се използва $, тъй като ще бъде в конфликт с референциите на библиотека jQuery или Zepto.
- Получаване на инстанцията на обект за достъп до ядрото на програмната рамка - Framework7 Core. Това е локалната променлива с име app (може да е друго, избрано от вас име), която се получава чрез викане на конструктора на Framework7. Конструкторът получава като аргумент JSON обект. Този JSON съдържа свойства и техните стойности, които са необходими за инициализация на приложението.

На Фиг. 6-4 е показан примерен програмен код за инициализация на едно Framework7 приложение. По-важните секции от този код са следните:

- **Секция 1**. В тази секция са описани базови свойства като: name (име на приложението); theme (име на CSS темата) и id (идентификатор на приложението) – необходим е при качване на приложението в магазин за приложения.
- **Секция 2**. В тази секция е описано как да дефинираме данни, които да са достъпни за всека страница, част от приложението. В конкретния случай свойство login съдържа информация, необходима за достъпване на услуга чрез име на потребител и парола.
- **Секция 3**. В тази секция са дефинирани всички методи, които трябва да са достъпни за всички страници, който изгражда приложението. В конкретния случай има само един root метод с име method1.

- **Секция 4**. В тази секция е зададена информация за всички налични пътища за маршрутизатора – свойство routes се инициализира чрез масива routes.

```
// Dom7
var $$ = Dom7;
// Framework7 App main instance
var app  = new Framework7({

  root: '#app',               // App root element, el: for v6
  id: 'bg.tugab.mis.app1', // App bundle ID
  name: 'App1',               // App name
  theme: 'auto',              // Automatic theme detection

  // App root data
  data: function () {
    return {
      login: {
        username: 'dstionterystalletlypecel',
        password: 'be0864bb0315d45b28d333d991b21fda4e22b672',
      },
    };
  },

  // App root methods
  methods: {
    method1: function () {
      app.dialog.alert('Hello');
    },
  },

  // App routes
  routes: routes,

});

// Init/Create views
var mainView = app.views.create('.view-main', {
    url: '/'
});
var leftView = app.views.create('.view-left', {
    url: '/'
});
```

1 2 3 4 5

Фиг. 6-4 Инициализация на приложението (файл app.js)

- **Секция 5**. В тази секция е реализирана инициализацията на два view обекта. Чрез тях се реализира превключване между индексната страница (index.html) и страницата settings.html. За целта се използва метод create. Като първи аргумент се задава CSS селектора на елемента, който ще бъде контейнер за съответната страница. При конкретния случай това са div контейнери с клас “view-main” и “view-left”. Като втори аргумент се задава url, който сочи пътя, необходим на маршрутизатора.

VII. Основни компоненти от изгледа на приложението

Програмна рамка Framework7 предоставя над 65 графични компоненти. Чрез своята CSS библиотека тази програмна рамка предоставя възможност за визуализиране на всеки един компонент в избран от вас изглед по начин, който симулира дизайна на тези компоненти при операционни системи Android и iOS (нативен изглед).

Стиловото форматиране се задава чрез специфични стойности на атрибут class. Следва описание (виж Фиг. 6-5) на по-често използваните на практика графични компоненти (изглед и програмен код):

Изглед на графичен компонент	HTML код
Текст в параграф	`<p> Текст в параграф </p>`

БУТОН БУТОН ЗАП...	`<p class="row">`
	`  <a href="#" class="col button">Бутон</a>`
	`  <a href="#" class="col button button-fill">`
	`  Бутон запълнен`
	`  </a>`
	`</p>`

БУТОН ПОВ... ПОВДИГНА...	`<p class="row">`
	`  <a href="#" class="col button button-raised">`
	`  Бутон повдигнат`
	`  </a>`
	`  <a href="#" class="col button button-raised button-fill">`
	`  Повдигнат запълнен`
	`  </a>`
	`</p>`

БУТОН ЗАК... ЗАКРЪГЛЕН...	`<p class="row">`
	`  <a href="#" class="col button button-round">Бутон закръглен</a>`
	`  <a href="#" class="col button button-round button-fill">Закръглен запълнен</a>`
	`</p>`

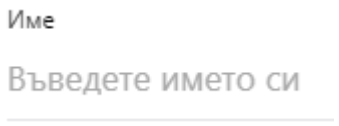
БУТОН С К... С КОНТУР ...	`<p class="row">`
	`  <a href="#" class="col button button-outline">`
	`  Бутон с контур`
	`  </a>`
	`  <a href="#" class="col button button-round button-outline">`
	`  С контур закръглен`
	`  </a>`
	`</p>`

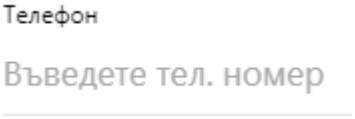
МАЛЪК С КО... С КОНТУР ЗА...	`<p class="row">`
	`  <a href="#" class="col button button-small button-outline">`
	`  Малък с контур`
	`  </a>`
	`  <a href="#" class="col button button-small button-round button-outline">`
	`  С контур закръглен`
	`  </a>`
	`</p>`

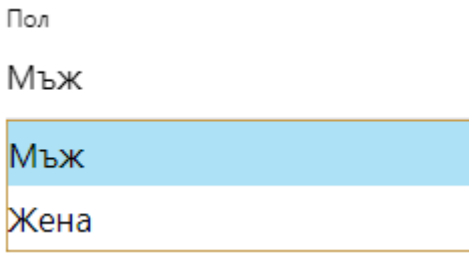
БУТОН МАЛЪК МАЛЪК ЗАКР...	`<p class="row">`
	`  <a href="#" class="col button button-small button-fill">`
	`  Бутон малък`
	`  </a>`
	`  <a href="#" class="col button button-small button-round button-fill">`
	`  Малък закръглен`
	`  </a>`
	`</p>`

	`<p class="row">`
	`  <a href="#" class="col button button-large button-raised">`
	`  Бутон голям`
	`  </a>`
	`  <a href="#" class="col button button-large button-fill button-raised">`
	`  Голям запълнен`
	`  </a>`
	`</p>`

	`<p class="row">`
	`  <a href="#" class="col button button-large button-fill button-raised color-green">`
	`  Бутон голям зелен`
	`  </a>`
	`  <a href="#" class="col button button-large button-fill button-raised color-red">`
	`  Голям червен`
	`  </a>`
	`</p>`

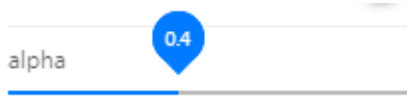
	`<div class="item-content item-input">`
	`  <div class="item-inner">`
	`  <div class="item-title item-label">`
	`  Име`
	`  </div>`
	`  <div class="item-input-wrap">`
	`  <input type="text"`
	`  placeholder="Въведете името си"/>`
	`  </div>`
	`  </div>`
	`</div>`

	`<input type="email" placeholder="E-mail"/>`
	`<input type="url" placeholder="Въведете URL"/>`
	`<input type="password" placeholder="Въветете парола"/>`
	`<input type="tel" placeholder="Въведете тел. номер"/>`



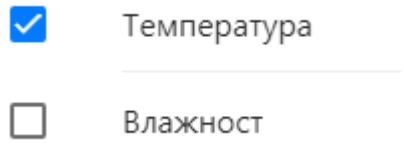

```
<div class="item-title item-label">Пол</div>
    <div class="item-input-wrap">
        <select>
            <option>Мъж</option>
            <option>Жена</option>
        </select>
    </div>
</div>
```


```
<div class="item-title item-label">
      Споделете Вашето мнение
</div>
<div class="item-input-wrap">
    <textarea placeholder="Според мен ..."
       class="resizable">
    </textarea>
</div>
```

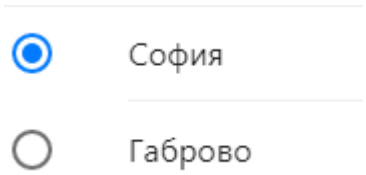



```
<div class="item-title item-label">
      Плъзгач
</div>
<div class="item-input-wrap">
    <div class="range-slider range-slider-init"
       data-label="true">
        <input type="range" value="0.5"
          min="0" max="1" step="0.1">
    </div>
</div>
```

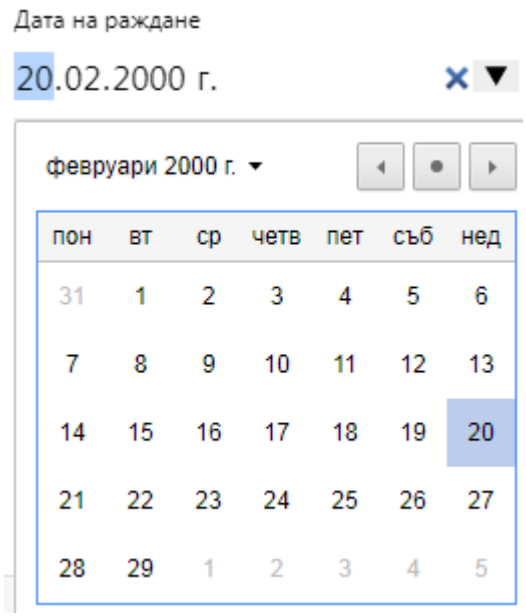

```
<div class="item-title">
      Вкл./Изкл.
</div>
<div class="item-after">
    <label class="toggle toggle-init">
        <input type="checkbox">
        <span class="toggle-icon"></span>
    </label>
</div>
```




```
<div class="list">
  <ul>
    <li>
      <label class="item-checkbox
         item-content">
      <input type="checkbox" name="data"
       value="Температура" checked="checked"/>
       <i class="icon icon-checkbox"></i>
       <div class="item-inner">
        <div class="item-title">
           Температура</div>
        </div>
      </label>
    </li>
    <li>
      Аналогичен код за влажност
    </li>
</div>
```




```
<li>
  <label class="item-radio item-content">
    <input type="radio" name="city"
       value="София" checked="checked"/>
    <i class="icon icon-radio"></i>
    <div class="item-inner">
      <div class="item-title">София</div>
    </div>
   </label>
</li>
<li>
  <label class="item-radio item-content">
    <input type="radio" name="city"
       value="Габрово"/>
    <i class="icon icon-radio"></i>
    <div class="item-inner">
      <div class="item-title">Габрово</div>
    </div>
  </label>
</li>
```




```
<div class="item-content item-input">
    <div class="item-inner">
        <div class="item-title item-label">
          Дата на раждане
        </div>
        <div class="item-input-wrap">
            <input type="date"
             placeholder="Въведи дата"
             value="2000-02-20"/>
        </div>
    </div>
</div>
```

Фиг. 6-5 Графични компоненти – изглед и програмен код

Подредбата на тези графични компоненти може да се реализира по четири различни начина:

- Статична подредба (static layout). Статичното оформление се използва най-често и включва навигационна лента, лента с инструменти и съдържание, което може да бъде превъртано (скролирано). Всяка страница може да има своя навигационна лента и лента с инструменти.
- Фиксирано оформление (fixed layout). Фиксираното оформление включва собствена навигационна лента и лента с инструменти, но съдържанието на страниците не може да се превърта.
- Through Layout. При това оформление навигационната лента и лентата с инструменти се показват фиксирани за всички страници в един изглед.
- Смесена подредба (mixed layout). При тази подредба можете да комбинирате различните видове подредби в един изглед.

VIII. Събития

Програмната рамка Framework7 предоставя интерфейс за създаване, следене и предизвикване (емитиране) на събития. Събитията в средата на Framework7 са обекти (event handlers). Тези обекти предоставят методи чрез които е възможно създаване на обект-събитие (еднократно или многократно следене на събитие), изтриване на обект-събитие, както и програмно активиране на събитие:

`[instance].on(event, handler)`	Създаване на обект-събитие.
`[instance].once(event, handler)`	Създаване на обект-събитие, който се изтрива веднага след като събитието се генерира.
`[instance].off(event)`	Изтриване на обект-събитие.
`[instance].emit(event, …args)`	Програмно генериране на събитие.

Например, ако желаете да прехванете инициализацията на всяка страница може да използвате следния програмен код:

```
app.on('pageInit', function (page) {
    app.dialog.alert (page.name); // показва името на страницата

});
```

Ако трябва да прехванете момента на инициализация на точно определена страница (в случая settings) използвайте следния програмен код:

```
$$(document).on('page:init', '.page[data-name="settings"]', function (page) {
    app.dialog.alert('Init settings page');
});
```

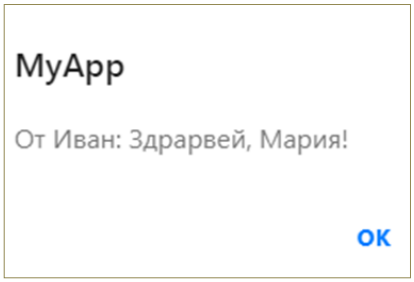
Нека да създадем потребителско ориентирано събитие с име “message” чрез което да изпращаме съобщения между страници от приложението. Прехващането (следенето) на това събитие се реализира със следния програмен код:

```
app.on('message', function(data) {
    app.dialog.alert(`От ${data.from}: ${data.text}, ${data.to}!`);
});
```

Когато се генерира събитието, към callback функцията се предна JSON обект, който има три свойства: from – кой изпраща съобщението; to – кой е получателя на съобщението и text – текстово описание на съобщението. Генерирането на събитието се реализира чрез метод emit:

```
app.emit('message', {from: 'Иван', to: 'Мария', text: 'Здрарвей'});
```

В резултат на това callback функцията ще визуализира Alert box, показан на Фиг. 6-6.



Фиг. 6-6 Резултат, получен след генериране на потребителско събитие “message”

IX. Работа с документи-шаблони

Съдържанието, което се доставя и интегрира в основната страница може да се получи по три начина:

- Статично – съдържанието се получава от HTML документ.
- Изцяло динамично – съдържанието се генерира програмно (JavaScript).
- Смесване на статично и динамично генерирано съдържание – в този случай може да се използват документи – шаблони.

Шаблоните (templates) позволяват адаптивно генериране на част от HTML кода, в зависимост от моментните стойности на определени променливи. Програмната рамка Framework7 има пълна интеграция на двигателя за шаблони Template7. Той е с малък размер и изключително бърз. Можете да я използвате при версия на Framework7 до 5.x.

Примерната структура на документ-шаблон е показана на Фиг. 6-7.

```
<template>
    <div class="page" data-name="login">
      <div class="navbar">                                          1
        <div class="navbar-inner sliding">
          <div class="left">
            <a href="#" class="link back">
              <i class="icon icon-back"></i>
              <span class="ios-only">Назад</span>
            </a>
          </div>
          <div class="title">Заглавие</div>
        </div>
      </div>
      <div class="page-content">                                    2
         <!-- Вмъкнете тук вашия HTML код -->
      </div>
    </div>
  </div>
</template>

<script>
  return {                                                          3
    data: function () {
        return {

        };
    },
    methods: {                                                      4
        method1: function() {

        },
    },
    on: {                                                           5
      pageInit: function(e, page) {

      },
    }
  };
</script>
```

Фиг. 6-7 Структура на документ-шаблон

Страницата се състои от две части:

- Секция template.
- Секция script.

В тялото на секция **template** е HTML кода на страницата. Ако е необходимо страницата може да има собствена навигационна лента и заглавие (**Секция 1**). Основният HTML код трябва да бъде в div контейнер от клас с име “page-content” (**Секция 2**).

В тялото на секция **script** е JavaScript кода, който се асоциира със страницата:

- **Секция 3** – предоставя възможност страницата да получи променливи, от които зависи съдържанието на страницата.
- **Секция 4** – JSON обект, който съдържа методи, които са видими в контекста на страницата.
- **Секция 5** – списък от callback функции, свързани с жизнения цикъл на страницата, например: pageMounted, pageInit, pageBeforeIn, pageAfterIn, pageBeforeOut, pageAfterOut и pageBeforeRemove).

9.1. Синтаксис на Template7

Templete7 е двигател на шаблони, който е активен по-подразбиране при Framework7 до версия 6.x. Template7 поддържа променливи (variables), блок-изрази (block expressions) и помощници (helpers).

Синтаксисът за интегриране на **променливи** в HTML контекст е следния:

- {{variable}} – вмъква стойността на променливата variable в текущия контекст.
- {{../variable }} - вмъква стойността на променливата variable в родителския контекст.
- {{this}} – вмъква стойността на променливата, чиято стойност е текущия контекст.
- {{object.property}} – вмъква в текущия контекст стойността на свойство property от обект object.

Синтаксисът на **блок-изразите** е следния:

- {{#each}} – начало на блок-израз (цикъл).
- {{/each}} – край на блок-израз.
- {{#each reverse="true"}} - начало на блок-израз с реверсиране на последователността на извличане на параметрите.
- {{else}} – начало на блок-израз с инверсна логика на {{#each}}.

Помощниците в Template7 са като предварително дефинирани функции, които извършват зададена логика с подавания им контекст. Синтаксисът на вградените в Template7 помощници е следния:

1. `{{#each}}...{{else}}...{{/each}}` - позволява последователна обработка на елементите на масив или свойствата на обект. След `#each` следва името на масив или JSON обект. Елементите на масива или свойствата на обекта се обработват последователно. В тялото на `#each` е възможно да използвате следните Template7 променливи:
   - `@index` - номер на текущия обработван елемент от масив.
   - `@first` - връща true, ако се обработва първият елемент от масив.
   - `@last` - връща true, ако се обработва последния елемент от масив.
   - `@key` - име на текущо свойство от JSON обект.

2. `{{#if}}...{{else}}...{{/if}}` – реализира рендиране на съдържанието от тялото на #if, ако контекста подаван към `#if` не е false, undefined, null или 0, в противен случай се реандира съдържанието от тялото на else.
3. `{{#unless}}...{{else}}...{{/unless}}` - реализира рендиране на съдържанието от тялото на `#unless`, ако контекста подаван към `#unless` е false, undefined, null или 0, в противен случай се реандира съдържанието от тялото на else.
4. `{{#with}}...{{/with}}` – заменя рендирания контекст с подавания към `#with` контекст (JSON обект).
5. `{{#variableName}}...{{/variableName}}` – Ако подавания контекст е масив този помощник работи като #each, ако е обект – работи като #with.
6. `{{join delimiter=""}}` – обединява елементите от масив в един низ, като разделител между елементите е символа зададен чрез delimiter.
7. `{{js "expression"}}` – позволява изпълнение на JavaScript код от тялото на шаблона. Този код най-често се използва с цел промяна на стойностите на свойствата на обекта, които е входен за шаблона контекст. В този случай свойствата се адресират чрез `this`, напрмер `this.property1`.
8. `{{js_if "expression"}}…{{/js_if}}` – позволява изпълнение на if-else от тялото на шаблона. Изразът (expression) е произволен JavaScript израз, който трябва да връща true или false.
9. `{{@root}}` – служи за достъп до root променливи, например: @root.username.

9.2. Практическо използване на документи-шаблони

9.2.1. Задача 1

Трябва да се визуализира по различен начин списък с книги. Те са описани чрез масив с име books, съдържащ JSON обекти (виж Фиг. 6-8).

```
books: [
    {
        title: 'JavaScript - Въведение в програмирането',
        author: 'Росен Иванов',
        year: 2018,
        instock: true,
    },
    {
        title: 'Eloquent JavaScript - A Modern Introduction to Programming',
        author: 'Marijn Haverbeke',
        year: 2014,
        instock: false,
    },
    {
        title: 'JavaScript for impatient programmers',
        author: 'Axel Rauschmayer',
        year: 2019,
        instock: true,
    },
]
```

Фиг. 6-8 Програмно описание на книгите от Задача 1

Ще използваме документ-шаблон чрез който книгите да се подредят в изгледа на страницата по четири различни начина:

1. Всички книги последователно във формат автор, заглавие и година.

2. Всички книги, които са в продажба.
3. Всички книги в номерирана последователност. За всяка книга, на нов ред да се разпечата името на текущо свойство и неговата стойност.
4. Всички книги, които са в продажба и са издадени през 2019 година.

На Фиг. 6-9 е показано примерно решение на четирите варианти на задачата.

1
```
<p>
  <ul>
    {{#each books}}
      <li>
        {{author}}, {{title}}, {{year}}
      </li>
    {{/each}}
  </ul>
</p>
```

2
```
<p>
  <ul>
   {{#books}}
     {{#if instock}}
       <li>
        {{author}}, {{title}}, {{year}}
       </li>
     {{/if}}
   {{/books}}
  </ul>
</p>
```

3
```
<p>
  <ul>
    {{#each books}}
      <li>
        Книга {{@index+1}}:
        {{#each this}}
          <div>{{@key}}: {{this}}</div>
        {{/each}}
      </li>
    {{/each}}
  </ul>
</p>
```

4
```
<p>
 <ul>
  {{#books}}
   {{#js_if "this.instock === true
        && this.year === 2019"}}
    <li>
     {{author}}, {{title}}: {{year}}
    </li>
   {{/js_if}}
  {{/books}}
 </ul>
</p>
```

Фиг. 6-9 Примерно решение на Задача 1

9.2.2. Задача 2

Да се напише документ-шаблон, който реализира форма за проверка на автентичността. За да не се въвежда името и паролата всеки път, те да се интегрират във формата програмно. Името и паролата да се получават от обект login (виж Фиг. 6-4, Секция 2).

Програмният код, който реализира Задача 2 е показан на Фиг. 6-10. Първоначално се получава обекта login (Секция 1), който се инициализира със стойността на root променлива login. След рендиране на съдържанието на HTML кода от контейнера от клас “page-content”, стойностите на атрибути value на етикети input се инициализират от JSON обекта login:

- Име на клиента: `{{login.username}}`
- Парола на клиента: `{{login.password}}`

При натискане на бутон ИЗПРАТИ се вика метод submit (виж Секция 2). Забележете, че викането на този метод се реализира чрез специфичния за Framework7 атрибут @click. Извличането на името и паролата на клиента се реализира чрез синтаксис, подобен на jQuery - създава се референция към желания DOM обект чрез CSS селектор, а чрез метод `val` се прочита въведената стойност за име на потребител и парола:

- `input[type=text]` // извлича името
- `input[type=password]` // извлича паролата

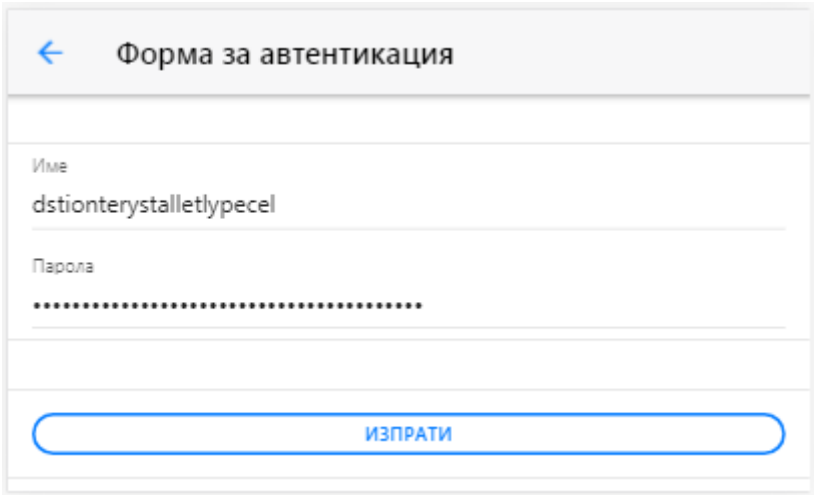
```
<template>
    <div class="page" data-name="login">
        Код от Секция 1, Фиг. 7
        <div class="page-content">
            <div class="list">
              <ul>
                <li class="item-content item-input">
                  <div class="item-inner">
                    <div class="item-title item-label">Име</div>
                    <div class="item-input-wrap">
                      <input type="text" name="username"
                         value="{{login.username}}" placeholder="Въведете име">
                    </div>
                  </div>
                </li>
                <li class="item-content item-input">
                  <div class="item-inner">
                    <div class="item-title item-label">Парола</div>
                    <div class="item-input-wrap">
                      <input type="password" name="password"
                         value="{{login.password}}" placeholder="Въведете парола">
                    </div>
                  </div>
                </li>
            </ul>
        </div>
        <div class="block block-strong">
            <a href="#" class="button button-small button-round button-outline"
              @click="submit()">
                ИЗПРАТИ
            </a>
        </div>
    </div>
</div>
</template>

<script>
  return {
    data: function () {
        return {
            login: this.$root.login,
        };
    },
    methods: {
        submit: function() {
            let username = $$('input[type=text]').val();
            let password = $$('input[type=password]').val();
            // Следва код, който изпраща името и паролата
            // на ресурс за проверка на автентичността.
        },
    },
  };
</script>
```

1

2

Фиг. 6-10 Примерна реализация на страницата с форма за проверка на автентичността

Резултатът, получен след зареждане на документа-шаблон, е показан на Фиг. 6-11.



Фиг. 6-11 Визуално представяне на формата за проверка на автентичността

9.2.3. Задача 3

Да се напише документ-шаблон, който визуализира сортирано състоянието на релета (идентификатор на релето и състояние на всяко реле). Да се предостави възможност клиентът да може да промени състоянието (включено или изключено) на избрани релета. Идентификаторите на релетата са низ, а състоянието – true (включено) или false (изключено). Документът-шаблон трябва да има преизползваем дизайн – при промяна на броя или идентификаторите на релетата, кодът на документа не трябва да се пише наново.

Релетата ще опишем чрез JSON обект, както е показано на Фиг. 6-12.

```
relays: [
    {
        id: 'relay-012',
        on: true
    },
    {
        id: 'relay-156',
        on: false
    },
    {
        id: 'relay-002',
        on: false
    },
    {
        id: 'relay-032',
        on: true
    }
],
```

Фиг. 6-12 Програмно описание на релетата

Този обект може да бъде записан в локална или отдалечена база данни, но за опростяване на кода ще дефинираме обекта като root данни за приложението.

Тъй като релетата трябва да се сортират по идентификатор, ще използваме JavaScript метод `sort` (виж Фиг. 6-13). Логиката на сортирането е зададена чрез ламбда функция, която се предава като аргумент към метод `sort`. Логиката за получаване на сортиране във възходящ ред, е следната:

a.id < b.id ? -1 : 1.

```
data: function () {
    return {
        relays: this.$root.relays.sort((a,b) => a.id < b.id ? -1 : 1),
    };
},
```

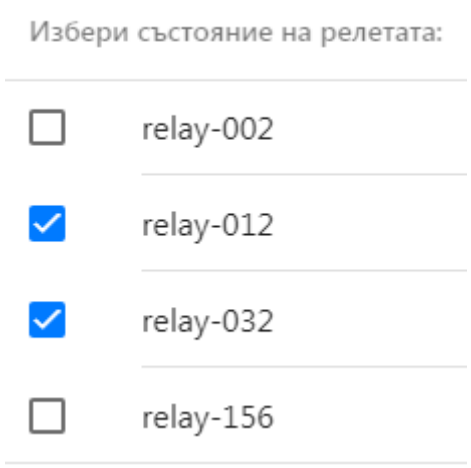
Фиг. 6-13 Получаване на сортирано по id описание на релетата

Ще използваме компонент “Check box”, за да визуализираме и променяме състоянието на релетата. Чрез възможностите на Template7 ще реализираме визуализиране на състоянието на релетата (виж Фиг. 6-14).

```
<div class="block-title">Избери състояние на релетата:</div>
<div class="list">
    <ul>
        {{#each relays}}
            <li>
                <label class="item-checkbox item-content">
                    <input type="checkbox" name="relay" value="{{id}}"
                       {{#if on}} checked="checked"{{/if}}/>
                    <i class="icon icon-checkbox"></i>
                    <div class="item-inner">
                        <div class="item-title">{{id}}</div>
                    </div>
                </label>
            </li>
        {{/each}}
    </ul>
</div>
```

Фиг. 6-14 HTML код на документа-шаблон

Първо, чрез помощник `#each` ще визуализираме в цикъл състоянието на всяко реле. Второ, чрез помощник `#if` ще проверяваме стойността на свойство `on`, за да разберем дали да определим каква да бъде стойността на атрибут checked от елемент input. Ако стойността на свойство `on` е true, трябва да се вмъкне атрибут checked със стойност checked. Резултатът, получен след рендиране и визуализиране на документа-шаблон, е показан на Фиг. 6-15.



Фиг. 6-15 Визуализация на състоянията на релетата (сортирано и статус)

X. Framework7 версия 6

При версия 6.x на програмна рамка Framework7 им редица промени, които трябва да отчитате при разработване на мобилни хибридни приложения, както и при необходимост от миграция на програмен код, написан за по-старите версии на програмната рамка.

10.1. DOM7

Framework7 v6 използва новия Dom7 v3. В него са направени промени във всички методи за итерация, като .each() и .map(). По-конкретно, редът на аргументите е променен, за да съответства на нативните итератори на масиви на JavaScript.

10.2. App root

Свойството root от конструктура на клас Framework7 е преименувано на el.

10.3. Template7

Библиотеката за шаблони Template7 е премахната от Framework7 v6. Ако приложението ви все още се нуждае от нея, просто я инсталирайте ръчно. Това не означава, че не можете да работите със документи-шаблони. Програмната рамка вече има собствен двигател за поддържане на шаблони. За да интегрирате променливи в HTML кода използвайте познатия от JavaScript синтаксис ${variable-name}, вместо {{variable-name}}.

На Фиг 6-16 е показан програмния код на документ-шаблон, който използва новия синтаксис. Страницата се изгражда след изпълнение на JavaScript кода от тялото на етикет script. Вика се анонимна функция на която могат да се предадат като аргументи всички системни обекти, необходими за функционирането на страницата, например props (достъп до параметрите, които се предават на ресурса), $on, $f7, $update и др. В края на програмния код на JavaScript трябва изпълните оператор return $render.

```
<template>
  <div class="page">
    <div class="navbar">
      <div class="navbar-bg"></div>
      <div class="navbar-inner">
        <div class="title">Profile</div>
      </div>
    </div>
    <div class="page-content">
      First Name: ${user.firstName}
    </div>
  </div>
</template>
<script>
  export default (props, { $on, $f7, $update }) => {
    let user = null;
    $on('pageInit', () => {
      // request user data on page init
      $f7.request.get('https://api.website.com/resource-name').then((res) => {
        user = res.data;
        // trigger re-render
        $update();
      });
    })
    return $render;
  }
</script>
```

Фиг. 6-16 Синтаксис на документ-шаблон при Framework7 v6

10.4. Компонент Virtual List

С премахването на Template7 функцията itemTemplate за визуализиране на елементи от виртуален списък чрез шаблон Template7 вече не се поддържа. Тя трябва да бъде заменена с renderItem.

10.5. Маршрутизатор

Framework7 v6 има напълно нов синтаксис и функционалност за компонента Router. Програмният код на приложенията, които използват версии 4 и 5 трябва да бъде пренаписан за новия синтаксис на компонент Router.

10.6. Хранилище

При Framework7 v6 не може да инициализирате поле `data` от конструктура на клас Framework7. Ако трябва да използвате данни, които са достъпни за всички страници можете да използвате *Framework7 Store* – новото локално хранилище за данни. Състоянието (*state*) е единственият обект, който съдържа цялото състояние на ниво приложение. Действията (*actions*) се използват за промяна на състоянието, за асинхронни манипулации или за извикване на други действия, свързани с хранилището. Обработващите getters методи се използват за връщане на данни от състоянието на хранилището. Достъпът до хранилището (и неговото състояние) може да се осъществи директно чрез препратка към инстанцията на хранилището, която сме създали, или чрез достъп до свойството `store` на инстанцията на Framework7. За да извикаме действие, свързано с хранилището, трябва да извикаме метод `store.dispatch` с името на действието, което искаме да извикаме.

Заключение

В тази глава се запознахте с програмна рамка Framework7 чрез която може да разработвате приложения с нативен изглед за Android, iOS, Microsoft и Web. Получихте информация за структурата на Framework7 проектите. Научихте и какви са основните графични компоненти, които тази рамка предоставя с цел изграждане на потребителския интерфейс. Разбрахте какво е маршрутизация и как се реализира тя при тази програмна рамка. В главата са описани състоянията през които преминава всяка страница. Запознахте се с документите-шаблони и как се използват те при рамка Framework7. Научихте как се прехващат, генерират и предизвикват събития. Накрая на главата са описани по-важните промени, които Framework7 версия 6.x налага.

Глава 7

Достъп до вградените сензори

I. Въведение

За реализиране на интерфейса с потребителя могат да бъдат използвани различни начини за въвеждане и извеждане на информация. На настоящият етап, въвеждането на информация най-често се реализира чрез докосване на екрана в една или множество точки, а извеждането на информация – чрез визуализиране на текст, графика и видео. Разбира се, съвременните мобилни устройства предоставят множество алтернативни начини както за въвеждане, така и за извеждане на информация. Това позволява създаване на *мултимодални интерфейси* (комбинират няколко канала за въвеждане и извеждане на информация) и *осезаем тип интерфейси* (комуникация със заобикалящите ни предмети по начин, характерен за хората, например докосване и жестове).

Възможните алтернативни канали за въвеждане на информация са следните:

- Разпознаване на говор – контрол и навигация чрез натурален говор или речеви команди. За целта се използват API за преобразуване на говор в текст, например Google Speech API и Bing Speech API. Основната част от тези услуги са облачно базирани и изискват регистрация, за да могат да бъдат използвани. При операционна система Android можете да използвате Speech API, което се базира на Google Speech API.
- Въвеждане на информация чрез доближаване на мобилното устройство до друго подобно или предмет от заобикалящият ни свят. За целта се използва технология Near Field Communications (NFC).
- Въвеждане на информация при доближаване до определена локация или обект. За целта се използва технология iBeacons.
- Промяна на логиката на работа на приложението в зависимост от положението на мобилното устройство в пространството. За целта се използват данните от вградения акселерометър, жироскоп и барометър.
- Промяна на логиката на работа на приложението в зависимост от посоката в която сочи мобилното устройство. За целта се използват данните от вградения компас.

Възможните алтернативни канали за извеждане на информация са следните:

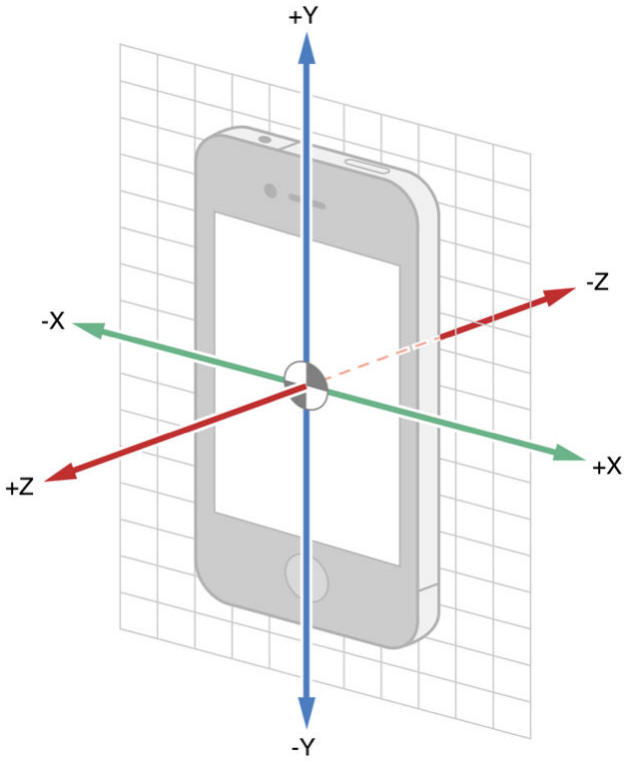
- Преобразуване на текст в говор – Text To Speech (TTS). За да реализирате това преобразуване (синтез на говор) може да използвате множество облачно базирани TTS API.
- Използване на вградения вибратор в мобилното устройство, за да се генерира вибро-кодирана информация.

II. Видове сензори

Най-често използваните сензори при мобилните устройства са акселерометър, жироскоп, компас и GPS приемник. Трябва да се има предвид, че в зависимост от ценовия клас на мобилното устройство някои от сензорите може да липсват или да са реализирани с евтини интегрални схеми, което води до много неточни резултати.

2.1 Акселерометър

Акселерометърът измерва ускорението в m/s^2 на мобилното устройство (телефон или таблет) в 3-мерното пространство. По подразбиране, данните от акселерометъра най-често се използват с цел разпознаване на позицията на мобилния телефон в пространството (режим portrait или landscape). Разбира се, вие може да използвате тези данни за реализация на друга функционалност. При мобилните устройства са вградени 3 акселерометъра, по един за всяка ос - X, Y и Z. Акселерометърът се използват за определяне на ускорението по всяка една от осите. Тези оси се определят спрямо мобилното устройство, а не спрямо земята (виж Фиг. 7-1).

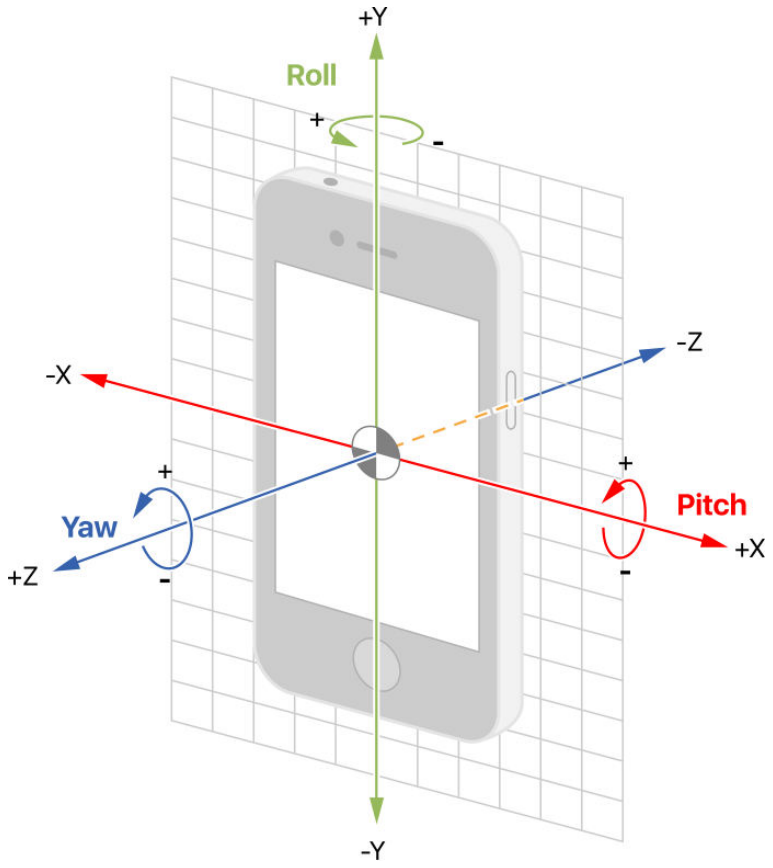


Фиг. 7-1 Ориентиране на посоките на ускорение спрямо позицията на телефона

Ако задържите екрана на телефона „към небето" и го придвижите нагоре, той ще отчете ускорение по оста Z. Ако го движите наляво и надясно, той ще открие промени по оста X. Същото важи и за оста Y, която засича движението нагоре и надолу (от гледна точка на телефона). Тъй като земната гравитацията също е ускоряваща сила, ако поставите мобилното устройство на равна повърхност с екрана нагоре, ще регистрирате ускорение приблизително 9.81 m/s^2 по оста Z, въпреки че телефонът не се движи. Така че, когато телефонът е в покой (или се движи с постоянна скорост), данните от акселерометъра не са 0, а са равни на земното гравитационното ускорение. Данните на акселерометъра са 0 само при свободно падане. Във всяка друга ситуация акселерометърът връща сумата от вектора на земната гравитацията и вектора на истинското ускорение.

2.2 Жироскоп

Ако искаме да определим въртенето на мобилното устройство спрямо трите оси, ще трябва да използваме данните от жироскопа. От акселерометъра не може да получим тази информация тъй като няма как да се определи дали ускорението е от завъртане или от земното ускорение. Жироскопът може да засича промените във въртенето (attitude) на телефона: наклон (pitch), преобръщане (roll) и отклонение (yaw) – виж Фиг. 7-2.



Фиг. 7-2 Ориентиране на посоките на въртене спрямо позицията на телефона

Чрез жироскопа може да измерим промените във въртенето на телефона по всяка една от осите – степен на завъртане и посока на завъртане. Тъй като жироскопът не се влияе от земната гравитация, той трябва да определя позицията спрямо себе си. Поради това се получава така наречения “дрейф”. Всяко текущо измерване на завъртане се базира на предишното завъртане (референтната рамка), като се добави откритата промяна в завъртането. Това с течение на времето води на натрупване на грешка.

Жироскопът сам по себе си не е достатъчен, за да се изчисли положението на телефона. Данните от жироскопа най-често се обединяват с данните от акселерометъра. Ускорението изкривява вектора на гравитация спрямо възприетия вектор на гравитация. Акселерометърът всъщност определя възприеманата гравитация, а не действителната гравитация или ускорение. Като използваме информацията за въртенето от жироскопа, можем да оценим какъв трябва да бъде действителният вектор на гравитация и да го извадим от възприетия вектор на гравитация, за да изчислим истинското ускорение. По подобен начин акселерометърът може да се използва за повторно калибриране на референтната стойност на жироскопа и за получаване на по-точна оценка за въртенето на мобилното устройство. Когато устройството е в покой, изходът на акселерометъра е равен на вектора на гравитация. Дрейфът на жироскопа може да се коригира периодично, като се използва векторът на гравитацията от акселерометъра.

2.3 Магнитометър

Ако трябва да се определи посоката, в която се движите спрямо земята, ще трябва да анализирате данните от друг сензор наречен магнитометър. Този сензор определя коя е посоката на магнитния север. Този сензор се нуждае от информация за въртенето на мобилното устройство, тъй като въртенето на телефона влияе върху оценката на

магнитното поле. Точна оценка на ориентацията на мобилното устройство може да се получи само след комбиниране на данните от магнитометъра с векторите на гравитацията и истинското ускорение. Магнитометърът обикновено не натрупва грешки с течение на времето, но е много бавен при засичането на движения и чувствителен към обекти, които имат собствено магнитно поле.

При повечето практически задачи се налага обединяване на данните от акселерометъра, жироскопа и магнитометъра (sensor fusion). Най-често за целта се използва цифров филтър на Калман. По този начин се минимизират недостатъците на всеки един от тези три сензора.

2.4 Барометър

Чрез барометъра може да се определи вертикално местоположение на мобилното устройство. При промяна на височината, барометърът засича получената разлика в налягането. Това е единственият начин да определим къде се намираме във височина в затворени пространства, където данните от GPS системата са недостъпни. Следователно, сензорът за налягане се използва основно за подобряване на оценката за местоположението на клиентите на услуги които използват това – Location Based services (LBS).

Сами по себе си барометрите не генерират достатъчно данни, за да изчислят точно вертикалното местоположение. Това е така, защото промените в налягането могат да се случат по множество причини, а не само поради реална промяна на височината. Може да се движите с асансьор или да има рязка промяна на времето при което атмосферното налягане се променя значително за кратко време. Ако времето ще бъде хубаво атмосферното налягане се покачва. Ако налягането се понижава, тогава вероятно ще вали дъжд, сняг или ще се появи друг вид лошо време, например буря. Барометърът няма как да направи разлика между реална промяна на височината на която се намира мобилното устройство и промяната на атмосферното налягане поради климатични промени.

Единственият начин да отстраните грешките при определяна на височината поради промените в атмосферното налягане е да е използвате данните от съществуващите фирми за измерване на местното налягане в реално време. Например, ако се намирате в САЩ може да използвате услугата Metropolitan Beacon System (MBS). Тя е разработка на NextNav. Тази компания става публична на Nasdaq през октомври 2021 г. след сливане с компанията със специални цели Spartacus Acquisition Corporation. Услугата MBS е предназначена да се използва в райони, където GPS или други сателитни сигнали за местоположение не могат да бъдат надеждно приемани. NextNav разгръща национална мрежа от хиперлокални метеорологични станции, захранвани от специализирани сензори за налягане. Тези сензори обработват данните от барометъра в телефоните на клиентите, като изпращат обратно данни за местните метеорологични условия, позволявайки на телефона да "настройва" промените в налягането, които не са свързани с надморската височина.

2.5 Global Positioning System (GPS)

Първоначално Глобалната Система за Позициониране (GPS) е създадена от САЩ за военни цели, но малко по-късно през 1980 г. е разрешена и за гражданска употреба. За да не може да се използва за военни цели от други страни се вмъква грешка при изчисляване на местоположението. Точността на локализация при GPS за граждански

цели е около 10m. При GPS най-малко 24 сателита са винаги в орбита около Земята. GPS приемниците не се свързват със сателитите и не предават информация към тях. Те само получават данни от тях. Спътниците са разположени в геостационарна орбита така, че поне четири от тях се „виждат" от всяка точка на Земята. За да получите точна оценка за местоположението е необходима пряка видимост между мобилното устройство и сателитите. Системата GPS не е единствената мрежа от спътници, която може да се използва за позициониране. Алтернативи на GSP са ГЛОНАСС (Русия), BDS (Китай) и "Галилео" (Европейски Съюз).

GPS спътниците постоянно предават радиосигнали към Земята. Всяко предаване включва местоположението на GPS спътника и времето, в което е изпратен сигналът. Всеки сателит има атомен часовник на борда, така че времето е много точно. Вградените в мобилните устройства GPS сензори работят като GPS приемник. Всеки GPS приемник „подслушва" сигналите от три или повече спътника. При налични три спътника може да се определи географската ширина и географската дължина. При наличие на четири или повече сателити може да се определи и надморската височина, както и посоката и скоростта на предвижване.

Сигналите от по-близките спътници пристигат по-рано, докато сигналите от по-отдалечените спътници пристигат по-късно. Действителната разлика във времето на получаване на радиосигналите е много малка, но може да бъде засечена от GPS приемника. Като сравнява времето на излъчване на сигнала и времето на пристигане на сигнала, приемникът може да оцени относителното разстояние между GPS приемника и видимите спътници. Използвайки метода на *трилатерация*, може да се определи местоположението на GPS приемника.

GPS не е единственият начин, по който устройствата могат да определят текущото ви местоположение. В зависимост от силата на сигналите от базовите станции на мобилните оператори е възможно да се определи местоположението на мобилните устройства. И в този случай се използва метода на трилатерация, но на базата на измерване на разликата в силата на сигналите от няколко (минимум три) базови станции. Някои GPS приемници използват данните от базовите станции и данните от GPS системата – Assist-ed GPS (A-GPS). Целта е по-бързо получаване на местоположението. Алгоритмите за изчисляване на GPS локацията връщат резултат полкова по-бързо, колкото по-точна е началната оценка за локацията. При A-GPS началната локация се получава чрез оценка за местоположението от базовите станции на мобилния оператор.

Друг начин за определяне на локацията на мобилните устройства е на да се използва информация от бази данни с информация за местоположението на Wi-Fi точки на достъп. Например, Google записва в база данни MAC адресите на всички Wi-Fi точки за достъп и относителната сила на сигнала в определени локации. За да получите местоположението си, е необходимо мобилното устройство да сканира за близки безжични мрежи, след което да изпрати тази информация до сървърите на Google. Като отговор ще получите вашата локация като географска ширина и географска дължина.

III. Достъп до акселерометъра

Достъпът до акселерометъра е възможен чрез плъгина "cordova-plugin-device-motion". Този плъгин може да се използва при множество мобилни ОС, в това число Android, iOS и Windows. За да се активира четенето на данни от акселерометъра може да се използват две функции:

- getCurrentAcceleration – Използва се при еднократно измерване на ускорението.

- watchAcceleration – Използва се при необходимост от многократно измерване на ускорението през определен интервал от време.

На Фиг. 7-3 е показано как се използва функция watchAcceleration.

```
var accelerometerId = null;

function startAccelerometer() {
    var options = {frequency: 500};
    accelerometerId =
       navigator.accelerometer.watchAcceleration(onSuccessAcc,onErrorAcc,options);
}

function onSuccessAcc(acceleration) {
    var x = acceleration.x;
    var y = acceleration.y;
    var z = acceleration.z;
    $$("#accelerometer").html("X: "+x+"<br/>Y: "+y+"<br/>Z: "+z+"<br/>");
}

function onErrorAcc() {
    $$("#accelerometer").html("Грешка при достъп до акселерометъра!");
    stopAccelerometer();
}
```

Фиг. 7-3 Работа с акселерометъра

Тя изисква три параметъра:

- onSuccessAcc – функция, която се вика при налични данни от сензора.
- onErrorAcc – функция, която се вика при грешка при функциониране на сензора.
- options – JSON обект, чрез който се задава интервалът от време в ms през който се получават данни от акселерометъра.

Всеки път, когато се получат данни от акселерометъра, се вика функция $onSuccessAcc$. Тя получава като параметър програмен обект ($acceleration$), който съдържа информация за ускорението на мобилното устройство по трите оси (x, y и z). Ако поради някаква причина не може да се получи информация от акселерометъра се вика функция $onErrorAcc$. Когато трябва да преустановите четенето на данни от акселерометъра, трябва да извикате функция $clearWatch$. Как се реализира това е показано на Фиг. 7-4.

```
function stopAccelerometer() {
    if (accelerometerId) {
        navigator.accelerometer.clearWatch();
        accelerometerId = null;
    }
}
```

Фиг. 7-4 Прекратяване на достъпа до акселерометъра

IV. Достъп до компаса

Достъпът до компаса е възможен чрез плъгина “cordova-plugin-device-orientation”. Този плъгин може да се използва при Android, iOS и Windows. За да се активира четенето на данни от компаса може да се използват две функции:

- getCurrentHeading – Използва се при еднократно измерване на посоката.
- watchHeading – Използва се при необходимост от многократно измерване на посоката през определен интервал от време.

Магнитният компас връща посоката в която е насочен мобилния телефон спрямо северния магнитен полюс (0°). Ще разгледаме четенето на данни от компаса чрез функция watchHeading. Тя изисква три параметъра:

- onSuccessCompass – функция, която се вика при налични данни от сензора;
- onErrorCompass – функция, която се вика при грешка при функциониране на сензора;
- options – JSON обект, чрез който се задава време-интервалът в ms през който се получават данни от компаса.

Използването на компаса е аналогично на използването на акселерометъра (виж Фиг. 7-5). При наличие на компас, започва да се вика функция onSuccessCompass през интервал, зададен чрез параметър options. На функцията се предава един параметър – обект, който съдържа информация от компаса. При конкретният пример се извлича стойността за посоката, при условие, че компасът е магнитен. Трябва да се има предвид, че вграденият компас може да е от електронен тип, а не магнитен. Магнитният компас сочи северният магнитен полюс, а електронният – северният полюс. Когато не трябва повече да получавате данни от компаса трябва да извикате функция clearwatch (виж Фиг. 7-6).

```
var compassId = null;

function startCompass() {
    var options = {frequency: 1000};
    compassId = navigator.compass.watchHeading(
              onSuccessCompass, onErrorCompass, options);
}

var onSuccessCompass = function(heading) {
    var value = Math.abs(heading.magneticHeading);
    $("#compass").html("Посока: " + value + "&deg;");
};

function onErrorCompass() {
    $("#compass").html("Грешка при достъп до компаса!");
    stopCompass();
}
```

Фиг. 7-5 Функции за работа с компаса

```
function stopCompass() {
    if (compassId) {
        navigator.compass.clearWatch(compassId);
        compassId = null;
    }
}
```

Фиг. 7-6 Прекратяване на четенето на данни от компаса

V. Достъп до GPS приемника

Ще използваме W3C Geolocation API, за да получим данни от вградения в мобилното устройство GPS приемник. За да получите достъп до Geolocation API е необходимо да използвате плъгина “cordova-plugin-geolocation”. Geolocation API предоставя възможност за получаване на позицията на мобилния клиент чрез по два начина:

- Груба оценка на позицията на клиента основно чрез трилатерация (на базата на известни MAC адреси на WiFi точки на достъп или при известна позиция на три най-близки базови станции (Cell ID) на мобилния оператор.
- Точна оценка на позицията чрез GPS.

Функциите, които Geolocation API предоставя, са следните:

- getCurrentLocation - еднократно получаване на позицията на клиента.
- watchPosition - последователно във времето получаване на позицията на клиента през зададен интервал от време.

И двата метода изискват три аргумента както е показано на Фиг. 7-7.

```
var options = {enableHighAccuracy:true, timeout: 3000, maximumAge: 20000};
var gpsId1 = navigator.geolocation.getCerrentPosition(onSuccess, onError, options);
var gpsId2 = navigator.geolocation.watchPosition(onSuccess, onError, options);
```

Фиг. 7-7 Активиране на получаване на данни от GSP приемника

Аргументите имат следното предназначение:

- onSuccess – callback функция, която се вика, ако заявката се изпълни успешно.
- onError – callback функция, която се вика, ако заявката не може да се изпълни.
- options – параметри за инициализация в JSON формат.

Имената на свойствата на обект options и тяхното предназначение са следните:

- **enableHighAccuracy** – приема стойност true при необходимост от точна оценка за позицията чрез GPS или false - при груба оценка на позицията.
- **timeout** – колко време (в ms) да се изчака от генериране на заявката до извикване на функция onSuccess. Ако след този интервал не се извика onSuccess управлението се предава на функция onError. При метод watchPosition този параметър задава интервала през който се получава информация от GPS приемника.
- **maximumAge** – времеинтервал в ms, чрез който се задава максималното време на изчакване през който може да се използва кешираната позиция на клиента. След този интервал, задължително трябва да се получи новата позиция на клиента. Целта е услугите, използващи позицията на клиентите, да останат работоспособни дори когато за определен интервал от време не може да се получат данни от GPS приемника (при липса на видимост на достатъчен брой GSP сателити).

Прекратяването на функциониране на watchPosition се реализира чрез метод clearWatch:

```
if (gpsId2) {
    navigator.geolocation.clearWatch(gpsId2);
    gpsId2 = null;
}
```

Фиг. 7-8 Прекратяване на действието на метод watchPosition

Позицията на клиента се получава всеки път, когато се извика callback функция onSuccess. Тази функция получава като аргумент обект, чрез който може да извлече информация от GSP приемника:

- latitude – географска ширина.
- longitude – географска дължина.

- altitude – надморска височина в m (необходими са минимум 4 сателита).
- speed – скорост на предвижване в km/h.
- accuracy – точност в хоризонтална посока в m.

На Фиг. 7-9 е показано примерно съдържание на метод onSuccess.

```
function onSuccessGps(position) {
    var longitude = position.coords.longitude;
    var latitude = position.coords.latitude;
    var alttitude = Math.round(position.coords.altitude);
    var speed = position.coords.speed.toFixed(2);
    var accuracy = position.coords.accuracy.toFixed(2);
    // обработка на данните
}
```

Фиг. 7-9 Функция onSuccess

Ако се активира функция onError, позицията на клиента не може да бъде изчислена. Причината за това може да се определи от обект error (виж Фиг. 7-10).

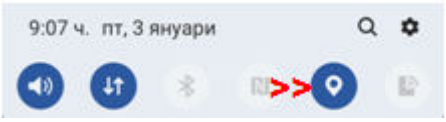
```
function onErrorGps(error) {
    let code = error.code;        // цифрово описание на грешката
    let info = error.messages;    // текстово описание на грешката
}
```

Фиг. 7-10 Функция onError

Възможните стойности за кода на грешката са дефинирани като константи:

- PositionError.POSITION_UNAVAILABLE – не е възможно получаването на позицията на клиента.
- PositionError.PERMISSION_DENIED – не е разрешено извличане на информация за позицията на клиента.
- PositionError.TIMEOUT – не е възможно получаването на позицията на клиента за времето зададено чрез параметър timeout.

За да получите достъп до GPS приемника трябва да разрешите получаване на местоположението (Settings -> Location -> Enable), както е показано на Фиг. 7-11.



Фиг. 7-11 Разрешаване на получаване на позицията

VI. Задача за изпълнение

Напишете хибридно мобилно приложение, което показва показанията от акселерометъра, компаса и GPS приемника. Чрез показанията на акселерометъра изчислете позицията на мобилното устройство. Да се разпознават следните ориентации:

- *„наклонен”.*
- *„към небето” – екранът сочи към небето.*
- *„към земята” – екранът сочи към земята.*

От данните на GPS приемника (географска ширина, географска дължина, височина и скорост на предвижване) и чрез използване на услуги за преобразуване на местоположение до адрес да се визуализира информация за адреса до който е най-близо мобилното устройство. Чрез услугата Google Static Map API да се визуализира GPS карта за региона в който е локализирано мобилното устройство.

6.1 Услуги за получаване на местоположение от GPS координати

Местоположението включва информация като име на държавата, име на града, име и номер на улицата. На базата на тази информация могат да се създадат услуги, които доставят персонализирано съдържание на своите клиенти. Например, ако имате услуга за реклама на ресторанти, всеки клиент може да получи контекстно-зависима и персонализирана реклама от ресторанта покрай който преминава. Ако наближава обед, информацията може да е свързана с това какво е обедното меню за деня, каква е цената на менюто и какви отстъпки се предлагат.

Съществуват множество услуги за получаване на местоположението при известни GSP координати. Тези координати са географска ширина и географска дължина. Най-известната от тези услуги е Google Geocoding. Услугата изисква задължителна регистрация, ключ за достъп, както и активиран план за заплащане чрез кредитна карта (дори да е безплатен ваучер за определена сума). Ще използваме две алтернативни на Google Geocoding услуги – Here и Geoapify.

6.1.1. Услуга Here

Услугата е достъпна след регистрация от следния адрес:

https://developer.here.com/

Услугата Here изисква регистрация чрез валиден email адрес. Изберете безплатен план за използване на услугата. Получете ключ за достъп до Here API чрез HTTPS заявки. При конкретният случай трябва да използвате Reverse Geocoding API. Заявката трябва да изпратите до следния адрес:

https://reverse.geocoder.ls.hereapi.com/6.2/reversegeocode.json

Параметрите, които трябва да изпратите към този ресурс, са следните:

- **prox** – низ, който съдържа GSP позицията на клиента във формат

  `"latitude, longitude".`

Може да се зададе и радиус спрямо тази точка с цел да се извлече информация за множество обекти, попадащи в този периметър, например

`"latitude, longitude, radius".`

За да получите своето местоположение, задайте радиус поне 2 пъти по-голям от точността на GPS приемника в метри (location accuracy), например при точност 10m, задайте радиус 20m.

- **mode** – режим на работа. Възможни са следните стойности:

  `retrieveAddresses`, `retriveAreas`, `retrieveLandmarks` и `retrieveAll`.

- **maxresults** – брой на резултатите (локации) в отговора.
- **gen** – коя версия на API да се използва.

- **apiKey -** ключ за достъп до услугата (43-байтов, Base64 кодиран низ).

Свалете програмния код, който показва как да използвате услугата чрез библиотека jQuery. За да използваме този код с Framework7 се налагат минимални промени в него: променете стойността на свойство `dataType` от jsonp на json и изтрийте стойността на свойство `jsonp` (виж Фиг. 7-12). Заявката се изпраща чрез метод `request`.

```
let proximity = `${latitude},${longitude},${radius}`;
app.request({
    url: 'https://reverse.geocoder.ls.hereapi.com/6.2/reversegeocode.json',
    type: 'GET',
    dataType: 'json',
    data: {
        prox: proximity,
        mode: 'retrieveAddresses',
        maxresults: '1',
        gen: '9',
        apiKey: 'Вашият API key'
    },
    success: function (data) {
        // получаване на името на държавата и града
        // от обект data
    },
    error: function (error) {
        // Чрез error.status може да получите кода на грешката,
        // например код 0 означава, че няма мрежова свързаност.
    }

});
```

Фиг. 7-12 Изпращане на заявка към услуга Here чрез Framework7

За да може да бъде обслужена заявката е необходимо да имате ключ за достъп до Reverse Geolocation API (API key). Този ключ се получава след регистрация и създаване на нов проект. При успешно получаване на местоположението се вика callback функция `success`. Обектът `data` съдържа необходимата информация в JSON формат.

Обектът data (виж Фиг. 7-13) трябва да бъде анализиран, за да извлечете желаната локация. Тази информацията се получава чрез свойство `Address`. За целта трябва да извлечете нулевия елемент на масива View. Така ще получите JSON обект от който ще получите стойността на нулевия елемент на свойство `Result`. Така ще имате достъп до обекти `Location` и `Address`. Пълната информация за адреса се съдържа в свойство `Label`. Чрез него можете да получите името на държавата, името на града, името и номера на улицата (при достатъчно точно определени GPS координатите). Програмния код, който извлича стойността на свойство Label е следния:

```
    let location = data.Response.View[0].Result[0].Location.Address.Label;
```

```
{
  "Response": {
    "MetaInfo": {
      "Timestamp": "2022-01-02T13:54:22.999+0000"
    },
    "View": [
      {
        "_type": "SearchResultsViewType",
        "ViewId": 0,
        "Result": [
          {
            "Relevance": 1,
            "Distance": 29.7,
            "MatchLevel": "houseNumber",
            "MatchQuality": {
              "Country": 1,
              "County": 1,
              "City": 1,
              "District": 1,
              "Street": [
                1
              ],
              "HouseNumber": 1,
              "PostalCode": 1
            },
            "MatchType": "pointAddress",
            "Location": {
              "LocationId": "NT_QzwwqbZCAWFp..qxX6djjA_0A",
              "LocationType": "address",
              "DisplayPosition": {
                "Latitude": 42.87532,
                "Longitude": 25.31613
              },
              "NavigationPosition": [
                {
                  "Latitude": 42.87533,
                  "Longitude": 25.31602
                }
              ],
              …,
              "Address": {
                "Label": "Улица Хаджи Димитър 4, 5306 Габрово, България",
                "Country": "BGR",
                "County": "Габрово",
                "City": "Габрово",
                "District": "Габрово",
                "Street": "Улица Хаджи Димитър",
                "HouseNumber": "4",
                "PostalCode": "5306",
              },
              "MapReference": {
                "ReferenceId": "809730557",
                "MapId": "UEAM19147",
                "MapVersion": "Q1/2021",
                 …
              }
            }
          }
        ]
      }
    ]
  }
}
```


Фиг. 7-13 Примерен резултат, получен при успешно изпълнена заявка към услуга Here

6.1.2. Услуга Geoapify

Услугата е достъпна след регистрация от следния адрес:

https://myprojects.geoapify.com/projects

Изберете безплатен план и създайте нов проект. Ще получите ключ (API key) с цел достъп до услугата. Може да тествате услугата Reverse Geocoding от следния адрес:

https://apidocs.geoapify.com/playground/geocoding#reverse

Програмният код, който изпраща заявка за локацията е показан на Фиг. 7-14.

```
app.request({
    url: 'https://api.geoapify.com/v1/geocode/reverse',
    type: 'GET',
    dataType: 'json',
    data: {
        lat: latitude,
        lon: longitude,
        apiKey: 'Вашият API key'
    },
    success: function (data) {
        // получаване на името на държавата и града
        // от обект data
    },
    error: function (error) {
        // Чрез error.status може да получите кода на грешката,
        // например код 0 означава, че няма мрежова свързаност.
    }
});
```

Фиг. 7-14 Изпращане на заявка към услуга Geoapify

Резултатът, който ще получите при успешно изпълнена заявка е показан на Фиг. 7-15. За да получите информация за локацията трябва да анализирате съдържанието на обекта `features`. Трябва да извлечете нулевият елемент на този масив. След това получете стойността на поле `formatted` от JSON обекта `properties`:

```
        let location = data.features[0].properties.formatted;
```

6.2 Потребителски интерфейс

Ще използваме само Framework7 Core графични компоненти, за да създадем потребителския интерфейс. Тъй като трябва да се визуализира малко като количество информация ще работим само с два изгледа:

- Main view – Показва информацията от акселерометъра, компаса и GPS приемника в отделни div контейнери.
- Left view – показва левия панел на приложението в който ще бъде менюто на приложението.

Ще използваме синя цветова схема и следното стилово форматиране:

- Светло син фон за всеки контейнер с информация от сензорите.
- Син цвят за текста, съдържащ информацията от сензорите.
- Червен цвят за текста, чрез който се описват възникнали грешки.
- Зелен цвят за текста за допълнителната информация – положение на телефона и адрес, получен след обратна геолокация.

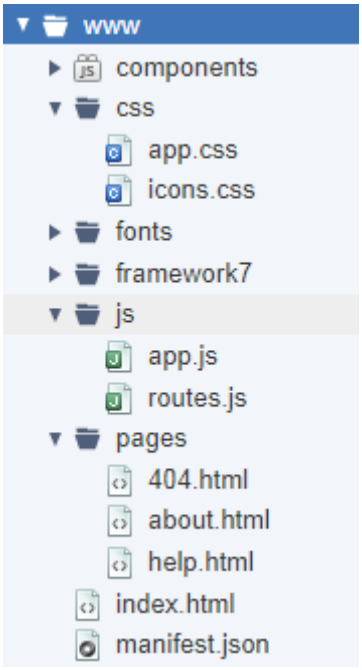
```
{
   "type":"FeatureCollection",
   "features":[
    {
     "type":"Feature",
     "properties":{
      "datasource":{
       "sourcename":"openstreetmap",
       "attribution":"© OpenStreetMap contributors",
       "license":"Open Database License",
       "url":"https://www.openstreetmap.org/copyright"
      },
      "housenumber":"40",
      "street":"Gen. Nikolov",
      "suburb":"кв. Тлъчници",
      "city":"Gabrovo",
      "county":"Gabrovo",
      "postcode":"5302",
      "country":"Bulgaria",
      "country_code":"bg",
      "lon":25.319757602267792,
      "lat":42.8893054,
      "distance":0,
      "result_type":"building",
      "formatted":"Gen. Nikolov 40, кв. Тлъчници, 5302 Gabrovo, Bulgaria",
       "address_line1":"Gen. Nikolov 40",
      "address_line2":"кв. Тлъчници, 5302 Gabrovo, Bulgaria",
      "category":"building.residential",
      "rank":{
       "popularity":3.5721445820882582
      },
      "place_id":"5136635ca2db51394059fe9364c2d4714540f00102f9012ae1e71300000000"
     },
     "geometry":{
      "type":"Point",
      "coordinates":[25.319757602267792,42.8893054]
     },
     "bbox":[25.319512,42.8890715,25.3200065,42.8895353]
    }
   ]
}
```


Фиг. 7-15 Примерен резултат при успешно изпълнена заявка към услуга Geoapify

6.3 Програмен код

Чрез развойната среда Monaca Cloud IDE създайте проект с име **Sensors**. Ще използваме само функциите на Framework7, за да реализираме логиката и потребителския интерфейс на приложението. Структура на проекта е показана на Фиг. 7-16. Предназначението на папките и файловете в тях е следното:

- **pages** – папка, която съдържа HTML файлове, необходими за менюто на приложението (about.html и help.html) и за визуализиране на информация при адресиране на несъществуващ ресурс (404.html).
- **js** – папка с JavaScrip код. Папката съдържа файл app.js чрез който се инициализира приложението и реализира базовата логика и файл routes.js чрез който се създава обект routes, необходим на маршрутизатора на Framework7.
- **css** – папка с CSS код. Съдържа файл icons.css, необходим за визуализиране на иконите с които работи приложението и файл app.css чрез който се задават специфични CSS правила.
- **Index.html** – индексен файл на приложението.



Фиг. 7-16 Структура на проект Sensors

6.3.1. Индексна страница index.html

Индексният файл съдържа стандартните за Framework7 Core проект секции от код. В тялото на етикет header трябва да заредите всички библиотеки и файлове с CSS код:

```
<link rel="stylesheet" href="framework7/css/framework7.min.css">
<link rel="stylesheet" href="css/icons.css">
<link rel="stylesheet" href="components/loader.css">
<link rel="stylesheet" href="css/app.css">
```

Програмният код, който създава левия панел на приложението с менюто е следния:

```
    <!-- Left panel with reveal effect when hidden -->
    <div class="panel panel-left panel-reveal">
      <div class="view view-left">
        <div class="page">
          <div class="navbar">
            <div class="navbar-inner sliding">
              <div class="title">Меню</div>
            </div>
          </div>
          <div class="page-content">
            <div class="block-title">Навигация в тялото на документа</div>
            <div class="list links-list">
              <ul>
                <li><a href="/about/" data-view=".view-main"
                            class="panel-close">Относно</a></li>
                <li><a href="/help/" data-view=".view-main"
                            class="panel-close">Помощ</a></li>
              </ul>
            </div>
          </div>
        </div>
      </div>
    </div>
```

Менюто има два елемента с имена “Относно” и “Помощ”. Пътищата до ресурсите, които се адресират са съответно “/about/” и “/help/”. Съдържанието на ресурсите ще се

визуализира в главния изглед на приложението, а не с самия панел. Това се задава със атрибут data-view на който е зададена стойност “.view-main”. Следователно, главният изглед трябва да е div контейнер с име на класа “view-main”:

```
<!-- Your main view, should have "view-main" class -->
<div class="view view-main safe-areas">
  <!-- Page, data-name contains page name which can be used in callbacks -->
  <div class="page" data-name="home">
    <!-- Top Navbar -->

    <!-- Toolbar-->

    <!-- Scrollable page content-->
    <div class="page-content">

      <div class="block">

        <div class="block sensor">
          <!—Данни от акселерометъра-->
        </div>

        <div class="block sensor">
          <!—Данни от компаса-->
        </div>

        <div class="block sensor">
          <!—Данни от GPS приемника-->
        </div>

      </div>
    </div>
  </div>
</div>
```

В основния изглед трябва да имаме div контейнер от клас “page”. Този контейнер съдържа навигационната лента (top navbar) и контейнера от клас “page-content” в който ще визуализираме информацията от сензорите. За целта са използвани три контейнера от клас “block sensor”.

При конкретният пример, навигационната лента съдържа иконата за менюто и заглавието на приложението – „Достъп до вградените сензори”. Програмният код, който създава навигационната лента е сленията:

```
      <div class="navbar">
        <div class="navbar-inner">
          <div class="left">
            <a href="#" class="link icon-only panel-open" data-panel="left">
              <i class="icon f7-icons ios-only">menu</i>
              <i class="icon material-icons md-only">menu</i>
            </a>
          </div>
          <div class="title sliding">Достъп до вградените сензори</div>
        </div>
      </div>
```

Програмният код, който създава контейнерите за информацията от сензорите е еднотипен. Използвани са 4 параграфа. Първият параграф от клас “title” съдържа наименованието на сензора. Вторият параграф съдържа данните, генерирани от сензора. Третият параграф съдържа допълнителната информация, извлечена въз основа на данните от сензора, например позиция на телефона в зависимост данните от акселерометъра. Последният параграф е за текст, който описва евентуална грешка,

възникнала по време на работа на сензора. Следва HTML кода за всеки един от сензорите:

```
<!—Данни от акселерометъра-->
<p class="title text-color-black">
  Акселерометър
</p>
<p class="accelerometer-data text-color-blue"></p>
<p class="device-orientation text-color-green"></p>
<p class="accelerometer-error text-color-red"></p>
<!—Данни от компаса-->
<p class="title text-color-black">
  Компас
</p>
<p class="compass-data text-color-blue"></p>
<p class="compass-error text-color-red"></p>
<!—Данни от GPS приемника-->
<p class="title text-color-black">
  GPS сензор
</p>
<p class="gps-data text-color-blue"></p>
<p class="gps-location text-color-green"></p>
<p class="gps-error text-color-red"></p>
<div class="iframe-container">
  <iframe class="google_map" src=""></iframe>
</div>
```

Тъй като по условието на задачата трябва да се покаже GPS карта с текущата локация на мобилното устройства, в секцията за GPS приемника е предвиден допълнителен div контейнер от клас “iframe-container” в който ще се визуализира картата. Ще използваме услугата Google Static Maps, която е безплатна. Тя изисква GPS картата да се интегрира в тялото на етикет iframe. Атрибут src ще бъде инициализират програмно след получаване на GPS координатите.

Цветовете за текста се задава чрез Framework7 атрибути “text-color-xxx”, където xxx е цвета, който искаме да използваме. По този начин не се налага писане на съответния CSS код.

В края на етикет body остана да заредим необходимите библиотеки и файлове с JavaScript код:

```
<!-- Framework7 library -->
<script src="framework7/js/framework7.min.js"></script>

<!-- App routes -->
<script src="js/routes.js"></script>

<!-- Your custom app scripts -->
<script src="js/app.js"></script>

<!-- Monaca JS library -->
<script src="components/loader.js"></script>
```

6.3.2. HTML страници

Страниците с HTML код, необходими за функциониране на приложението са само три:

- Файл **about.html** – съдържа кода, който се визуализира при избор на път “/about/”.
- Файл **help.html** – съдържа кода, който се визуализира при избор на път “/help/”.
- Файл **404.html** – съдържа кода, който се визуализира при избор на път, който не е описан в routes.js.

Следва съдържанието на файл about.html:

```
<div class="page" data-name="about">
  <div class="navbar">
    <div class="navbar-inner sliding">
      <div class="left">
        <a href="#" class="link back">
          <i class="icon icon-back"></i>
          <span class="ios-only">Назад</span>
        </a>
      </div>
      <div class="title">Относно</div>
    </div>
  </div>
  <div class="page-content">
    <div class="block-title">Приложението</div>
    <div class="block block-strong">
      <p>Моля, въведете тук информация за приложението.</p>
    </div>
    <div class="block-title">Автора</div>
    <div class="block">
      <p>Моля, въведете тук информация за автора.<p>
    </div>
  </div>
</div>
```

Цялото съдържание трябва да е в div контейнер от клас “page”. При превключване на контекста от маршрутизатора, този контейнер ще замени съдържанието на съответния контейнер от клас “page”, дефиниран във файл index.html. При натискане на бутон back (Назад) съдържанието на основния изглед ще се възстанови. Това, което искате да се визуализира при избора на “Относно” от менюто трябва да бъде в div контейнер от клас “page-content”.

Следва съдържанието на файл help.html:

```
<div class="page">
  <div class="navbar">
    <div class="navbar-inner sliding">
      <div class="left">
        <a href="#" class="link back">
          <i class="icon icon-back"></i>
          <span class="ios-only">Назад</span>
        </a>
      </div>
      <div class="title">Помощ</div>
    </div>
  </div>
  <div class="page-content">
    <div class="block block-strong">
      <p>Моля, въведете тук информация как се работи с приложението.</p>
    </div>
  </div>
</div>
```

Кодът на файл 404.html е следния:

```
<div class="page">
  <div class="navbar">
    <div class="navbar-inner sliding">
      <div class="left">
        <a href="#" class="link back">
          <i class="icon icon-back"></i>
          <span class="ios-only">Назад</span>
        </a>
      </div>
      <div class="title">404</div>
    </div>
  </div>
  <div class="page-content">
    <div class="block block-strong">
      <p>Съжаляваме, заявената от вас информация не бе намерена.</p>
    </div>
  </div>
</div>
```

6.3.3. Маршрутизация

Описанието на всички маршрути са описани чрез обект routes във файл routes.js:

```
routes = [
  {
    path: '/',
    url: './index.html',
  },
  // Страница about
  {
    path: '/about/',
    url: './pages/about.html',
  },
  // Страница help
  {
    path: '/help/',
    url: './pages/help.html',
  },
  // Default route (404 page). MUST BE THE LAST
  {
    path: '(.*)',
    url: './pages/404.html',
  },
];
```

Не забравяйте да зададете маршрута за страница 404 като последен JSON обект в масива routes.

6.3.4. Инициализация

Инициализацията на приложението се реализира чрез програмния код от файл app.js. Създаването на обекти $$ и app с цел достъп до DOM7 и Framework7 Core са стандартни и няма да бъдат разглеждани (виж Глава „Програмна рамка Framework7”, секция „Инициализация на приложението”). Създайте обекти за всеки един от използваните изгледи:

```
// Init/Create main view
var mainView = app.views.create('.view-main', {
  url: '/'
});
// Init/Create left panel view
var leftView = app.views.create('.view-left', {
  url: '/'
});
```

Прехванете събитие “deviceready”, за да разберете кода е заредена в паметта библиотеката Cordova. От тялото на callback функцията, която обработва това събитие вмъкнете следния програмен код:

```
// Wait Cordova.js to be ready
$$(document).on('deviceready', function () {
  // Скриване на iframe етикета
  $$('.google_map').hide();

  startAccelerometer();
  startCompass();
  startGps();
});
```

Първо, използвайте метод `hide`, за да скриете съдържанието (рамката) на `iframe` етикета в който ще се визуализира GPS картата. След това, последователно извикайте методите чрез които се получават данни от акселерометъра, компаса и GPS приемника.

6.3.4. Програмен код за акселерометъра

Програмният код, който стартира четенето на данни от акселерометъра и изчислява позицията на мобилното устройство в пространството е следния:

```
const Z_THRESHOLD_VALUE = 9.5;
var accelerometerId = null;
var lastPosition = null;
// ---------------------------------------------------------------------------
function startAccelerometer() {
  var options = {
    frequency: 250
  };
  accelerometerId = navigator.accelerometer.watchAcceleration(success, error, options);
}
// ---------------------------------------------------------------------------
function success(acceleration) {
  var x = acceleration.x.toFixed(3);
  var y = acceleration.y.toFixed(3);
  var z = acceleration.z.toFixed(3);
  $$(".accelerometer-data").html(`X acceleration: ${x} m/s<sup>2</sup><br/>
   Y acceleration: ${y} m/s<sup>2</sup><br/>Z acceleration: ${z} m/s<sup>2</sup>`);
  var accelerationSign = Math.sign(z);
  if (Math.abs(z) < Z_THRESHOLD_VALUE) {
    lastPosition = 'наклонен';
  }
  else {
    if (lastPosition !== "към небето" && accelerationSign > 0) {
      lastPosition = "към небето";
    }
    else if (lastPosition !== "към земята" && accelerationSign < 0) {
      lastPosition = "към земята";
    }
  }
  $$('.device-orientation').text('Ориентация на телефона: ' + lastPosition);
}
// ---------------------------------------------------------------------------
function error(error) {
  let info;
  let message = error.message;
  if (message === undefined) {
    info = 'Липсва акселерометър!';
  }
  else {
    info = "Грешка при достъп до акселерометъра:<br/>" + message;
    stopAccelerometer();
  }
  $$(".accelerometer-error").html(info);
}
```

```
function stopAccelerometer() {
  if (accelerometerId) {
    navigator.accelerometer.clearWatch();
  }
}
```

Метод startAccelerometer използва метод watchAcceleration, за да се получават данни от сензора за ускорение на всеки 250 ms (свойство frequency от обект options). При успешно получени данни (ускорение по трите оси X, Y и Z) се вика callback функция success. При грешка се вика callback функция error.

Функция success получава като аргумент обекта acceleration. Чрез свойства x, y и z получаваме стойностите за ускорението. Закръгляваме данните за ускорението до третия знак след десетичната точка чрез метод toFixed. Следва асемблиране на информацията за ускорението по трите оси и показването ѝ в div контейнера от клас “acceletometer-data”.

Продължаваме с определяне на ориентацията на мобилното устройство. Тъй като трябва да разберем дали устройството е с екрана към небето или към земята, то трябва да анализираме ускорението по оста Z. За целта дефинираме константа с име Z_THRESHOLD_VALUE със стойност малко по-малка от земното ускорение - 9.5 m/s^2. Ще

ни трябва и знака на ускорението. Получаваме го чрез метод sign от библиотека Math. Приемаме, че ако ускорението по оста Z е по-малко от така дефинираната прагова стойност, то устройството е наклонено. В противен случай устройството е с екрана „към небето” или „към земята”. За да разберем дали ориентацията е „към небето” или „към земята” дефинираме променливата lastPosition чрез която ще помним последната ориентация на устройството. Ако последната ориентация не е „към небето” и ускорението е с положителен знак, то новата ориентация е „към небето”. Ако последната ориентация не е „към земята” и ускорението е с отрицателен знак, то новата ориентация е „към земята”.

6.3.5. Програмен код за компаса

Ще стартираме компаса да измерва посоката спрямо северния магнитен полюс през интервал от 1 секунда. За целта ще използваме метод watchHeading. Програмният код е следния:

```
var compassId = null;
function startCompass() {
  var options = { frequency: 1000 };
  compassId = navigator.compass.watchHeading(successCompass, errorCompass, options);
}
var successCompass = function (heading) {
  var value = Math.abs(heading.magneticHeading);
  value = value.toFixed(2);
  $$(".compass-data").html(`Посока: ${value}&deg;`);
};
function errorCompass(error) {
  let info;
  let message = error.message;
  if (message === undefined) {
    info = 'Липсва компас!';
  }
  else {
    info = "Грешка при достъп до компаса:<br/>" + message;
    stopCompass();
  }
  $$(".compass-error").html(info);
}
```

```
function stopCompass() {
  if (compassId) {
    navigator.compass.clearWatch(compassId);
    compassId = null;
  }
}
```

При успешно получени данни от компаса (callback функция successCompass) стойността на посоката в градуси се извлича от свойството magneticHeading. Трябва да имате предвид, че при някои мобилни устройства не се използва магнитен компас с цел да се намали цената.

6.3.5. Програмен код за GPS приемника

Ще инициализираме четенето на данните от GPS приемника да се реализира през 4 секунди. Ако за над 30 секунди няма данни от GPS приемника – да се използва последната валидна локация на мобилното устройство от кеша. Програмният код за стартиране на четенето от GPS приемника се реализира чрез метод startGps:

```
var gpsId = null;
var firstTime = true;
const PROXIMITY_RADIUS = 12;
const MAP_ZOOM = 17;
function startGps() {
  if (navigator.geolocation) {
    var options = { enableHighAccuracy: true, timeout: 4000, maximumAge: 30000 };
    gpsId = navigator.geolocation.watchPosition(successGps, errorGps, options);
    $$(".gps-data").html("Изчакване за връзка с GPS приемника ...");
  }
  else {
    $$(".gps-error").html("Невъзможно е получаването на позицията!");
  }
}
```

Първо, трябва да проверим наличието на GPS приемник. Това се реализира чрез стойността на свойство geolocation от обекта navigator. Чрез функция watchPosition ще активираме четене на данните от GPS приемника в цикъл през 4 секунди. При получаване на нови данни се вика callback функция successGps:

```
function successGps(position) {
  var lon = position.coords.longitude;
  var lat = position.coords.latitude;
  var alt = Math.round(position.coords.altitude);
  var speed = position.coords.speed;
  if (speed != null) {
    speed = speed.toFixed(2);
  }
  var accuracy = position.coords.accuracy.toFixed(2);

  $$(".gps-data").html(`Географска дължина: ${lon}<br/>
                        Географска ширина: ${lat}<br/>
                        Височина: ${alt} m<br/>
                        Скорост: ${speed} km/h<br/>
                        Точност: ${accuracy} m`);
  if (firstTime && accuracy < PROXIMITY_RADIUS) {
    //getLocation1(lat, lon, PROXIMITY_RADIUS*2);
    getLocation2(lat, lon);
    firstTime = false;
  }
}
```

Информацията която ще визуализираме е следната:

- Географска дължина (longitude) в градуси.
- Географска ширина (latitude) в градуси;
- Надморска височина (altitude) в метри.
- Скорост на движение (speed) в m/s.
- Точност на измерването (accuracy) в метри.

Трябва да се отчете, че стойността за скоростта е вярна само при видими 4 и повече сателити. Колкото е по-малка стойността за точност на локализация (accuracy), толкова по-точно е определена локацията. Приема се, че при accuracy<12m резултатите са задоволителни. Информацията от GPS приемника се записва динамично в div контейнер от клас “gps-data”. Ако точността е под 12m (константа PROXIMITY_RADIUS) *еднократно* се извлича адреса, който съответства на тези координати. За целта може да се използват две функции:

- getLocationHERE – намира адреса чрез услуга Here.

- getLocationGEOAPIFY - намира адреса чрез услуга Geoapify.

Следва програмния код на функция getLocationHERE:

```
function getLocationHERE(latitude, longitude, radius) {
  let proximity = `${latitude},${longitude},${radius}`;
  app.request({
    url: 'https://reverse.geocoder.ls.hereapi.com/6.2/reversegeocode.json',
    type: 'GET',
    dataType: 'json',
    data: {
      prox: proximity,
      mode: 'retrieveAddresses',
      maxresults: '1',
      gen: '9',
      apiKey: 'Вашият API ключ'
    },
    success: function (data) {
      let location = data.Response.View[0].Result[0].Location.Address.Label;
      $$(".gps-location").html(`Местоположение:<br/>${location}`);
      showGpsMap(latitude, longitude, MAP_ZOOM);
    },
    error: function (error) {
      let status;
      if (error.status === 0) {
        status = 'Моля, разрешете достъпа до мрежата.';
      }
      else {
        status = `Невъзможно получаване на местоположението:<br/>
                                                ${error.statusText}`;
      }
      $$(".gps-error").html(status);
    }
  });

}
```

Функцията изисква три аргумента: географска ширина, географска дължина и радиус в метри около изчислената локация. В така дефинираната окръжност се търсят обекти (сгради). При конкретният пример стойността на radius е два пъти по-голяма от стойността на константа PROXIMITY_RADIUS. По този начин е по-вероятно получаване на адреса на сградата, дори когато изчислената локация не попада точно в рамките на контура на сградата.

Ще използваме метод request, за да изпратим заявка към услуга Here. Не забравяйте да зададете коректна стойност на свойство apiKey – вашият ключ за достъп до Reverse Geo-location API. При успешно изпълнение на заявката се вика функция success. Следва извличане на адреса (location) от JSON отговора. Тази информация се вмъква динамично в div контейнер от клас “gps-location”.

Чрез метод showGpsMap се получава GPS карта, центрирана спрямо получените GPS координати:

```
function showGpsMap(lat, lon, zoom) {
  $$('.google_map').attr('src',
    `https://maps.google.com/?q=${lat},${lon}&z=${zoom}&output=embed`);
  $$('.google-map').show();
}
```

Параметърът zoom определя какъв да бъде мащаба на картата. Колкото стойността на този параметър е по-голяма, толкова по-подробна ще бъде картата. За да получим GPS

картата е необходимо да адресираме ресурс https://maps.google.com. Параметрите, които се предават на ресурса са следните:

- q – стойностите на географската ширина и географската дължина, разделени със запетая.
- z – стойност за мащаба (zoom).
- output – тъй като GPS картата се интегрира в iframe етикет трябва да се зададе стойност embed.

Контейнерът за GPS картата е div етикет от клас “google-map”. По подразбиране той е скрит. Налага се да разрешим неговата визуализация чрез метод show.

Ако искате да получите адреса чрез услуга Geoapify използвайте програмния код от функция getLocationGEOAPIFY:

```
function getLocationGEOAPIFY(latitude, longitude) {
  app.request({
    url: 'https://api.geoapify.com/v1/geocode/reverse',
    type: 'GET',
    dataType: 'json',
    data: {
      lat: latitude,
      lon: longitude,
      apiKey: 'Вашият API ключ'
    },
    success: function (result) {
      if (result.features.length) {
        let location = (result.features[0].properties.formatted);
        $$(".gps-location").html(`Местоположение:<br/>${location}`);
        showGpsMap(latitude, longitude, MAP_ZOOM);
      } else {
        $$(".gps-location").html('Не е намерен адрес!');
      }
    },
    error: function (error) {
      // обработка на грешки …
    }
  });
}
```

При получаване на грешка при работа с GPS приемника се вика callback функция errorGps:

```
function onErrorGps(error) {
  let info = error.message;
  $$(".gps-error").html(`GPS грешка: ${info}<br/>
                Проверете дали е резрешено получаване на местоположението!`);
  stopGps();
}
function stopGps(id) {
  if (gpsId) {
    navigator.geolocation.clearWatch(gpsId);
    gpsId = null;
  }
}
```

Основната причина за евентуална грешка е забраната за получаване на геолокация. В този случай приложението уведомява потребителя и преустановява получаването на данни от GPS приемника. Това се реализира чрез функция stopGps (clearWatch).

6.3.6. Стилово форматиране

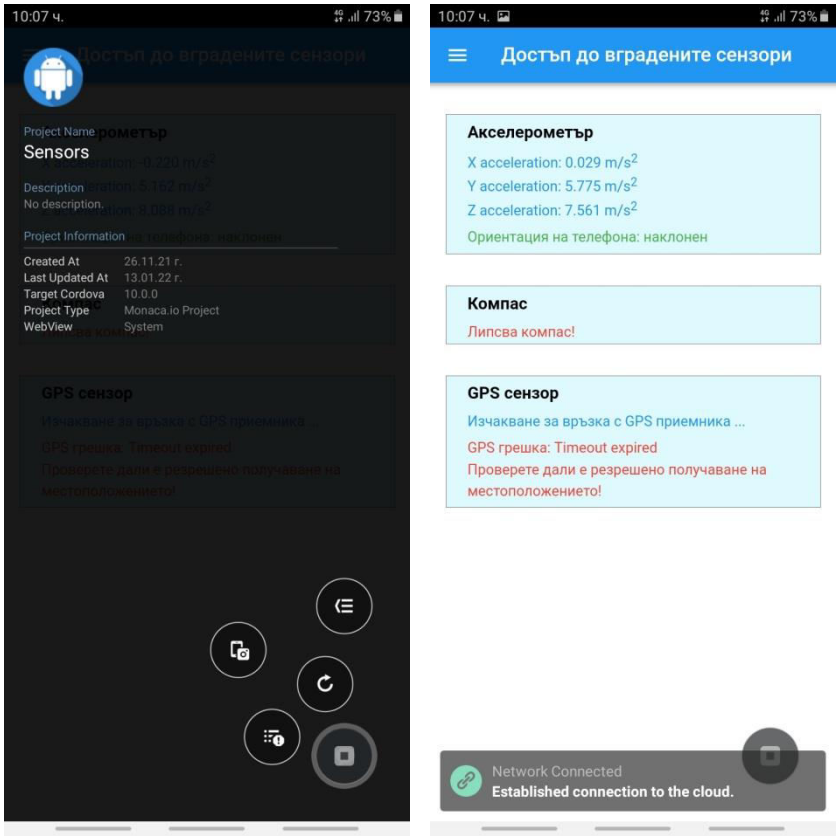
Необходимото стилово форматиране за приложението се реализира чрез CSS правилата от файл app.css:

```
body {
  padding: 0;
  margin: 0;
  font-family: system-ui;
}
p {
  font-size: 1.8vmax;
  padding: 0;
  margin: 5px;
}
p.title {
  font-size: 2.1vmax;
  font-weight: 700;
  padding-top: 5px;
  padding-bottom: 0;
```

```
div.sensor {
  border: 1px solid #808080;
  background: #defaff;
  padding-top: 5px;
  padding-bottom:5px;
  margin: 0;
}
.iframe-container {
  display: table-row;
  height: 100%;
  overflow: hidden;
}
.iframe-container iframe {
  width: 100%;
  height: 100%;
  border: 1px solid #606060;
  margin: 10px 0 20px 0;
  padding: 0;
  display: block;
}
```

6.4 Тестване на приложението

След като въведете програмния код и отстраните всички синтактични грешки можете да тествате приложението. Не може да използвате вградения в Monaca Cloud IDE симулатор, тъй като чрез него няма как да се реализира достъп до сензорите. Затова използвайте Monaca Debugger. Стартирайте Monaca Debugger и от листа с активни приложения изберете Sensors. Изчакайте да се синхронизира съдържанието на приложението. На Фиг. 7-17 са показани два екрана, получени при тестване на приложението. Първият екран (a) показва информация за приложението, а вторият (b) – основния изглед на приложението. При конкретният пример в мобилното устройство няма вграден компас, а получаването на локацията е забранена.



a) b)

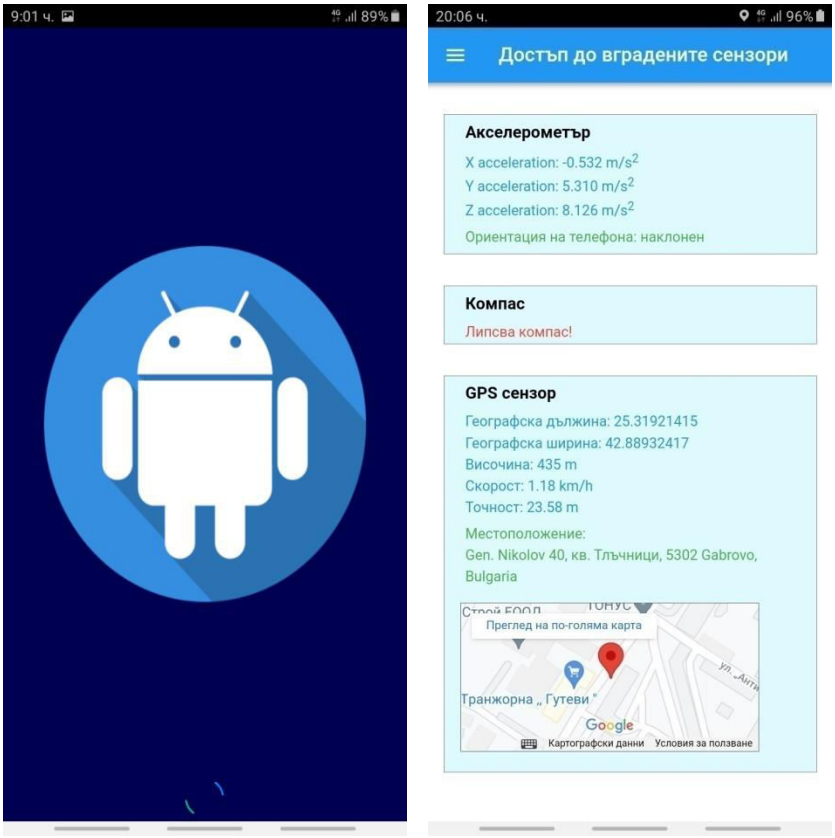
Фиг. 7-17 Тестване на приложението с Monaca Debugger:
a) Информация за приложението; b) Основен изглед на приложението.

6.5 Изграждане на приложението

След като се уверите, че приложението работи както се очаква може да получите инсталационен файл. Преди това задайте информацията, без която изграждането на приложението не е възможно. Трябва да изберете за коя операционна система ще компилирате, името на приложението, версията на SDK и др. Не забравяйте да създадете свои икони и начален екран. За повече информация вижте Глава 4 „Развойна среда Monaca", секция „Изграждане на приложението".

От менюто на Monaca Cloud IDE изберете "Build", а след това изграждане за ОС Android. Изчакайте докато изграждането завърши. Ще получите инсталационен APK файл, който трябва да прехвърлите на своето мобилно устройство. Това може да стане чрез сканиране на генерирания QR код или чрез прехвърляне на APK файла чрез USB или Bluetooth.

На Фиг. 7-18 са показани два екрана, които показват работата на приложението на реално устройство – начален екран (a) и основния изглед на приложението (b).



a) b)

Фиг. 7-18 Резултат, получен след стартиране на приложението:
a) Начален екран; b) Основен изглед на приложението.

Заключение

В тази глава се запознахте с предназначението и начина на работа на основните програмно достъпни сензори, вградени в съвременните мобилни устройства: акселерометър, жироскоп, магнитометър, барометър и GPS приемник. Научихте се как програмно може да четете данните от акселерометъра, компаса и GPS приемника. Запознахте се с програмния код на мобилно приложение, което демонстрира работа с тези сензори, както и използване на услуги за преобразуване на GPS координати до пощенски адрес (Here и Geoapify). Въз основа на тази информация ще можете да разработвате собствени проекти, които използват данните от сензорите с цел създаване на приложения, базирани на местоположението на потребителите или проекти със специфичен потребителски интерфейс.

Глава 8

Достъп до бази данни

I. Въведение

Голяма част от мобилните услуги имат нужда от място за съхранение на специфични за услугата данни. Управлението на достъпа до тези данни (създаване на нови данни, четене, обновяване и изтриване на данни) най-често се реализира чрез *сървъри за управление на бази данни*. Хостването на базата данни може да бъде на предвиден за целта сървър, на няколко сървъра или в облачна инфраструктура. При последният вариант отпада необходимостта от администриране на сървъра и осигуряване на сигурността на данните.

На практика, при повечето съвременни мобилните услуги най-често използваните бази данни са нерелационни (NoSQL). Терминът NoSQL е съкращение от „Not only SQL". NoSQL базите данни са съвременна технология, която намира все по-широко приложение, тъй като добре се интегрира с идеите за BigData и Internet Of Things (IoT). Пионерите в теза технология са Google със своята BigTable и Amazon с Dynamo. На базата на идеите, заложени в BigTable и Dynamo, се създава базата данни Cassandra, която се използва от Facebook. От архитектурата на Google BigTable е инспирирана разработката на Hadoop, а от архитектурата на Dynamo - Riak.

Основните предимства на NoSQL базите данни са:

- Опростен дизайн, не изискват прилагането на строга схема за данните, която важи за цялата база.
- По-лесна реализацията на *хоризонтално* мащабиране.
- Възможност за работа в реално време с много големи масиви от данни, благодарение на ефективния начин за извличане и филтриране на данни чрез MapReduce.
- По-лесна реализация на *силно разпределени* бази данни.
- По-лесна реализация на синхронизация между отделните сървъри за управление на базите данни при използване на разпределена архитектура.

Основните разновидности на NoSQL базите данни са следните:

1) Документен тип (MongoDB, CouchDB, Couchbase).
2) Колонен тип (Cassandra, HBase).
3) Ключ-стойност (BigTable, Redis, Riak);
4) Граф-базирани (HyperGraphDB, Neo4j).

Документен тип бази данни

При тези бази данни идеята е данните да се съхраняват в *документи*, които могат да се обединяват в *колекции*. Форматът на описание на данните в документите най-често е JavaScript Object Notation (JSON) или Binary JSON (BSON).

Ключ-стойност (Key-value) бази данни

Записът на данните при този тип бази данни е във формат ключ-стойност. Най-често данните се описват като JSON обект. Това означава, че тези бази данни не използват определена схема, която дефинира начина на описание на данните. Това от своя страна води до по-ефективно описание на информацията, тъй като не се налага еднотипните елементи от една база данни да се описват с едни и същи атрибути (някои могат да се пропуснат, ако не са необходими, а не да им се назначава стойност по подразбиране).

Колонен тип бази данни

При колонните бази данни информацията е организирана в колони, които се адресират чрез ключ (уникално име), стойност и времеви маркер (timestamp). Това наподобява key-value базите данни, с тази разлика, че тук се използва и времеви маркер. Той се използва се цел разграничаване на валидните от остарелите данни. Това води до по-лесна синхронизация на информацията при използване на бази с разпределена архитектура.

Базирани на графи бази данни

С цел бързо извличане на информация от базата данни е необходимо данните в базата да бъдат моделирани чрез структури като дървета или графи. Например, повечето бази данни от документен тип използват B или B+ дърво за моделиране на данните. Когато релациите на елементите от базата са неопределен брой, по-добрият вариант е те да се моделират чрез граф.

Когато информацията в базата данни е над определен обем, става невъзможно достъпването ѝ в реално време, ако данните са съсредоточени в една база данни (един нод). В този случай се налага изграждане на клъстер от сървъри, които обслужват заявките на клиентите. Такива бази данни е прието да се наричат разпределени бази данни. При тях се намалява значително времето за обслужване на заявките на клиентите, но на преден план остава друг проблем – гарантиране на съгласуваност на информацията. Ако един клиент обнови информация в базата данни на един от сървърите, клиентите, които се обслужват от другите сървъри и адресират същата информация, трябва да получат нейната последната стойност. За да е възможно това се налага поддържането на синхронизация на информацията в отделните нодове (*съгласуваност*). Най-лесният, но и най-бавен начин за гарантиране на съгласуваност е клиентите да изчакат докато информацията от която се интересуват се обновява. Този начин обаче не гарантира достъпност до информацията в реално време. За една разпределена база данни не може да се гарантира съгласуваност (consistency), наличност (availability) и разпределяне на части (работа в разпределена среда) в един и същи момент от време. Тази теорема е дефинирана Ерик Брюър и за кратко се нарича CAP (Consistency, Availability, and Partition tolerance) теорема.

Консистентност: Всеки клиент, независимо чрез кой нод на базата данни се обслужва, получава едни и същи данни както останалите клиенти, независимо от протичащите конкурентни обновления (всички реплики на документите имат една и съща версия).

Наличност: Дори да има нефункциониращи нодове, клиентите имат възможност да четат и записват информация.

Разпределяне на части: Базата данни може да бъде разпределена и да се обслужва от множество сървъри. Системата остава работоспособна при наличие на проблеми при някой от сървърите.

Всяка база данни в даден момент от време гарантира само два от тези параметри:

- AP бази данни: CouchDB, Cassandra, SimpleDB, Riak, Dynamo.
- CP бази данни: Couchbase, MongoDB, Big Table, Hypertable, Hbase, Redis.
- CA бази данни: Всички релационни бази данни, например MySQL

Ако достъпността е приоритет пред съгласуваността, системата трябва да предостави възможност за запис на информация в някой от нодовете на системата и след това да има механизъм със забавяне във времето, който да гарантира съгласуваност на информацията. В този случай се казва, че базата поддържа евентуална съгласуваност. Такава база данни например е CouchDB (IBM Cloudant). Вместо да се блокира достъпа до базата по време на обновяване на информация, CouchDB използва Multi-Version Concurrency Control (MVCC), за да се обслужат конкурентните заявки до базата данни. Идеята е за всеки документ да се поддържа версия, която се отразява автоматично чрез стойността на поле с име „_rev". Обновяването на съдържанието на документ води автоматично до създаване на нов документ, който съдържа най-новата информация, но старият документ не се изтрива. Тази стратегия позволява обслужване на конкурентните заявки без значимо забавяне на отговорите. Нека клиент А чете данни от документ Х. В същият момент клиент В прави опит за запис в документ Х. Записът започва веднага, без да се чака транзакцията с клиент А да завърши, тъй като за клиент В се създава копие на документа с нова версия. Ако клиент С желае да прочете данни от същия документ, той автоматично ще бъде насочен да прочете съдържанието на най-новата версия на документа.

II. Примерна мобилна услуга с достъп до база данни

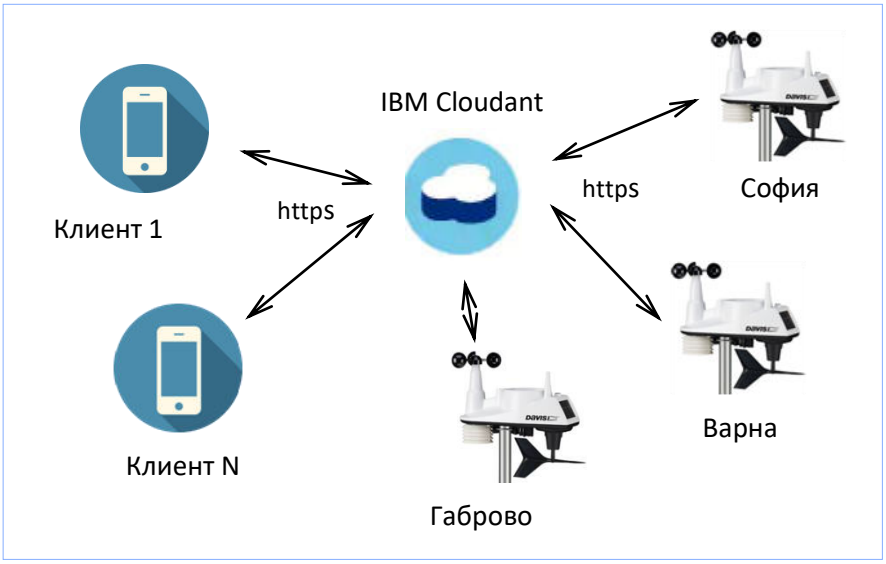
Ще разработим мобилно хибридно приложение, което визуализира информация за времето в избран от клиента град, например: температура, влажност на въздуха, атмосферно налягане и UV индекс. Нека информацията за времето се получава от специализиран хардуер (метеорологична станция), който има възможност за изпращане на информацията за времето през определен интервал от време посредством протокол HTTPS.

2.1 Избор на база данни

При този пример няма нужда от използване на релационна база данни. Гарантирането на съгласуваност не е от първостепенно значение, тъй като информацията от базовите станции рядко се променя. Ще използваме документен тип база данни, като за всеки град ще се създаде документ, който съдържа информация за времето. Можем да избегнем необходимостта от сървър, който да дава връзката между клиентите, станциите и базата данни. За целта ще използваме облачно базирана база данни, достъпът до която се реализира чрез протокол HTTPS, който и станциите поддържат. Подходящ избор е MongoDB Azure и IBM Cloudant, която е Web варианта на Apache CouchDB. Ще използваме IBM Cloudant, тъй като тя е подходяща при мобилни услуги при които се чете много по-често от документите, отколкото те се обновяват. Тази база данни позволява създаване на безсървърни (serverless) мобилни услуги. Това означава, че няма да е необходимо да се пише какъвто и да е back-end код.

Архитектурата на услугата е показана на Фиг. 8-1. Клиентите на услугата разполагат с мобилно устройство и мобилно приложение чрез което достъпват базата данни чрез стандартни HTTPS заявки. Приложението прави запитване до базата за времето в избран град. При всяка заявка се връща съдържанието на документ, който отговаря на избрания град. Данните в тези документи се обновява динамично от метеорологични станции само при засичане на изменение на един или няколко от следените параметри. Това

изменение трябва да е по-голямо от предварително зададена прагова стойност. Мобилните клиенти имат право само да четат от базата данни, а метеорологичните станции – да четат и да записват.

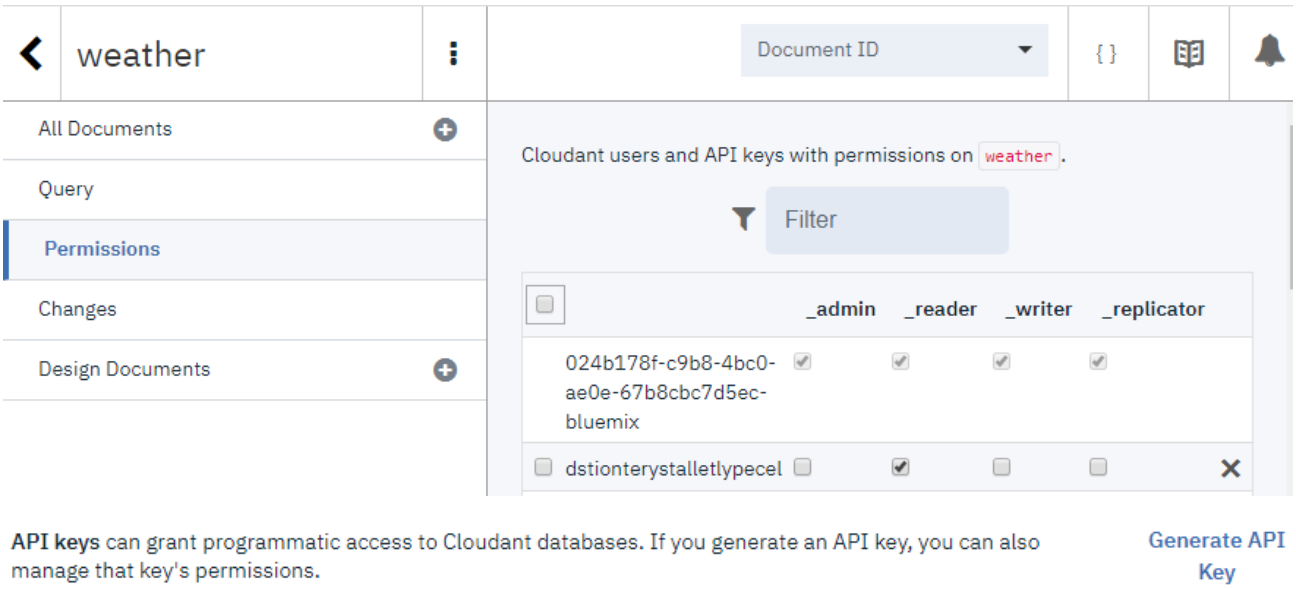


Фиг. 8-1 Архитектура на мобилната услуга

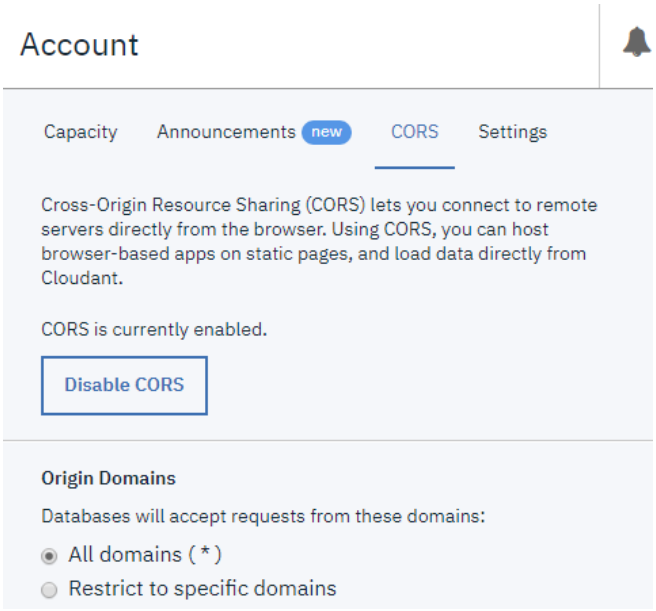
За да използвате базата данни IBM Cloudant е необходимо да се регистрирате като клиент на IBM Cloud. Реализирайте регистрация от адрес:

https://www.ibm.com/cloud/cloudant

Регистрацията изисква данни за банкова карта, въпреки, че може да изберете безплатен период на използване на услугата. Активирайте зареждане на база данни IBM Cloudant като ресурс във вашия профил. След логване към IBM Cloud изберете вече активираната услуга, а след това Dashboard. Ще получите достъп до Web интерфейса на услугата. Създайте база данни с име weather (non partitioning database). Създайте ключове (API keys) за достъп до базата данни. Ключът за клиентите трябва да позволява само четене от базата данни, а ключа за метеорологичните станциите да позволява и четене и запис (виж Фиг. 8-2).



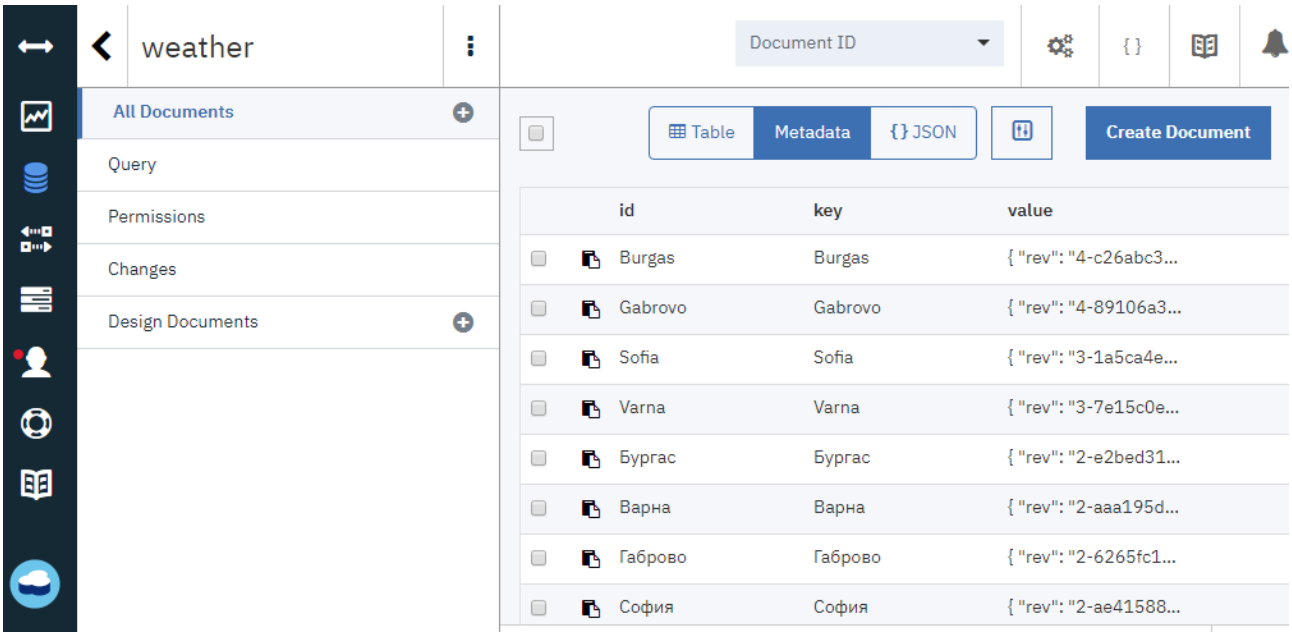
Фиг. 8-2 Създайте ключове за достъп до базата данни



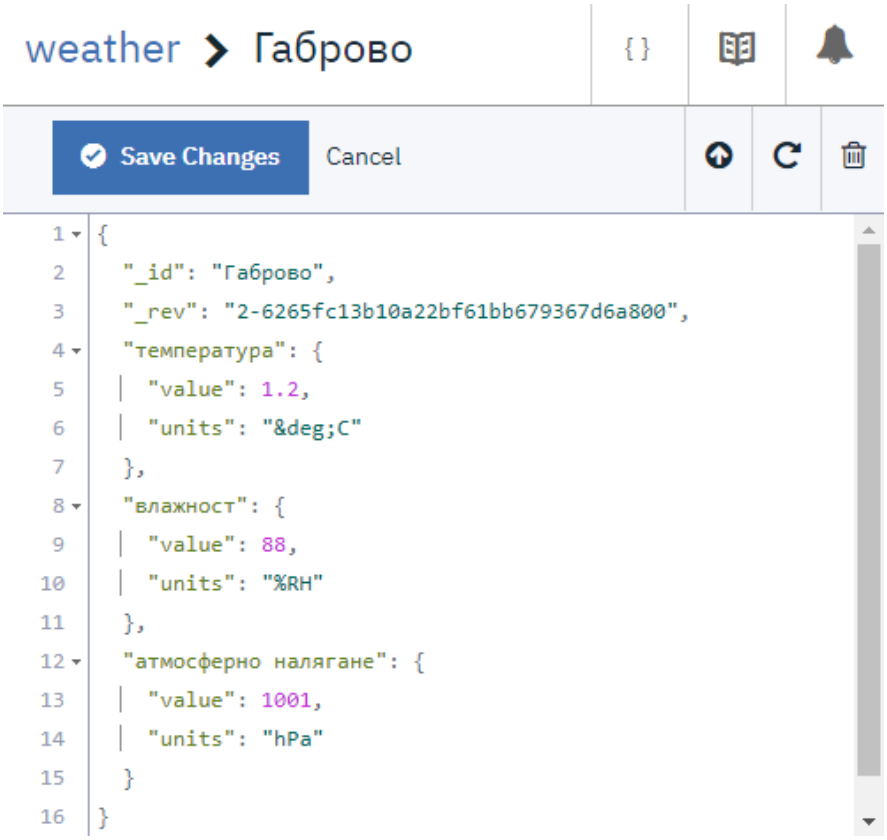
Фиг. 8-3 Настройка на достъпа до базата данни

Разрешете директния достъп (serverless) до базата данни чрез HTTPS заявки. За целта от Account изберете CORS (Cross-Origin Resource Sharing) -> Enable CORS и разрешете достъпа до базата данни от кой да е хост (виж Фиг. 8-3).

Създайте необходимите документи към базата данни weather. На фиг. 8-4 е показано съдържанието на база данни weather. В случая за всеки град има два документа – един на български език и един на английски език.



Фиг. 8-4 Документи в база данни weather



Фиг. 8-5 Съдържание на документ „Габрово”

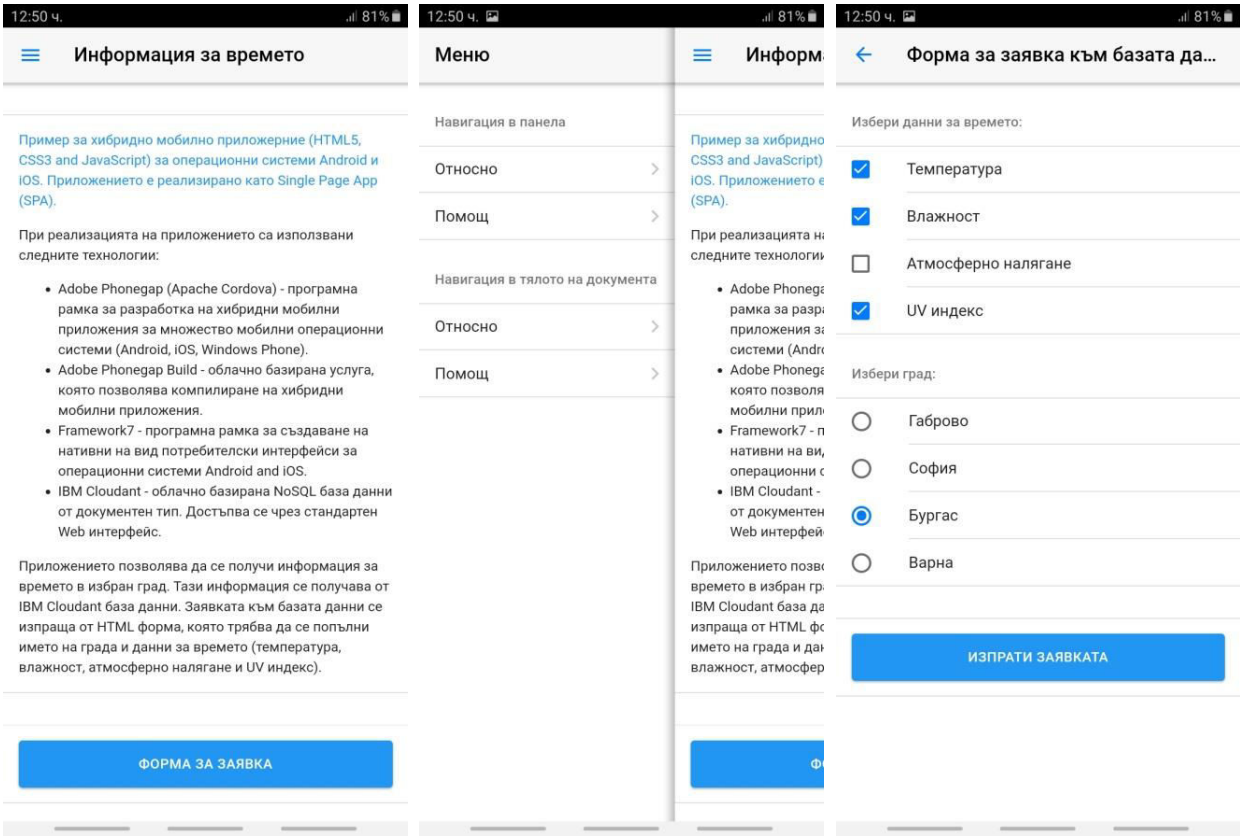
Всеки документ съдържа информация за времето в JSON формат. На Фиг. 8-5 е показано съдържанието на документ с име „Габрово”. Системните свойства в документа започват с долна черта, например ‘_id’ и ‘_rev’. Потребителските данни са JSON обекти с имена „температура”, „влажност” и „атмосферно налягане”. Всеки обект има по две свойства: value и units. Стойността на свойство „value” е последно измерената стойност за конкретния параметър, а стойността на свойство „units” – мерната единица.

2.2 Мобилно приложение

Мобилното приложение трябва да позволява информиране на клиентите за предназначението на приложението и да реализира HTTPS заявки към базата данни с цел извличане и визуализиране на информация за времето в избран от клиента град.

2.2.1. Потребителски интерфейс

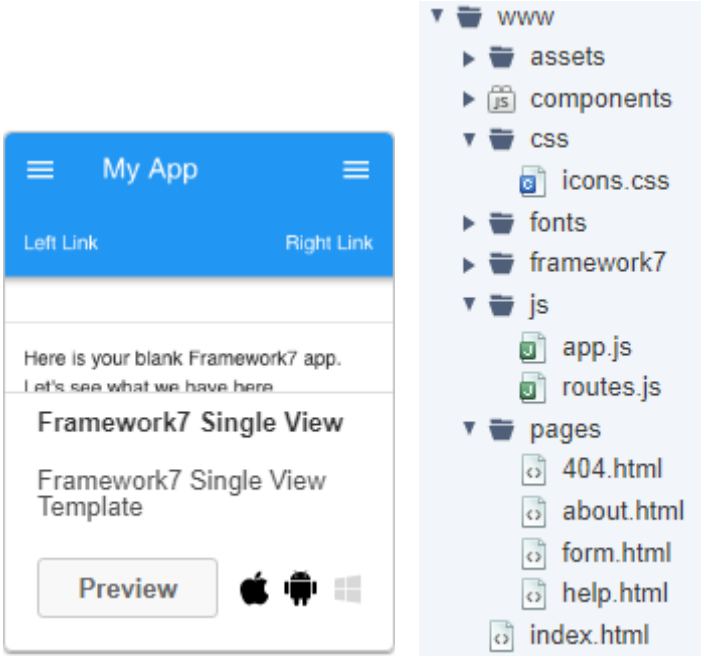
Нека приложението има главен изглед и ляв панел в който е менюто на приложението. От менюто да се получава помощна информация как се използва приложението, както и информация за автора. Да се предостави форма за изпращане на заявка към базата данни. Клиентът да има възможност да избере за кои параметри на времето иска да получи информация (check box) и името на града (radio бутони). Отделни екрани, които показват потребителският интерфейс, са показани на Фиг. 8-6.



Фиг. 8-6 Потребителски интерфейс на мобилното приложение

2.2.2. Структура на проекта

Създайте нов проект със шаблон за дизайн “Framework7 Single View” (виж Фиг. 8-7a). Структурата на проекта е показана на Фиг. 8-7b.



a) b)

Фиг. 8-7 Избор на шаблон за дизайн и структура на проекта

Приложението ще бъде изградено от следните файлове:

- **index.html** – основен документ на приложението, съдържа основния изглед.
- Папка pages. Съдържа следните HTML документи:
  - **about.html** – документ, който съдържа информация за автора.
  - **help.html** – документ, който съдържа информация как се използва приложението.
  - **404.html** – документ, който се зарежда когато се прави опит за достъп до документ, който не е регистриран за рутиране.
  - **form.html** – документ-шаблон, който показва форма за избор на параметри за времето и градовете, достъпни за клиента.
- Папка js. Съдържа потребителския JavaScript код:
  - **app.js** – програмен код за инициализация на приложението.
  - **routes**.js – програмен код с необходимите маршрути.
- Папка css. Ще използваме стандартното стилово форматиране на Framework7. Поради тази причина папка css ще съдържа само стиловото форматиране на иконите (icons.css).

2.2.3. Маршрутизация

Връзките между виртуалния път до ресурсите и тяхното действително местоположение в папка www се задава във файл routes.js (виж Фиг. 8-8).

```
routes = [
  {
    path: '/',
    url: './index.html',
  },
  {
    path: '/about/',
    url: './pages/about.html',
  },
  // Left Panel Pages
  {
    path: '/help/',
    url: './pages/help.html',
  },
  // Page Loaders
  {
    path: '/form/',
    componentUrl: './pages/form.html',
  },
  // Default route (404 page). MUST BE THE LAST
  {
    path: '(.*)',
    url: './pages/404.html',
  },
];
```

Фиг. 8-8 Съдържание на файл routes.js

2.2.4. Инициализация на приложението

Инициализацията на приложението да се реализира чрез програмен код от файл app.js (виж Фиг. 8-9). В този файл създаваме обекти за достъп до Dom7 ($$) и Framework7 (app). Задайте идентификатор и име на приложението.

```
// Dom7
var $$ = Dom7;
// Framework7 App main instance
var app = new Framework7({
  root: '#app',                        // App root element
  id: 'bg.tugab.kst.weather',          // App bundle ID
  name: 'Информация за времето',       // App name
  theme: 'auto',                       // Automatic theme detection
  // App root data
  data: function () {
    return {
      login: {
        name: 'dstionterystalletlypecel',
        password: 'be0864bb0315d45b28d333d991b21fda4e22b672',
      },
      server: {
        port: 443,
        protocol: 'https://',
        host:
     '024b178f-c9b8-4bc0-ae0e-67b8cbc7d5ec-bluemix.cloudantnosqldb.appdomain.cloud',
        database: 'weather',
      },
      weather: {
        data: [
          {
            value: 'Температура',
            selected: true,
          },
          {
            value: 'Влажност',
            selected: true,
          },
          {
            value: 'Атмосферно налягане',
            selected: false,
          },
          {
            value: 'UV индекс',
            selected: false,
          },

        ],
        city: [
          {             value: 'Габрово',
            selected: true,
          },
          {
            value: 'София',
            selected: false,
          },
          {
            value: 'Бургас',
            selected: false,
          },
          {
            value: 'Варна',
            selected: false,
          },
        ],
      },
    };
  },
  // App routes
  routes: routes,

});
```

```
// Init/Create left panel view
var leftView = app.views.create('.view-left', {
  url: '/'
});

// Init/Create main view
var mainView = app.views.create('.view-main', {
  url: '/'
});

// wait until cordova JS library is loaded
$$(document).on('deviceready', function () {
  // put your code here
});
```

Фиг. 8-9 Инициализация на приложението

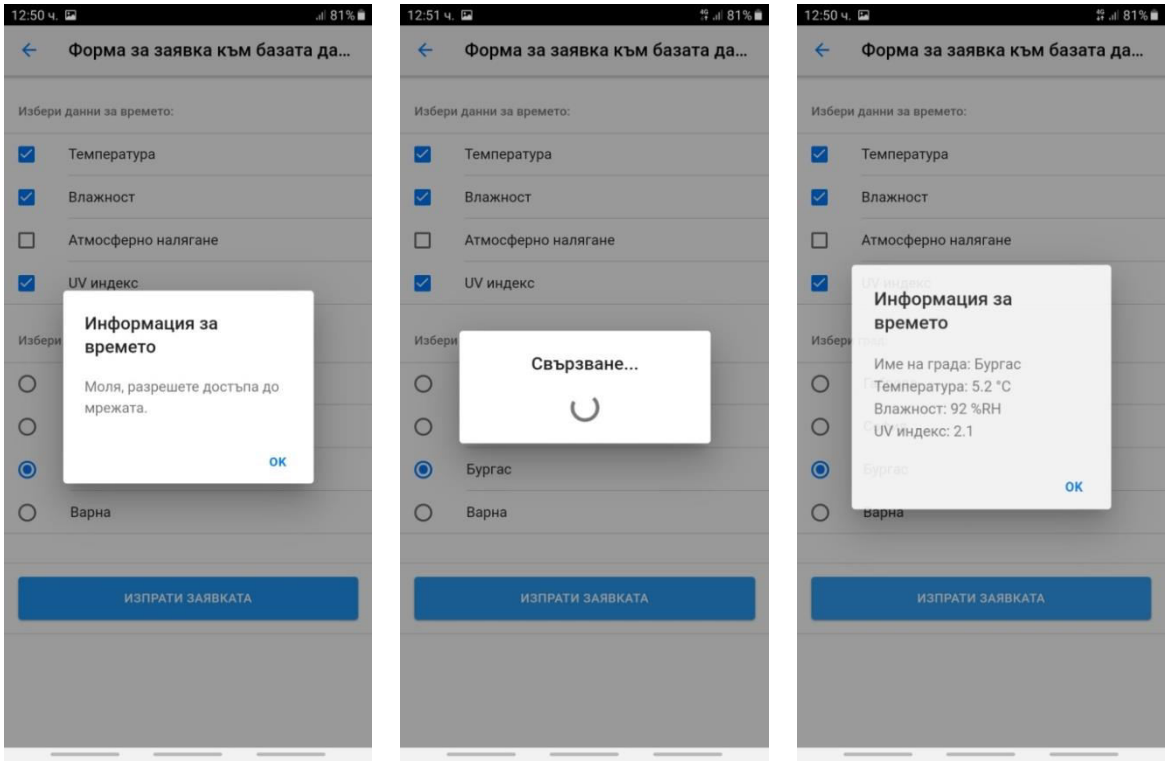
За работа на приложението са необходими следните root данни:

- Обект **login**. Съдържа ключа (name) и паролата (password) за достъп до документите от базата данни с информация за времето в отделните градове. Ако създадете своя база данни, променете стойностите на свойства name и password.
- Обект **server**. Съдържа информация, необходима за формиране на URL до база данни IBM Cloudant. Ако създадете своя база данни, променете стойността на свойство host.
- Обект **weather**. Съдържа информация, необходима за динамично създаване на формата за заявка към базата данни.
  - Масивът `data` описва всеки един от параметрите на времето (value) с които се работи (температура, влажност, атмосферно налягане и UV индекс) и дали всеки параметър трябва да бъде избран или не при след визуализиране на формата. Ако свойство selected е true, то съответния параметър ще се изобрази като избран. Визуализацията на параметрите на времето се реализира чрез компонент от тип “check box”.
  - Масивът `city` описва имената на градовете (свойство value) и кой град е избран по подразбиране (свойство selected=true).

И накрая, създайте необходимите view обекти. При конкретното приложение това са главния изглед (mainView) и левия панел (leftView).

2.2.5. Форма за изпращане на заявки към базата данни

Програмният код, който трябва да визуализира формата за изпращане на заявка към сървъра за управление на базата данни трябва да е във файл **form.html**. Тъй като формата трябва да се изгради динамично (на базата на стойността на root променлива weather), то е необходимо да използваме документ-шаблон. Интерфейсът с потребителя трябва да изглежда така, като е показано на Фиг. 8-10. Документът трябва да има навигационна лента (navbar) със заглавие и бутон за връщане към главния изглед. Параметрите за времето са визуализирани като “check box”, а имената на градовете чрез “radio” бутони. Ако липсва мрежова свързаност да се изведе съобщение „Моля, разрешете достъпа до мрежата.”. Заявката трябва да се изпрати при натискане на бутон „Изпрати заявката”. При свързване с базата данни да се показва компонент от тип “pre-loader” с текст „Свързване…”. След получаване на данните да се визуализира желаната от клиента информация (виж Фиг. 8-10).



Фиг. 8-10 Форма за изпращане на заявка към базата данни

На Фиг. 8-11 е показан програмния код, който дефинира желания документ-шаблон.

```
<template>
  <div class="page" data-name="form">
    <div class="navbar">
       // заглавие на страницата и икона Back
    </div>
    <div class="page-content">

    <div class="block-title">Избери данни за времето:</div>
    <div class="list">
      <ul>
        // динамично изграждане на секцията с "check box"
      </ul>
    </div>
    <div class="block-title">Избери град:</div>
    <div class="list">
      <ul>
        // динамично изграждане на секцията с "радио" бутоните
      </ul>
    </div>
    <div class="block block-strong">
        <div class="row">
            <div class="col-100">
                <a href="#" class="item-link col
                    button button-large button-fill button-raised color-blue submit"
                    @click="submit()">
                    Изпрати заявката
                </a>
            </div>
        </div>
    </div>
  </div>
</div>
</template>
```

Фиг. 8-11 Шаблон за изграждане на формата за изпращане на заявката

Изграждането на формата се реализира чрез Tempate7 помощници {{#each }} и {{#if }}. На Фиг. 8-12 е показано как може да се генерира секцията за избор на данни за времето.

```
{{#each data}}
   <li>
      <label class="item-checkbox item-content">
         <input type="checkbox" name="data" value="{{value}}"
               {{#if selected}} checked="checked"{{/if}}/>
         <i class="icon icon-checkbox"></i>
         <div class="item-inner">
            <div class="item-title">{{value}}</div>
         </div>
      </label>
   </li>
{{/each}}
```

Фиг. 8-12 Създаване на “check box” за избор на данните на времето

На Фиг. 8-13 е показано как може да се генерира секцията за избор на града.

```
{{#each city}}
   <li>
      <label class="item-radio item-content">
         <input type="radio" name="city" value="{{value}}"
               {{#if selected}} checked="checked"{{/if}}/>
         <i class="icon icon-radio"></i>
         <div class="item-inner">
            <div class="item-title">{{value}}</div>
         </div>
      </label>
   </li>
{{/each}}
```

Фиг. 8-13 Създаване на “radio” бутони за избор на града

Обектите `data` и `city` се получават чрез JavaScript код, който следва веднага след шаблона (виж Фиг. 8-14).

```
<script>
   return {
      data: function() {
         return {
            data: this.$root.weather.data,
            city: this.$root.weather.city,
         };
      },
      methods: {
         // методи, необходими за функциониране на страницата
      },
   };
</script>
```

Фиг. 8-14 JavaScript код от страницата-шаблон

Методите от свойство methods са два:

- Метод **sendQuery** служи за изпращане на заявката към IBM Cloudant.
- Метод **submit** се вика при натискане на бутон “Изпрати заявката”. Пренасочването на препратката се реализира чрез Framework7 атрибут @click (виж Фиг. 8-11).

Метод submit извлича всички данни, необходими за изпращане на заявка към базата данни: URL към базата данни, име и парола за авторизация на достъпа, както и

избраните от формата данни за времето и града. Следва изпращане на заявката чрез метод sendQuery. На Фиг. 8-15 е показан програмния код на метод **sendQuery**.

```
sendQuery: function(url, hash) {
  const TIMEOUT = 6000;
  app.dialog.preloader('Свързване...');
  var promise = new Promise(function(resolve, reject) {
    app.request({
      type: 'GET',
      url: url,
      async: true,
      cache: false,
      contentType: 'application/json',
      dataType: 'json',
      timeout: TIMEOUT,
      beforeSend: function(req) {
        req.setRequestHeader('Accept', 'application/json');
        req.setRequestHeader('Authorization', 'Basic ' + hash);
      },
      success: function(data) {
        resolve(data);
      },
      error: function(data) {
        reject(data);
      }
    });
  });
  return promise;
},
```

Фиг. 8-15 Метод sendQuery

Метод sendQuery използва системния метод request, за да изпрати GET заявка до базата данни. Методът получава два аргумента:

- **url** – низ, който сочи документа до който се желае достъп.
- **hash** - базата данни IBM Cloudant изисква базова авторизация на достъпа. За целта е необходимо да кодирате ключа и паролата чрез алгоритъм Base64 (функция btoa). Разделителят между ключа и паролата трябва да бъде символ двоеточие:

```
let hash = btoa(key + ':' + password);
```

Тъй като метод request работи асинхронно, ще използваме обект-обещание (promise), за да разберем кога методът връща информация. При успешно изпълнена заявка се вика callback функция **success**. При възникване на грешка се вика callback функция **error**. При успех обектът data съдържа желания документ в JSON формат. При неуспех – обектът data съдържа информация за грешката. Чрез свойство timeout задаваме колко време трябва да се изчака сървърът да върне отговор. При конкретният пример – 6 секунди (6000 ms). Данните за авторизация на достъпа се задава чрез параметър Authorization от заглавния блок на заявката. Трябва да използвате базова авторизация (Basic). Забележете, че параметър Authorization се задава чрез свойство beforeSend (sendRequestHeader). Това е необходимо, за да се предотврати браузърът автоматично да анулира това поле.

Програмният код на метод **submit** е показан на Фиг. 8-16. Започва се с извличане на необходимите данни с цел формиране на URL за достъп до базата данни, данните за базова авторизация, както и избрания град и данните за времето от които се интересуваме. Ако не е избрана нито една от възможните параметри за времето се

показва съобщение “Трябва да изберете поне един параметър!” и функцията завършва (return).

```
      submit: function() {
        let self = this;
        let username = this.$root.login.name;
        let password = this.$root.login.password;

        let protocol = this.$root.server.protocol;
        let port = this.$root.server.port;
        let host = this.$root.server.host;
        let database = this.$root.server.database;

        let dataValues = [];
        let cityData = $$('[name="city"]:checked');
        let city = cityData[0].defaultValue

        let dataSelected = $$('[name="data"]:checked');
        let n = dataSelected.length;
        if (n === 0) {
          app.dialog.alert("Трябва да изберете поне един параметър!");
          return;
        }
        let url = encodeURI(protocol+host+':'+port+'/'+database+'/'+city);
        let hash = btoa(username + ':' + password);
        self.sendQuery(url, hash)
          .then(
            function(document) {
              let result = 'Име на града: ' + city + '<br/>';
              for (let i = 0; i < n; i++) {
                let name = dataSelected[i].defaultValue;
                let nameLowerCase = name.toLowerCase();
                let feature = document[`${nameLowerCase}`];
                if (feature !== undefined) {
                  let value = feature.value;
                  let units = feature.units;
                  if (value !== undefined) {
                    result += `${name}: ${value} ${units}<br/> `;
                  }
                }
              }
              app.dialog.alert(result);

            }).catch(
            function(error) {
              console.log(error);
              let status = '';
              if (error.status === 0) {
                status = 'Моля, разрешете достъпа до мрежата.';
              } else {
                status = "Невъзможно свързване с базата данни: "+error.statusText;
              }
              app.dialog.alert(status);

            }).finally(function() {
              app.dialog.close();
          });
        }
```

Фиг. 8-16 Метод submit

Адресът на желания документ (URL) се получава по следния начин:

```
protocol + host + ':' + port + '/' + database + '/' + city
```

където city е името на избрания град. Тъй като имената на градовете са на български език, се налага URL низа да бъде URL кодиран, преди да бъде изпратена заявката:

```
let url = encodeURI(url);
```

Ако всички данни са налични се вика метод sendQuery. При успешно изпълнение на заявката управлението се предава на функция `then`, а при грешка – на функция `catch`.

При успешно изпълнение на заявката се получава обект document. В цикъл (for) от базата данни за документ с име, което съвпада с името на града се извлича информация за всички избрани чрез компонент “check box” параметри за времето. Ако в документа има информация за даден параметър се получава неговата стойност (value) и мерна единица (units). Данните за всеки параметър за времето се добавят към променлива result. Крайният резултат се показва чрез “Alert box”.

При неуспешно изпълнение на заявката се извлича кода на грешката (error.status). Код със стойност 0 означава, че няма мрежова връзка. Ако е необходимо може да анализирате и други кодове за статуса на отговора, например timeout и грешка при авторизация на достъпа. Тази информация може да получите и от свойство statusText (обект error).

2.2.6. Тестване и изграждане на приложението

След като въведете програмния код и изчистите всички синтактични грешки тествайте приложението чрез Monaca Debugger. Ако приложението е напълно функционално пременете към изграждане на приложение за OS Android.

2.3 Разширяване на функциите на приложението

Създайте мобилно приложение, което симулира работата на метеорологична станция. Приложението да предоставя форма, която позволява избор на град и показва текущите стойности на всички параметри на времето. Клиентът да може да променя стойностите на всеки един от тези параметри и да изпраща заявка към базата данни с цел запис на обновената информация. Да се потвърждава всеки успешен запис в базата данни.

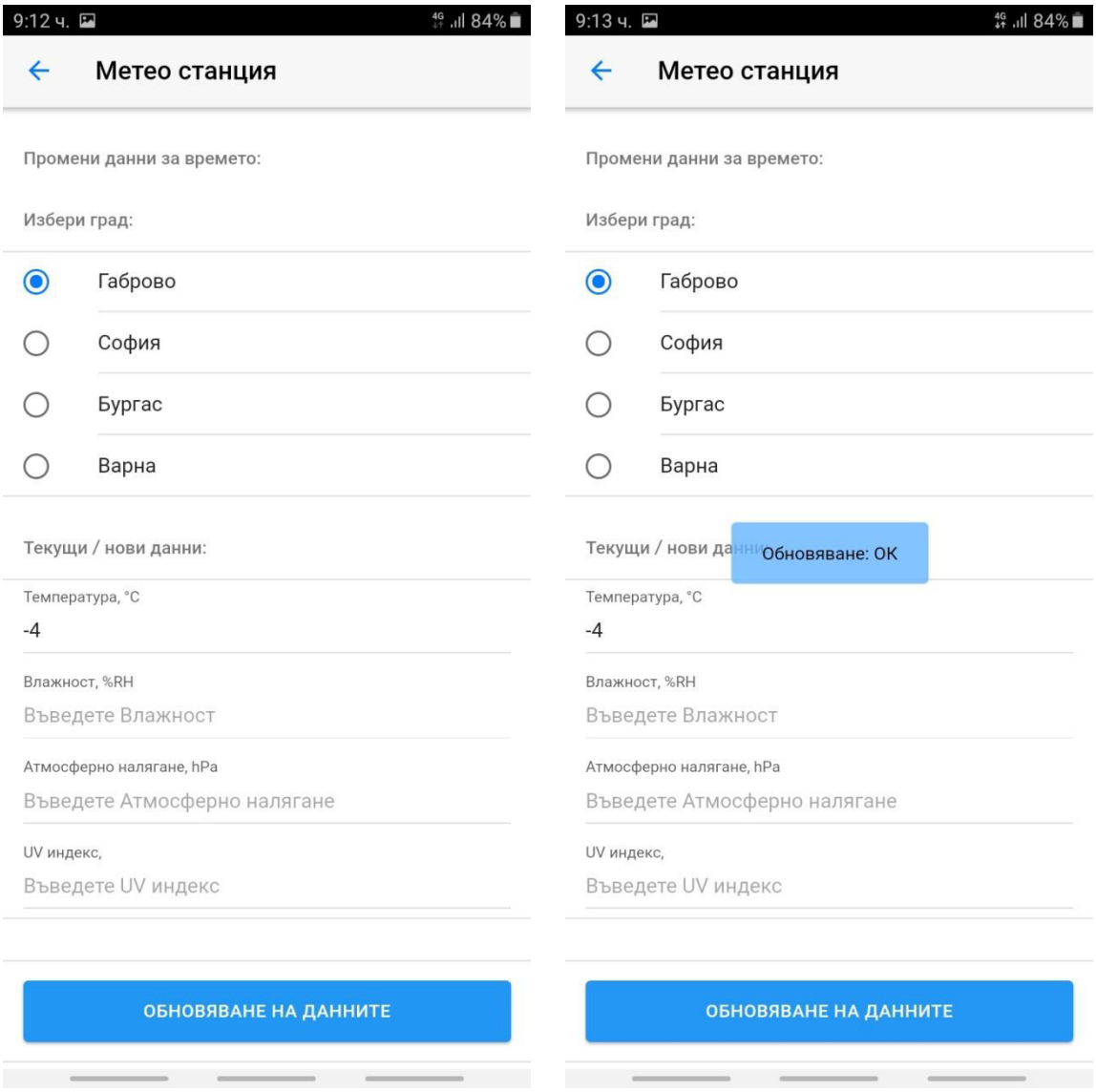
Създайте нова двойка ключ-парола чрез които ще достъпвате базата данни с цел четене и запис. При IBM Cloudant не е възможно промяна на част от съдържанието на документ. Всяка промяна се свежда до презаписване на съдържанието на целия документ. Следователно, промяната се реализира в три стъпки: 1) Прочита се съдържанието на документа като JSON обект; 2) Реализират се необходимите промени на свойствата на обекта; и 3) Записва се промененият документ в базата данни.

Заявката за запис е идентична със заявката за чете с разлика на две свойства на обекта, който се предава като аргумент на метод app.request():

- Свойство “type” трябва да има стойност PUT.
- Въвежда се ново свойство “data” със стойност съдържанието на документа като низ. За целта използвайте метод JSON.stringify(doc), за да конвертирате JSON обекта-документ (doc) до низ.

Потребителят трябва да може да обновява избрани параметри за времето. За целта динамично се създава форма за въвеждане на информацията. Обновява се стойността на тези параметри за който има въведена информация. Да се валидира въведената стойност за всеки един параметър.

На Фиг. 8-17 е показано как трябва да изглежда страницата за симулиране на работата на метеорологична станция.



Фиг. 8-17 Изглед на формата за симулиране на метеорологична станция

Заключение

В тази глава научихте какво са нерелационни бази данни и какви са техните предимства. Научихте се да създавате услуги без сървърен код (serverless apps). Проектът от тази глава е пример за това как се реализират асинхронни комуникационни канали чрез Framework7 с облачно базирани услуги – база данни IBM Cloudant.

Глава 9

RSS новини

I. Въведение

RSS (RDF Site Summary или Really Simple Syndication) е вид Web емисия, която позволява на потребителите и приложенията да получават достъп до актуализации на уебсайтове в стандартизиран формат, който може да се чете от компютър. Рамката за описание на ресурси (RDF) е стандарт на World Wide Web Consortium (W3C), първоначално разработен като модел за метаданни. Абонирането за RSS емисии може да позволи на потребителя да следи много различни уебсайтове в един агрегатор за новини, който постоянно следи сайтовете за ново съдържание, като премахва необходимостта потребителят да ги проверява ръчно. Новинарските агрегатори на съдържание (RSS четци) могат да бъдат вградени в браузър, да бъдат инсталирани на настолен компютър или на мобилно устройство. RSS документът (наричан "емисия") е XML файл, който съдържа пълен или обобщен текст и метаданни, като заглавие, дата на публикуване и име на автора (Wikipedia).

II. Дефиниране на стратегия

Ще създадем мобилно приложение, което извлича от RSS канали и визуализира на български език новини. За да увеличим броя на потребителите на приложението ще предоставим възможност за избор на категория на новините (международни, спортни, технически и др.). Колкото повече е броя на предлаганите категории новини, толкова повече потребители би имало приложението. Приложението ще визуализира само новини, без рекламна информация. Целта е да се разработи мобилно приложение, чрез което максимално бързо и с минимален трафик да се визуализират новини от RSS канали, които са систематизирани по тип. Приложението трябва да предоставя избор на типа на новините. Потребителят да може да получава кратко описание на последните n на брой новини и ако желае – да разгледа цялото съдържание на избрана от него новина.

Ще използваме последната версия (6.x) на технологична рамка Framework7, за да реализираме приложението. Ще работим с компоненти само от тази програмна рамка, за да реализираме потребителския интерфейс на приложението.

III. Анализ

За извличане на метаданни от RSS канали се използват парсери на RSS емисии. Тъй като има множество формати за RSS емисии, универсалните парсери са с голям размер на програмния код. За да опростим приложението ще използваме собствен парсер на RSS емисии. Ще анализираме RSS новините от сайта fakti.bg, които са систематизирани в 12 RSS канала. Данните, които ще предоставим на потребителите ще бъдат следните:

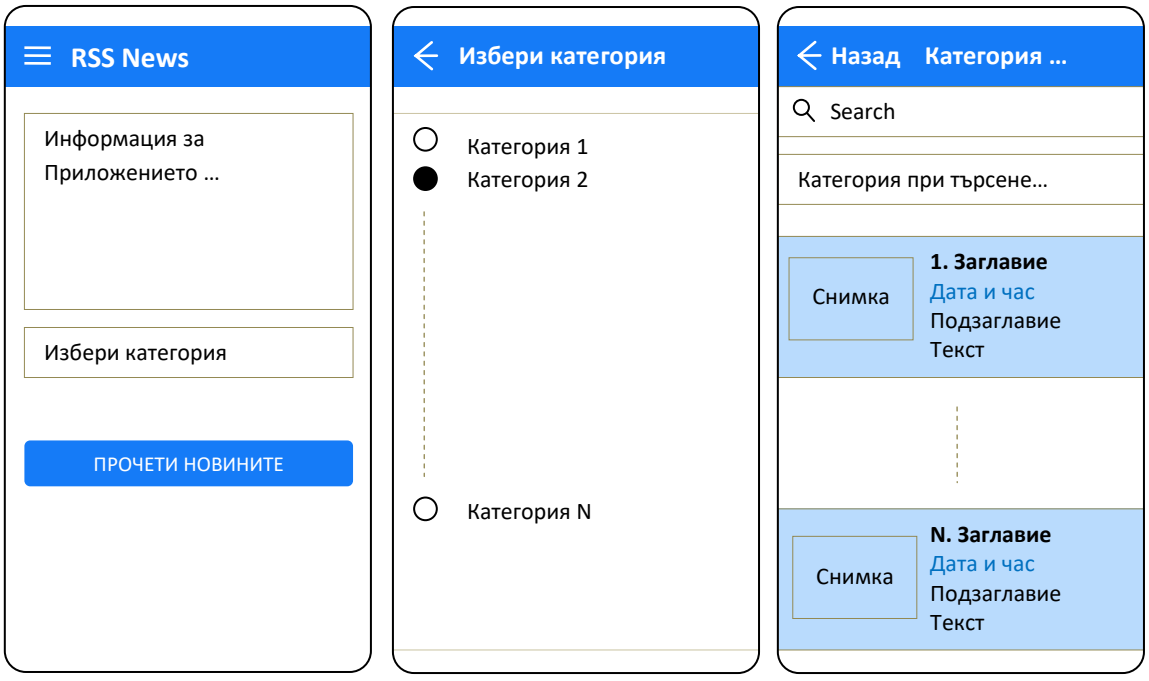
- Заглавие на новината.
- Подзаглавие на новината.
- Дата и час на генериране на новината.
- Снимка, ако има такава.
- Текстово описание на новината.

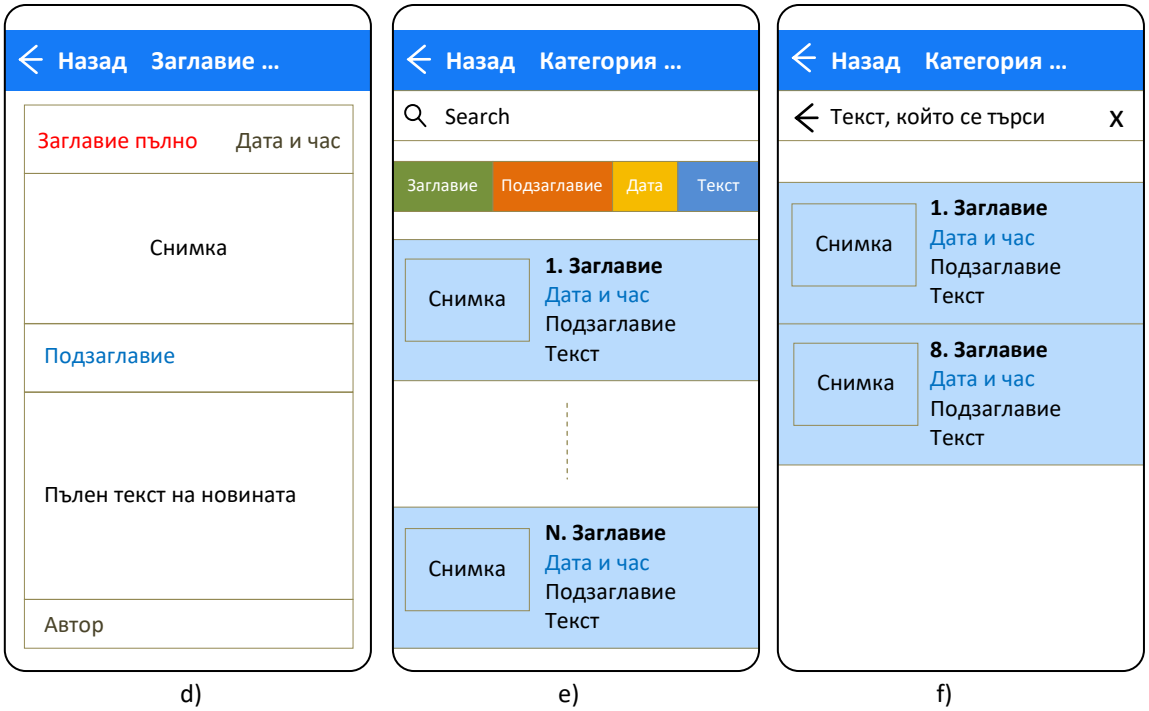
- Автор.

Ще предоставим възможност за филтриране на новините по един от следните параметри: заглавие, подзаглавие, дата и текстово описание на новината.

IV. Потребителски интерфейс

Ще използваме дизайн на потребителския интерфейс, който бързо и систематизирано да визуализира желаните от потребителя новини. На Фиг. 9-1 е скициран желания потребителски интерфейс. Ще използваме графични компоненти само от програмна рамка Framework7, за да гарантираме нативния изглед на приложението.





Фиг. 9-1 Скициране на потребителския интерфейс

Началната страница (Фиг. 9-1a) трябва да предоставя възможност за избор на категория на новините. Желателно е изборът на категория да се реализира в отделен прозорец, за да не се налага превъртане на екрана вертикално. Най-добре е да използваме компонент “virtual select” (виж Фиг. 9-1b).

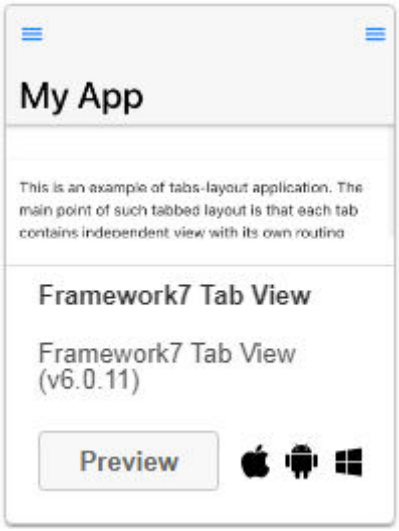
След натискане на бутон “Прочети новините” се реализира основната логика на приложението – свързване с източника на RSS новини, извличане на новините като XML документ, извличане на и визуализиране на новините, както е показано на Фиг. 9-1c. За визуализация на новините ще използваме компонент “virtual list”, а за филтрирането им – “search bar”. За да не се превърта “search bar”, ще го визуализираме като “subnavbar” – непосредствено под заглавието на приложението. Изборът на категория за филтриране на новините ще реализираме чрез компонент “swipeout”. Чрез изтегляне с пръст в дясно се виждат достъпните категории (виж Фиг. 9-1e). При въвеждане на текст в поле Search остават само новините, които отговарят на избрания филтър (виж Фиг. 9-1f).

При избор на конкретна новина, в нов прозорец се показва пълна информация за новината (виж Фиг. 9-1d). За да форматираме наличните метаданни ще използваме компонент “card”. Избрани са следните цветове за визуализация на данните:

- Червено за заглавието на новината.
- Синьо за подзаглавието на новината.
- Сиво за автора, датата и часа.
- Черно за текстовото описание на новината.

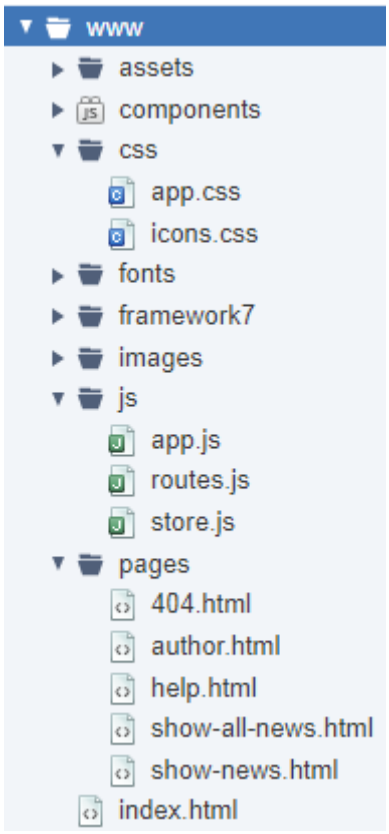
V. Програмна реализация

Ще използваме версия 6.x на програмна рамка Framework7, за да реализираме приложението. За целта създайте празен проект (Blank Project) и прехвърлете в него съдържанието на папка www от готовия проект. Можете да използвате шаблон “Framework7 Tab View” (виж Фиг. 9-2), който е пример как да се работи с версия 6.x на Framework7. Програмният код на рамката (JavaScript и CSS) е в папка Framework7. Ако използвате версия 6.x на Framework7 можете да изградите приложението за ОС Android, iOS или Windows.



Фиг. 9-2 Избор на шаблон “Framework7 Tab View”

Структурата на проекта “RSS News” е показана на Фиг. 9-3.



Фиг. 9-3 Структура на проекта RSS News

Потребителският програмен код, който реализира проекта, е следния:

- Индексен файл **index.html** – визуализира информация какво представлява приложението, компонент “smart select” чрез който се избира тип на новините и бутон за стартиране на процедурата за извличане на RSS новини.
- Папка **pages** – съдържа 4 html документа, които имат следното предназначение:
  - Файл **show-all-news.html** – шаблон чрез който се визуализира съдържанието на всички новини от избран тип (виж Фиг. 9-1c).
  - Файл **show-news.html** – шаблон чрез който се визуализира пълната информация за избрана новина (виж Фиг. 9-1d).
  - Файл **author.html** – съдържа информация за автора.
  - Файл **help.html** – съдържа помощна информация за приложението.
  - Файл **404.html** – съдържанието на този документ се визуализира при заявка, за която липсва маршрут.
- Папка **js** – съдържа 3 JavaScript файла, които се използват както следва:
  - Файл **app.js** – съдържа програмния код за инициализация на приложението.
  - Файл **routes.js** – създава масив routes чрез който се описват всички маршрути.
  - Файл **store.js** – съдържа програмния код чрез който се създава локално хранилище в което ще запишем данните, необходими за функциониране на приложението.

5.1 Индексен файл

От индексния файл трябва да заредите необходимия JavaScript и CSS код. Всички CSS правила се зарежда чрез етикет script от тялото на етикет head:

```
<link rel="stylesheet" href="framework7/framework7-bundle.min.css">
<link rel="stylesheet" href="css/icons.css">
<link rel="stylesheet" href="css/app.css">
<link rel="stylesheet" href="components/loader.css">
```

В края на етикет body заредете всички JavaScript библиотеки и файлове в следната последователност:

```
<!-- Framework7 library -->
<script src="framework7/framework7-bundle.min.js"></script>
<!-- App routes -->
<script src="js/routes.js"></script>
<!-- App store -->
<script src="js/store.js"></script>
<!-- App scripts -->
<script src="js/app.js"></script>
<script src="components/loader.js"></script>
```

Програмният код, който реализира интерфейса на приложението е показан на Фиг. 9-4.

```
<div class="page">
 <div class="page-content">
  <div class="block">                                                    1
   <p class="text-color-blue">
    Пример за хибридно мобилно приложение което чете и визуализира данни
    от RSS канали . . .
   </p>
  </div>
  <div class="block">                                                    2
   <div class="list">
    <ul>
     <li>
      <a class="item-link smart-select smart-select-init">
       <select name="news-category" class="news-category">
       </select>
       <div class="item-content">
        <div class="item-inner">
         <div class="item-title">Избери категория</div>
        </div>
       </div>
      </a>
     </li>
    </ul>
   </div>
  </div>
  <div class="block block-strong">                                       3
   <a href="/newsAll/" class="item-link col button button-large
     button-fill button-raised color-blue" data-view=".view-main">
     Прочети новините
   </a>
  </div>
 </div>
</div>
```

Фиг. 9-4 Съдържание на основната страница

Информацията, която се визуализира е структурирана чрез три div контейнера от клас “block”. В първият контейнер се съдържа текст, който описва какво е RSS новини. Вторият контейнер съдържа “smart select” компонент чрез който потребителят има възможност да избере категория на новините. Информацията за категорията на новините се вмъква динамично в тялото на етикет select по време на инициализация на приложението.

Третият контейнер съдържа бутон чрез който се адресира ресурс “/newsAll/”. След натискане на бутона трябва да се зареди документа-шаблон “show-all-news.html”.

5.2 Инициализация на приложението

Инициализацията на приложението се реализира чрез програмния код от файл app.js (виж Фиг. 9-5).

```
// Dom7
var $$ = Dom7;
// Framework7 App main instance                                          1
var app = new Framework7({
  el: '#app',                       // App root element
  id: 'bg.tugab.kst.rssnews',       // App bundle ID
  name: 'RSS News',                 // App name
  theme: 'auto',                    // Automatic theme detection
  view: {
    stackPages: true,               // view parameters
  },
  lazy: {
    sequential: false,              // lazy parameters
    observer: false,                // …
  },
  store: store,                     // Store object
  routes: routes,                   // Router object
});
// Init/Create main view                                                 2
var mainView = app.views.create('.view-main', {
  url: '/'
});
// Init/Create left panel view
var leftView = app.views.create('.view-left', {
  url: '/'
});
// Build smart select component                                          3
let rssNewsSorted =
    store.state.rssnews.sort((a, b) => a.category < b.category ? -1 : 1);
let newsCategory = $$('.news-category');
rssNewsSorted.forEach(news => {
  let category = news.category;
  newsCategory.append(
      `<option value="${category}">${category}</option>`
  );
  newsCategory.val("Последни").change();
});
```

Фиг. 9-5 Инициализация на приложението – файл app.js

След създаване на обект $$ за достъп до DOM7 се създава обект app чрез викане на конструктора на клас Framework7 (Секция 1).

Следва създаване на обекти за достъп до двата изгледа с които приложението работи: основен изглед (mainView) и изглед за левия панел (leftView) - програмен код от Секция 2.

Програмния код от Секция 3 създава динамично компонент “smart select” чрез който потребителят може да избере една от всички категории новините (radio button). Имената на всяка категория (поле category), както и URL адресът (поле url) за достъп до съответния RSS канал са записани в Framework7 store – масив rssnews. Достъпът до този обект се реализира по следния начин:

```
store.state.rssnews
```

За да може потребителят да вижда категориите новини, подредени по азбучен ред, се реализира сортиране на полетата на масив rssnews по стойността на поле category:

```
store.state.rssnews.sort((a, b) => a.category < b.category ? -1 : 1);
```

Чрез метод forEach се реализира програмен цикъл чрез който за всяка категория на новините в тялото на етикет select (клас “news-category”) се вмъква по един етикет option. За целта се използва метод append. Накрая, чрез метод change се задава категория “Последни” да е избраната по подразбиране:

```
newsCategory.val("Последни").change();
```

5.3 Хранилище

Програмната рамка Framework7 от версия 6 има вградена поддръжка за управление на състоянието на приложението - Framework7 Store. Тя служи като централизирано хранилище за данните с които приложението работи. Създаването на хранилището на приложението се реализира чрез програмния код от файл store.js (виж Фиг. 9-6). Ще използваме хранилището за буфериране на информация за категориите новини (масив rssnews) и за мета данните за последно прочетените новини (newsdata). Създаването на обект store се реализира чрез метод createStore. Състоянието (state) е единственият обект, който съдържа цялото състояние на ниво приложение.

```
var createStore = Framework7.createStore;
const store = createStore({
 state: {
  newsdata: [],
  rssnews: [
   {
    id: 0,
    url: 'https://fakti.bg/feed',
    category: 'Последни'
   },
   . . .
   {
    id: 14,
    url: 'https://fakti.bg/feed/bulgaria',
    category: 'България'
   },
  ],
 },
});
```

Фиг. 9-6 Хранилище на приложението – файл store.js

5.4 Маршрутизатор

При конкретното приложение маршрутизатора трябва да реализира 4 статични маршрута и 2 динамични маршрута (виж Фиг. 9-7). Статичните маршрути са към следните ресурси: index.html, author.html, help.html и 404.html. Динамичните ресурси обработват заявките към “/newsAll/” и /show/news/:newsId/”. Заявката към “/newsAll” трябва да бъде пренасочена към ресурса “show-all-news.html”, заявката към “/show/news/:newsId/” – към ресурса “show-news.html”. Ресурсът “show-all-news.html” трябва да визуализира новините от избрана категория. Преди да се предаде управлението на този ресурс, трябва да се провери дали достъпа до мрежата е разрешен. За целта ще използваме филтриране на заявката чрез свойство async. Стойността на това свойство е анонимна функция, която

проверява наличието на мрежова свързаност. Това се реализира чрез плъгина “Network Information” (`navigator.connection.type`).

```
routes = [
  {
   path: '/',
   url: './index.html',
  },
  {
   path: '/author/',
   url: './pages/author.html',
  },
  {
   path: '/help/',
   url: './pages/help.html',
  },
  {
   path: '/newsAll/',
   async: function ({ resolve, reject }) {
    var networkState = navigator.connection.type;
    if (networkState !== Connection.NONE) {
     resolve({
       componentUrl: './pages/show-all-news.html',
     });
    }
    else {
     app.dialog.alert('Моля, разрешете мрежовата свързаност!');
     reject();
    }
   },
  },
  {
   path: '/show/news/:newsId/',
   async: function ({ to, resolve }) {
    var newsId = to.params.newsId;
    var newsInfo = store.state.newsdata[newsId];
    resolve({
      componentUrl: './pages/show-news.html',
     },
     {
      props: {
       news: newsInfo,
      }
     }
    );
   },
  },
  // Default route (404 page). MUST BE THE LAST
  {
   path: '(.*)',
   url: './pages/404.html',
  },
];
```

Фиг. 9-7 Маршрутизатор – файл route.js

Ако типа на връзката не е `Connection.NONE` се преминава към пренасочване към ресурса, тъй като има налична мрежова свързаност. За целта към метод `resolve` се предават аргумент `componentUrl` – пътя до ресурса. Ако липсва мрежова свързаност се вика метод `reject`, за да се блокира пренасочването към ресурса.

По аналогичен начин се филтрира маршрута към ресурса “show-news.html”. Неговата задача е да визуализира всички мета данни свързани с конкретна новина. За целта към заявката “/show/news” се предава параметър newsId – идентификатор на новината, която

трябва да се визуализира. Чрез обект `to`, който се предава на анонимната функция се получава достъп до параметър `newsId`:

```
var newsId = to.params.newsId;
```

От хранилището извличаме мета данните за тази новина:

```
var newsInfo = store.state.newsdata[newsId];
```

Следва викане на метод `resolve`, за да разрешим пренасочването до желания ресурс. Към ресурса “show-news.html” се предава свойство `news`, чиято стойност е JSON обект с мета данните за конкретната новина.

5.5 Визуализация на всички новини от избрана категория

Получаването на новините от избрана категория като XML файл, парсването на документа (извличане на мета данните за новините) и динамичното създаване на обекти за филтриране и визуализиране на новините се реализира чрез ресурса “show-all-news.html”. Това е документ-шаблон. Трябва да имате предвид, че Framework7 6.x не поддържа двигателя Template7, защото има вградена поддържа за работа с шаблони. Вместо да вграждате променливи в HTML кода чрез {{variable}} трябва да използвате следния синтаксис ${variable}.

На Фиг. 9-8 е показан HTML кода и част от JavaScript кода от ресурса “show-all-news.html”. В навигационната лента задаваме възможност за връщане към предходния изглед и заглавието на страницата – избраната категория новини (Секция 1). Под навигационната лента (subnavbar) визуализираме компонент от тип searchbar чрез който потребителя да може да филтрира новините (Секция 2). По този начин компонента за търсене ще бъде фиксиран и няма да се превърта със съдържанието на страницата. Съдържанието на страницата (Секция 3) съдържа два div контейнера. Единият контейнер е от клас “searbar-not-found” и служи за показване на текст, че няма съвпадение по зададения ключ за търсене на новини. Другият контейнер е от клас “searchbar-found”. Този контейнер съдържа динамично изграден компонент от тип “virtual list” чрез който се визуализират новините.

Програмният код за четене на RSS канали, за парсване на XML код, извличане на мета данни за новините и динамично създаване на компонентите от потребителския интерфейс се намира в секция 4 (виж Фиг. 9-8). Ламбда функцията, обявена като **export default** получава достъп до редица обекти като props, $f7, $theme, $on и $store. Чрез метод readNews се реализира пълната функционалност от четене на RSS канали до запазване на мета данните за новините в масив от JSON обекти. Метод readNews се вика при инициализиране на страницата (състояние pageInit). При успешното изпълнение на този метод, получените мета данни се записват в хранилището и се създават всички компоненти на потребителския интерфейс. При получаване на изключение след изпълнение на метод readNews се визуализира информация за проблема.

Програмният код на метод readNews е показан на Фиг. 9-9. Дефинират се константи чрез които се задава максималния брой на новините, които да се анализират, както и необходимите регулярни изрази за парсване на съдържанието на RSS новините. Свързването със източника на новини се реализира чрез метод request. Адресът на RSS канала (URL) се получава от хранилището на базата на текущо избраната категория новини. За типа на отговора (dataType) трябва да се зададе “xml”. Парсването на XML отговора с новините се реализира чрез клас DOMParser:

```
var doc = $$(new DOMParser().parseFromString(data, 'text/html'));
```

```
<template>
  <div class="page" data-name="newsAll">
```

1

```
    <div class="navbar">
    <div class="navbar-bg"></div>
    <div class="navbar-inner sliding">
     <div class="left">
      <a href="#" class="link back">
       <i class="icon icon-back"></i>
       <span>Назад</span>
      </a>
     </div>
     <div class="title">${category}</div>
```

2

```
    <div class="subnavbar">
      <form data-search-container=".virtual-list-news"
          data-search-item="li" data-search-in=".item-title"
          id="news-search" class="searchbar searchbar-init">
       <div class="searchbar-inner">
        <div class="searchbar-input-wrap">
         <input type="search" placeholder="Search"/>
          <i class="searchbar-icon"></i>
          <span class="input-clear-button"></span>
        </div>
        <span class="searchbar-disable-button if-not-aurora">Cancel</span>
       </div>
      </form>
     </div>
    </div>
  </div>

  <div class="searchbar-backdrop"></div>
```

3

```
  <div class="page">
   <div class="page-content">
    <div class="list virtual-list-search media-list"></div>
    <div class="list simple-list searchbar-not-found">
     <ul>
      <li>Няма съвпадение по зададения ключ!</li>
     </ul>
    </div>
    <div class="list virtual-list-news media-list searchbar-found"></div>
    <div class="block" id="news-not-found"></div>
   </div>
  </div>
</div>
</template>
<script>
```

4

```
export default (props, { $f7, $theme, $on, $store }) => {

  const readNews = (url) => {
     // read RSS news, parse XML doc and return metadata as JSON
  };
  $on('pageInit', (e, page) => {
     // Call readNews method: write metadata into store,
     // create search bar and virtual list components
  };
  return $render;
}
```

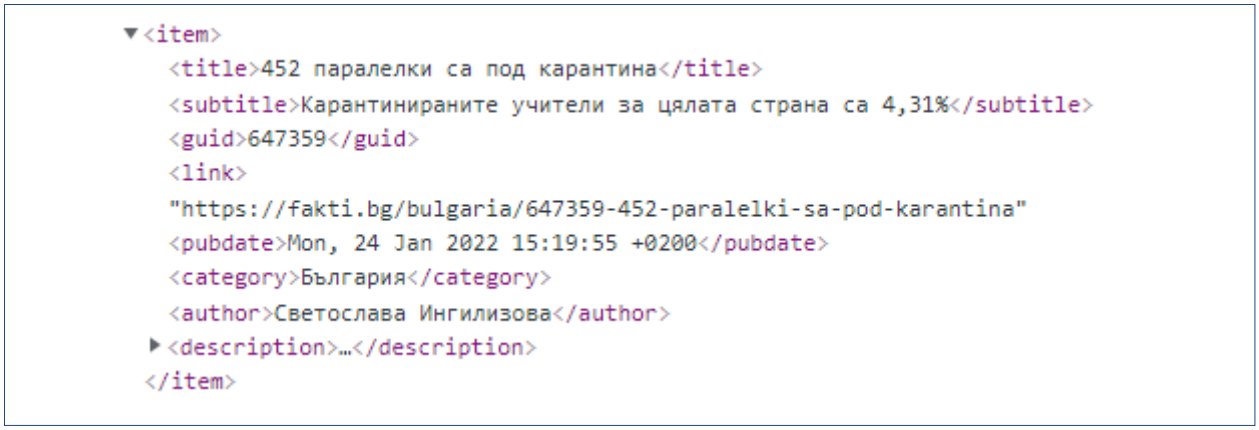
```
<script>
```

Фиг. 9-8 Визуализиране на всички новини от избрана категория – файл “show-all-news.html”

```
const readNews = (url) => {
 var news = [];
 const MAX_NUM_OF_NEWS = 20;
 const regex = /^<\!\[CDATA\[|\]\]>$/g;
 const regex1 = /<blockquote.*class="(.*?)".*>(.*?)<\/blockquote>/gi;
 const regex2 = /<script.*async(.*?)".*>(.*?)<\/script>/gi;
 const imageMatch = /<img [^>]*src="[^"]*"[^>]*>/gm;
 const imageReplace = /.*src="([^"]*)".*/;

 var promise = new Promise(function(resolve, reject) {
  app.request({
   url: url,
   type: 'GET',
   async: true,
   timeout: 5000,
   dataType: 'xml',
   success: function(data) {
    var index = 0;
    var doc = $$(new DOMParser().parseFromString(data, 'text/html'));
    try {
     doc.find('item').forEach(function(item) {
      let title = $$(item).find('title').text();
      let subtitle = $$(item).find('subtitle').text();
      let link = $$(item).find('link');
      let imageUrl = link[0].nextSibling.data;

      let pubDate = $$(item).find('pubDate').text();
      let dateObj = new Date(pubDate);
      let year = dateObj.getFullYear();
      let day = dateObj.getDate().toString().padStart(2, '0');
      let month = (dateObj.getMonth() + 1).toString().padStart(2, '0');
      let hours = dateObj.getHours().toString().padStart(2, '0');
      let minutes = dateObj.getMinutes().toString().padStart(2, '0');
      let pubDateAndTime = `${day}.${month}.${year}, ${hours}:${minutes} `;

      let author = $$(item).find('author').text();
      let description = $$(item).find('description').text()
        .replace(regex, '').replace(regex1, '').replace(regex2, '');
      let descriptionHtml = $$(item).find('description').html()
        .replace(regex, '');
      let images = descriptionHtml.match(imageMatch)
          .map(x => x.replace(imageReplace, '$1'));

      let numberOfImages = images.length;
       if (numberOfImages == 0) {
        images.push('images/noimage.png');
       }
       let data = {
        newsid: index, author: author, title: title, subtitle: subtitle,
        date: pubDateAndTime,  info: description, imageurl: images[0],
       };
       news.push(data);
       index++;
       if (index === MAX_NUM_OF_NEWS) {
        throw 'break';
       }
     });
    } catch (e) {
     if (e != 'break') throw e;
    }
    resolve(news);
   },
   error: function(error) {
    reject(error);
   }
  });
 });
 return promise;
}
```

Фиг. 9-9 Метод readNews

Така полученият обект `doc` предоставя метод `find` чрез който може да търсим определени свойства, които представляват интерес за нас. На Фиг. 9-10 е показано част от съдържанието на обекта `doc`, свързано с конкретна новина.



Фиг. 9-10 Структурно описание на новина в XML формат

Всички мета данни се описват чрез стойността на свойство `item`. Свойствата, които ще използваме са следните:

- `title` – заглавие на новината.
- `subtitle` – подзаглавие на новината.
- `category` – категория на новината.
- `pubdate` – дата на публикуване на новината.
- `author` – автор на новината.
- `description` – описание на новината

Трябва да имате предвид, че структурата на свойство `description` може да се различава за различните RSS канали. При конкретното приложение ще използваме регулярни изрази, за да извлечем необходимите данни, генерирани от RSS каналите на fakti.bg. Ако трябва приложението да обработва данни от други канали, трябва да използвате библиотека, която да предостави пълната функционалност за един RSS парсер. След като бъдат извлечени мета данните за една новина се създава обект `data`, който в JSON формат описва тези данни. Така формираните данни за новина се записват в масив `news`.

Тъй като метод request работи асинхронно ще използваме обект-обещание, за да разберем кога метод readNews е завършил и как е завършил – успешно или с грешка. При успешно изпълнение съдържанието на обекта news се връща чрез метод `resolve`. При грешка, обектът описващ грешката (error) се връща чрез метод `reject`.

Програмния код, който се изпълнява при инициализация на страницата е показан на Фиг. 9-11. От компонент “smart-select” извличаме идентификатора на избраната категория новини, а от хранилището получаваме URL към RSS канала за тази категория:

```
var newsCategory = $$('.news-category');
var newsCategoryId = newsCategory.prop('selectedIndex');
var newsCategoryUrl = $store.state.rssnews[newsCategoryId]['url'];
```

Следва викане на метод readNews на когото като аргумент се предава newsCategoryUrl. При успешно изпълнение на readNews се вика кода от `then`, а при грешка – кода от `catch`.

```
$on('pageInit', (e, page) => {
   $f7.dialog.preloader('Моля, изчакайте...');
   var newsCategory = $$('.news-category');
   var newsCategoryId = newsCategory.prop('selectedIndex');
   var newsCategoryUrl = $store.state.rssnews[newsCategoryId]['url'];

   readNews(newsCategoryUrl).then(function(items) {
    $store.state.newsdata = items;

    var searchbar = app.searchbar.create({                                  1
     el: '.searchbar',
     searchContainer: '.virtual-list-news',
     searchIn: '.item-title',
    });

    $$('.virtual-list-search').on('click', '.search', function() {          2
     let name = $$(this).attr('data-id');
     $$('#search-item-name').html(name.toUpperCase());
     searchbar.params.searchIn = '.item-' + name;
     app.swipeout.close('.swipeout', function() {} );
    });

    var searchVirtualList = app.virtualList.create({                        3
     . . .
    });

    var virtualListNews = app.virtualList.create({
     . . .
    });

    app.lazy.create('.virtual-list-news');                                  4
    $$('img.lazy').trigger('lazy');

   }).catch(function(error) {
    let status = '';
    if (error.status === 0) {
     status = 'Моля, разрешете достъпа до мрежата.';
    } else {
     status = "Мрежова грешка: " + error.statusText;
    }
    app.dialog.alert(status);

   }).finally(function() {
    app.dialog.close();
   });

  });
```

Фиг. 9-11 Инициализация на страницата - pageInit

При успешно изпълнение метод readNews връща обекта `items`, който съдържа мета данните за всички новини. Съдържанието му се буферира в хранилището – обект `news-data`. Следва създаване на компонент “serachbar” чрез който ще стане възможно филтриране на новините (секция 1). Източникът в който се търси е div контейнер от клас “virtual-list-news”. За да изградим потребителския интерфейс ще използваме два компонента от тип “virtual-list”. *Първият*, `searchVirtualList` (Секция 3) се използва с цел избор на ключ за търсене. Разрешени са следните ключове:

- Заглавие на новината – title.

- Подзаглавие на новината –subtitle.
- Дата на новината – date
- Описание на новината – info.

За по-лесен избор на ключ за търсене се използва компонент “swipeout”. За да види и избере ключ, потребителят трябва да изтегли с пръст надясно полето в което пише „Плъзнете с пръст надясно, за да изберете ключ за търсене”. По подразбиране се използва заглавието като ключ (виж Фиг. 9-12).

```
var searchVirtualList = app.virtualList.create({
 el: '.virtual-list-search',
 items: [{}],
 renderItem(item) {
  return `
    <li class="swipeout">
     <div class="swipeout-content swipeout-search">
      <a href="#" class="item-link item-content">
       <div class="item-inner">
        <div class="item-title-row">
         <div class="item-title">
          Плъзнете с пръст надясно, за да изберете ключ за търсене.
         </div>
         <div class="item-after" id="search-item-name">TITLE</div>
        </div>
       </div>
      </a>
     </div>
     <div class="swipeout-actions-left">
      <a href="#" class="color-green search" data-id="title">Заглавие</a>
      <a href="#" class="color-orange search" data-id="subtitle">Подзаглавие</a>
      <a href="#" class="color-yellow search" data-id="date">Дата</a>
      <a href="#" class="color-blue search" data-id="info">Текст</a>
     </div>
    </li>`;
  },
  height: $theme.ios ? 63 : ($theme.md ? 73 : 77),
});
```

Фиг. 9-12 Компонент за избор на ключ за търсене

При избор на ключ се активира кода от Секция 2 (виж Фиг. 9-11). Извлича се името на ключа чрез прочитане на стойността на атрибут “data-id”:

```
let name = $$(this).attr('data-id');
```

Следва промяна на ключа за търсене за компонент “searchbar”:

```
searchbar.params.searchIn = '.item-' + name;
```

Вторият компонент от тип “virtual list” - `virtualListNews` (Секция 3) се използва с цел визуализиране на мета данните за новините от избраната категория. Програмният код, който създава динамично този компонент е показан на Фиг. 9-13. Инициализацията на компонента изисква задаване на стойности за следните свойства:

- **el** – CSS селектор на контейнера, който съдържа виртуалния списък с новини.
- **items** – данни, чрез които ще се изгради динамично списъка.
- **searchAll** – функция, която реализира филтъра за новините. Методът трябва да върне масив (found) в който са записани индексите на новините, за които филтъра връща true. При конкретния пример чрез метод indexOf се проверява дали

съдържанието на новините, което отговаря на избрания ключ, съдържа текста въведен в полето за търсене.

- **renderItem** – метод, който динамично генерира HTML кода, който визуализира новините. Методът се вика за всеки елемент в масива items.

```
var virtualListNews = app.virtualList.create({
 el: '.virtual-list-news',
 items: items,
 searchAll: function(query, items) {
  var found = [];
  for (var i = 0; i < items.length; i++) {
   if (items[i].name.indexOf(query.trim()) >= 0) {
    found.push(i);
   }
  }
  return found;
 },
 renderItem(item) {
  return `
   <li class="swipeout">
    <div class="swipeout-content">
     <a href="/show/news/${item.newsid}/" class="item-link item-content news">
      <div class="item-media">
       <img class="lazy demo-lazy" data-src="${item.imageurl}" width="110"/>
      </div>
      <div class="item-inner">
       <div class="item-title-row">
        <div class="item-title"><b>${item.newsid+1}</b>. ${item.title}</div>
       </div>
       <div class="item-subtitle">${item.subtitle}</div>
       <div class="item-text item-date text-color-blue">${item.date}</div>
        <div class="item-text item-info">${item.info}</div>
      </div>
     </a>
    </div>
   </li>`;
  },
  height: $theme.ios ? 63 : ($theme.md ? 73 : 77),
 });
```

Фиг. 9-13 Компонент за визуализиране на новините от избрана категория

За всяка новина се визуализира следната информация:

- Изображение, свързано с новината – контейнер от клас “item-media”.
- Заглавието на новината – контейнер от клас “item-title”.
- Подзаглавие на новината – контейнер от клас “item-subtitle”.
- Дата на създаване на новината – контейнер от клас “item-date”.
- Описание на новината – контейнер от клас “item-info”.

За да намалим трафика с RSS каналите, изображенията се зареждат само ако попадат в изгледа на браузъра (**lazy images**). Програмна рамка Framework7 позволява работа с lazy компоненти, включително и изображения. За целта е необходимо да се зададе клас “lazy” за етикет img. Вместо атрибут “src”, трябва да се използва атрибут “data-src”. Освен това, при инициализация на приложението (файл app.js) трябва да зададете следните стойности за свойство `lazy`:

```
lazy: {
  sequential: false,
  observer: false,
},
```

Ако свойство `sequential` е true, то lazy изображения ще се зареждат едно по едно, когато се появят в изгледа. Ако свойство `observer` е true ще се използва Intersection Observer, ако се поддържа от браузъра. Приложният програмен интерфейс на Intersection Observer предоставя начин за асинхронно наблюдение на промените в пресичането на целеви елемент с други елементи или с изгледа на документа. Може да зададете стойност и на свойство `threshold`. По подразбиране изображенията се зареждат, когато се появят изцяло в изгледа. Използвайте свойство `threshold`, ако искате да заредите изображенията по-рано. Ако зададете стойност 25, изображението ще се зареди когато 25 пиксела от него се появят в изгледа. Имайте предвид, че lazy изображенията ще оптимизират трафика, но ще затруднят плавното превъртане на информацията във виртуалния списък. Тъй като виртуалният списък с новините се създава динамично, за да може DOM7 да отчете наличието на lazy компоненти е необходимо да се изпълни кода от секция 4. Той използва метод `app.lazy.create`, за да предизвика обновяване на DOM, както и симулиране на събитие lazy за всички изображения от клас lazy (метод `trigger`).

5.6 Визуализация на информацията за избрана новина

Визуализацията на избрана новина от списъка с новини се реализира чрез програмния код от документа-шаблон “show-news.html” (виж Фиг. 9-14).

```
<template>
  <div class="page">
    <div class="navbar">
      <div class="navbar-bg"></div>
      <div class="navbar-inner sliding">
        <div class="left">
          <a href="#" class="link back">
            <i class="icon icon-back"></i>
            <span class="md-only">Назад</span>
          </a>
        </div>
        <div class="title">${news.title}</div>
      </div>
    </div>
    <div class="page-content">
      <div id="news-container">
        <div class="card demo-news-card">
          <div class="card-header">
            <p class="text-color-red">${news.title}</p>
            <p class="text-color-gray">${news.date}</p>
          </div>
          <div class="card-content card-content-padding">
            <img src="${news.imageurl}" style="max-width:100%" />
            <p class="text-color-blue">${news.subtitle}</p>
            <p>${news.info}</p>
          </div>
          <div class="card-footer">
            <p class="text-color-gray">Автор: ${news.author}</p>
          </div>
        </div>
      </div>
    </div>
  </div>
</template>

<script>
  export default (props) => {
    const news = props.news;
    return $render;
  }
</script>
```

1

2

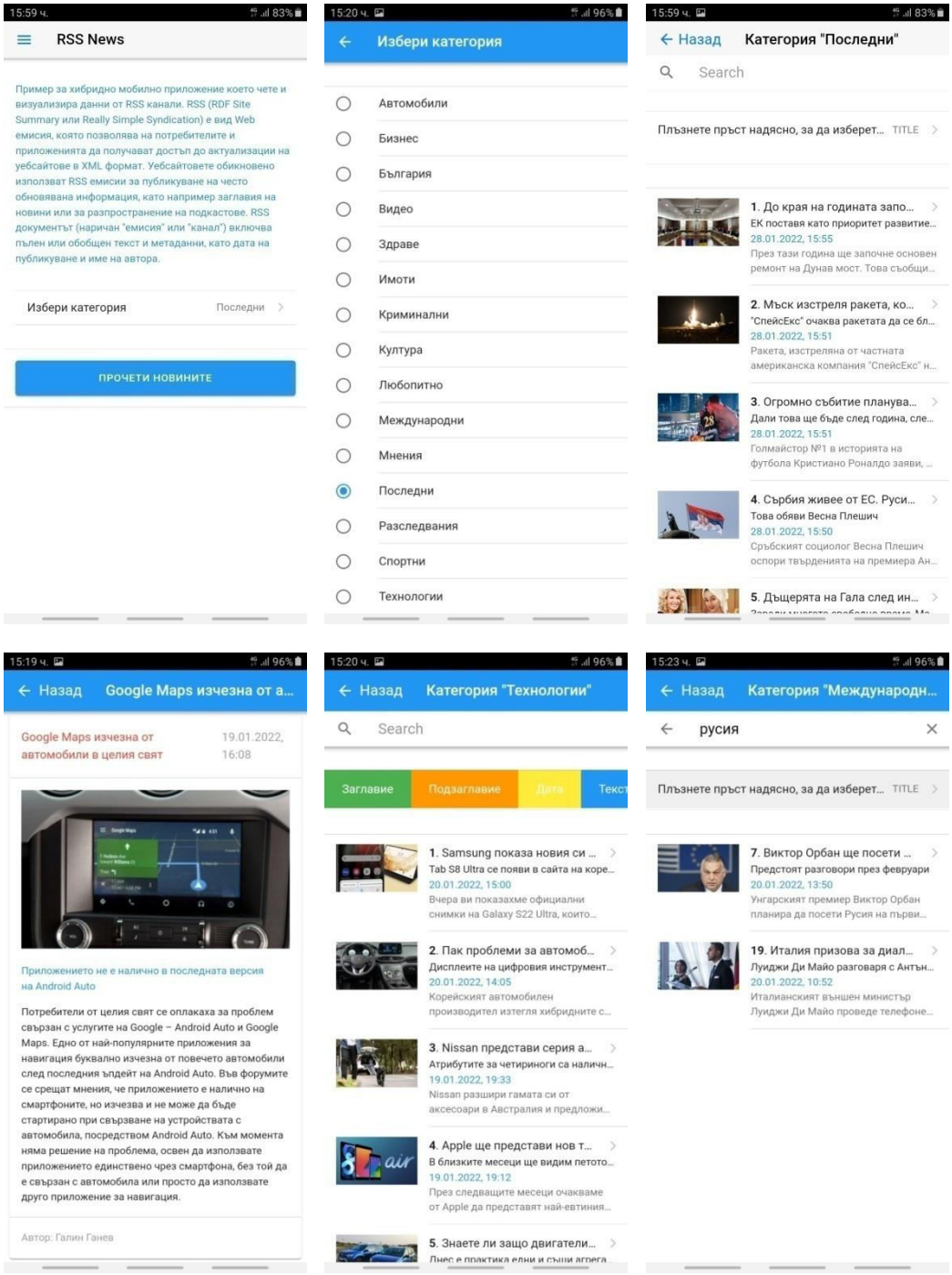
3

Фиг. 9-14 Визуализиране на избрана новина – файл “show-news.html”

Страницата получава обекта `news` (Секция 3), който съдържа мета данните на избраната новина (виж Фиг. 9-6). Заглавието на страницата съвпада със заглавието на новината (виж Секция 1). Мета данните за всяка новина се визуализират чрез компонент `card` (Секция 2). Контейнерът от клас “card-header” съдържа заглавието и датата на новината. Изображението, свързано с новината, подзаглавието и текстовото описание са в контейнер от клас “card-content”. Авторът на новината е в контейнер от клас “card-footer”.

VI. Тестване на приложението

Приложението се изгражда като Android проект. На Фиг. 9-15 са показани екрани, демонстриращи работата на приложението.



Фиг. 9-15 Резултати, получени след тестване на приложението

Отляво-надясно са показани следните екрани: начален екран, форма за избор на категория на новините, визуализиране на всички новини от избрана категория, визуализиране на избрана новина, избор на ключ за филтриране на новините и резултат след филтриране на новините.

Тъй като има последователно влагане на страници при тяхното визуализиране, за да не се изтрие кеша на предходните страници трябва при инициализация на приложението (виж файл app.js) да зададете стойност true за свойство `stackPages`:

```
view: {
  stackPages: true,
}
```

Заключение

В тази глава научихте какво представляват RSS каналите за новини, в какъв формат се разпространява тази информация и как може тя да бъде анализирана с възможностите на Framework7. Проектът към тази глава демонстрира работа с едни от най-използваните графични компоненти на Framework7 с цел изграждане на потребителския интерфейс. Запознахте се с това какво са lazy компоненти и как Framework7 зарежда lazy изображения. Затвърдена бе работата с документи-шаблони и използването на локалното хранилище на приложението.

Глава 10

Работа с метеорологични данни

I. Въведение

Съществуват множество Web услуги, които предлагат достъп до метеорологични данни. Тези данни се събират както чрез локални метеорологични станции, така и радарни станции и чрез метеорологични спътници (https://en.sat24.com). На базата на събраните

данни и математически модели може да се реализират краткосрочни и дългосрочни прогнози за времето. Освен за масовия потребител, тази информация е важна и за редица служби и организации (военни, пътна сигурност, национални предупреждения за времето и др.).

Всяка услуга, която предоставя метеорологични данни има своите предимства и недостатъци. В тази глава ще разгледаме проектирането и реализацията на мобилно приложение което информира потребителите за текущото време в избран от тях град, както и прогноза за следващите n на брой дни. Ще използваме приложните програмни интерфейси, които услугата *Open Weather Map* предоставя:

https://openweathermap.org/

Тази услуга позволява достъп до актуални данни за времето за всяко място, зададено чрез GPS координати, включително над 200 000 града. Потребителите получават информация за основни метеорологични параметри като температура, усещане за температурата, валежи, вероятност за валежи, влажност, атмосферно налягане, облачност, скорост на вятъра и др. Услугата предоставя и:

- История за 40 години с часова детайлност.
- Текущо състояние с актуализация на всеки 10 минути.
- Прогноза за следващия един час с детайлност по минути.
- Прогноза за 4 дни с точност до час.
- Прогноза за 16 дни, 4 пъти на ден - през нощта, през деня, вечерта и сутринта.
- Прогноза за 30 дни.
- Глобални карти на валежите въз основа на радарни данни, сателитни изображения и с помощта на машинно обучение.

Услугата Open Weather Map използва модела Openweather NWP (цифрово прогнозиране на времето). Този модел позволява да се изчисляват метеорологични данни за всяка локация. За целта се използва конволюционна невронна мрежа, която събира и обработва широк набор от източници на данни, за да покрие всяко място и да вземе предвид местните нюанси на климата. Технологията позволява да се постигне разделителна способност от около 500 m и точност между 90-100 %. Сред източниците на данни, които се използват, са 82 000 метеорологични станции; национални метеорологични агенции (NOAA, Environment Canada, Met Office и др.), радари, както и метеорологични спътници.

II. Дефиниране на стратегия

Ще създадем мобилно приложение с име Weather Forecast, което показва в един изглед информация за времето в момента на генериране на заявката, както и прогноза за следващите n на брой дни.

Данните за текущото време да бъдат следните:

- Температура.
- Усещане за температура.
- Атмосферно налягане.
- Влажност на въздуха.
- Скорост на вятъра.
- UV индекс.
- Текстово описание за времето (слънчево, облачно и др.).
- Икона, отговаряща на текстовото описание.

Прогнозните данни трябва да включват:

- Минимална и максимална стойност на температурата за деня.
- Текстово описание за времето.
- Икона, отговаряща на текстовото описание.

Ще реализираме задачата като използваме два от приложните програмни интерфейси на Open Weather Map:

- **One Call API** – информация за времето в момента на генериране на заявката и прогноза за следващите n на брой дни. Информацията се връща за локация, зададена чрез GPS координати.
- **Current Weather Data API** – информация за текущото време по зададено име на град или идентификатор на град. Това API освен информация за времето връща и GPS координатите на локацията.

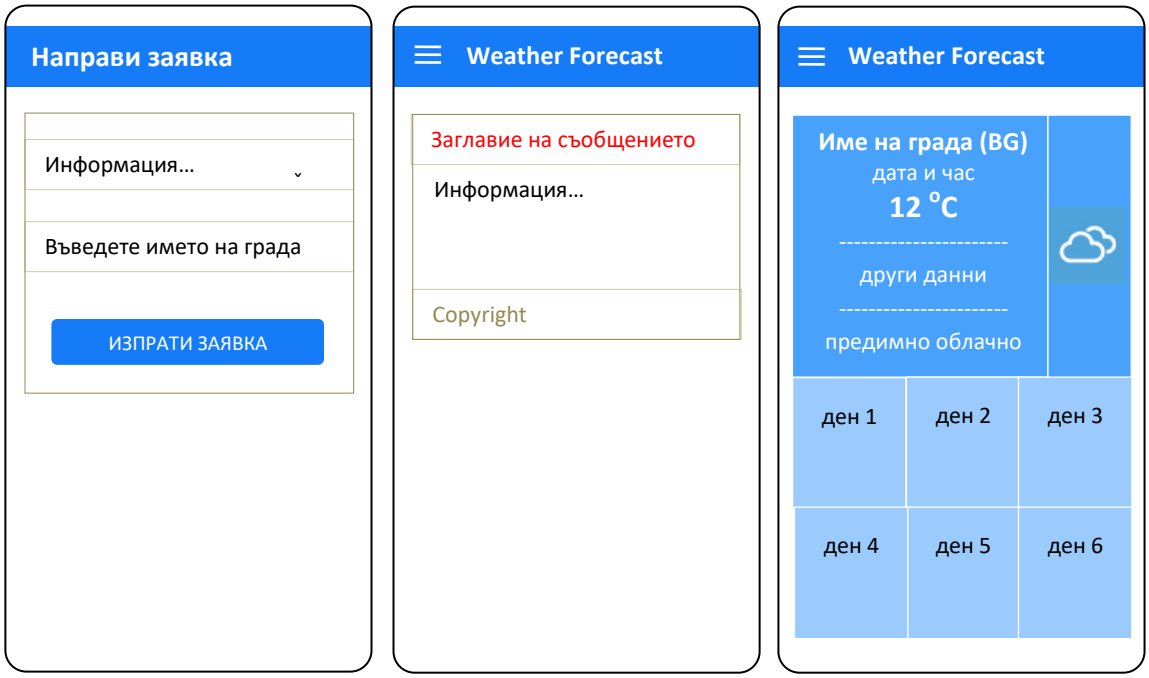
Ще използваме Current Weather Data API, за да получим GPS координатите на града, името на който е зададено от потребителя. Чрез тези координати ще направим заявка към One Call API, за да получим данните за текущото време и прогнозните данни.

III. Потребителски интерфейс

Ще използваме дизайн на потребителския интерфейс, който да позволява максимално бързо потребителят да получи желаната информация. За целта, приложението ще има само един изглед в който ще се показва: информация за метеорологичното време; информация за липса на мрежова свързаност и формата за въвеждане на името на град. Формата за въвеждане на името на града ще бъде визуализирана в левия панел на приложението. На Фиг. 10-1 е скициран желания потребителски интерфейс. Ще използваме графични компоненти само от програмна рамка Framework7, за да гарантираме нативния изглед на приложението.

Формата за въвеждане на името на града (виж Фиг. 10-1а) съдържа три компонента:

- Компонент „акордеон” чрез който се описва какво трябва да се въведе в полето за име на града.
- Компонент за въвждане на името на града.
- Бутон за изпращане на заявка към Open Weather Map.



Фиг. 10-1 Скициране на потребителския интерфейс

Заявката към Open Weather Map не трябва да може да се изпраща докато не се въведе валидно име за град. Валидността на името се проверява чрез *регулярен израз*. Ако липсва мрежова свързаност трябва да се визуализира информация за това. Всички съобщения за грешки се визуализират в компонент `card`, както е показано на Фиг. 10-1b. При успешно извличане на данните за времето информацията се представя във вида показан на Фиг. 10-1c. Данните за текущото време се визуализират на син фон с бели символи, а данните за прогнозата - на светлосин фон с черни символи. При конкретният пример прогнозните данни са за шест дни. Трябва да се предвиди възможност за пренареждане на прогнозната информация в зависимост от размерите на екрана (responsive дизайн).

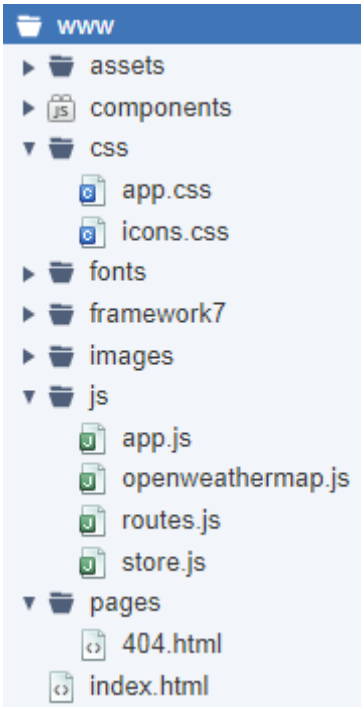
IV. Програмна реализация

Ще използваме версия 6.x на програмна рамка Framework7, за да реализираме приложението. За целта създайте празен проект (Blank Project) и прехвърлете в него съдържанието на папка www от готовия проект. Структурата на проекта “Weather Forecast” е показана на Фиг. 10-2.

Потребителският програмен код, който реализира проекта, е следния:

- Файл **index.html** – съдържа програмния код който изгражда главния изглед и левиа панел на приложението.
  - Папка **pages** – съдържа един документ - файл **404.html**. Съдържанието на този документ се визуализира при заявка, за която липсва маршрут.
- Папка **js** – съдържа 4 JavaScript файла, които се използват както следва:
  - Файл **app.js** – съдържа програмния код за инициализация на приложението.
  - Файл **routes.js** – създава масив routes чрез който се описват всички маршрути.
  - Файл **store.js** – съдържа програмния код чрез който се създава локално хранилище в което ще запишем данните, необходими за функциониране на приложението.

- Файл **openweathermap.js** – съдържа функции за достъп до One Call API и Current Weather Data API, както и програмния код за парсване на отговора и динамично генериране на изгледа с резултата.



Фиг. 10-2 Структура на проект “Weather Forecast”

4.1. Използвани API от Open Weather Map

Информацията за текущото време и за прогнозата се получава чрез **One Call API**. Чрез една заявка получаваме достъп до следните данни:

- Текущо време във формат Unix timestamp.
- Текущо време и прогнозни данни за времето.
- Минутна прогноза за 1 час.
- Почасова прогноза за 48 часа.
- Дневна прогноза за 7 дни.
- Национални предупреждения за времето.
- Исторически данни за времето за предходните 5 дни в JSON формат.

Форматът на заявката е следния:

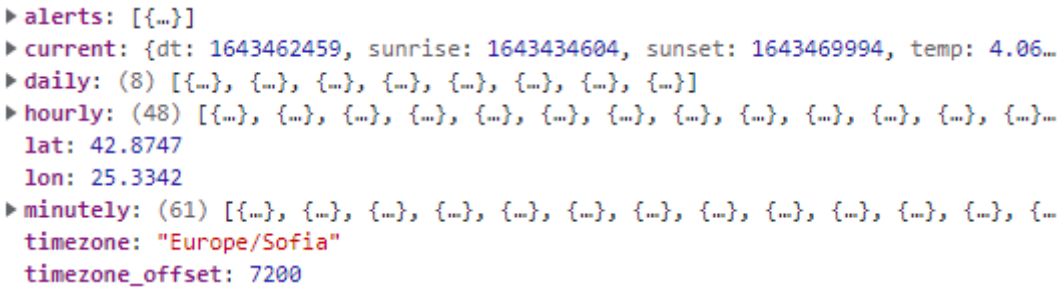
https://api.openweathermap.org/data/2.5/onecall

Параметрите към ресурса onecall са следните:

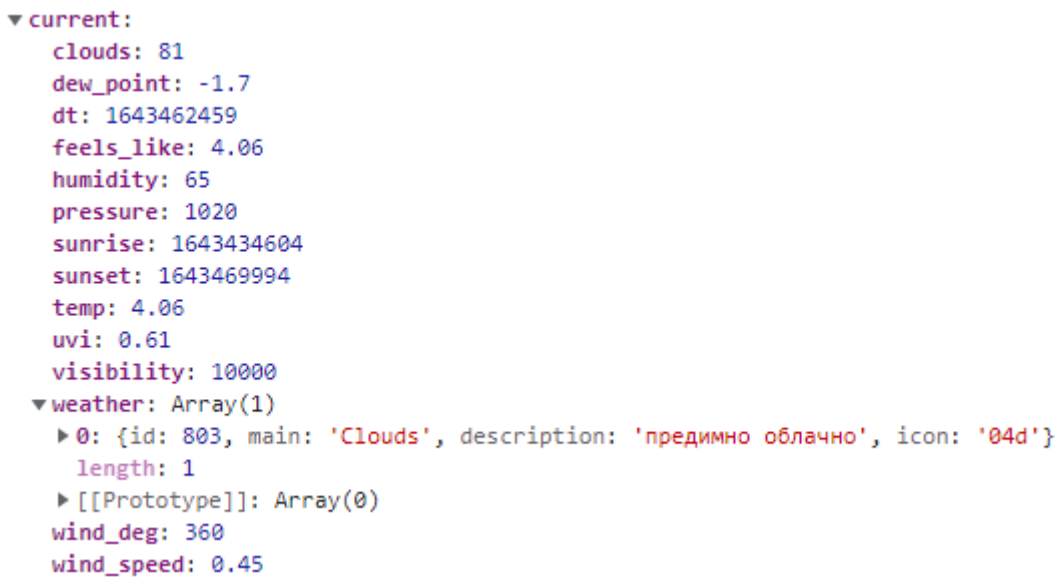
- **lat**, **lon** – GPS локация (географска ширина и дължина).
- **lang** – двубуквен код за езика, чрез който се обменя информация с API (заявки и отговори), например bg.
- **units** – мерни единици. Налични са стандартни, метрични и имперски единици. Изберете metric, за да използвате използваната в Европа метрична система.
- **exclude** - чрез този параметър можете да изключите някои части от метеорологичните данни от отговора. Така се оптимизира трафика със сървъра. Това трябва да бъде низ, който съдържа типовете данни, разделени със запетая (без интервали).

- appid - вашият уникален API ключ за достъп до услугата.

Отговорът, който One Call API връща, е в JSON формат по подразбиране (виж Фиг. 10-3). Информацията, която представлява интерес за приложението е в стойността на свойство `current` (виж Фиг. 10-4).



Фиг. 10-3 Отговор при заявка към One Call API



Фиг. 10-4 Съдържание на свойство current

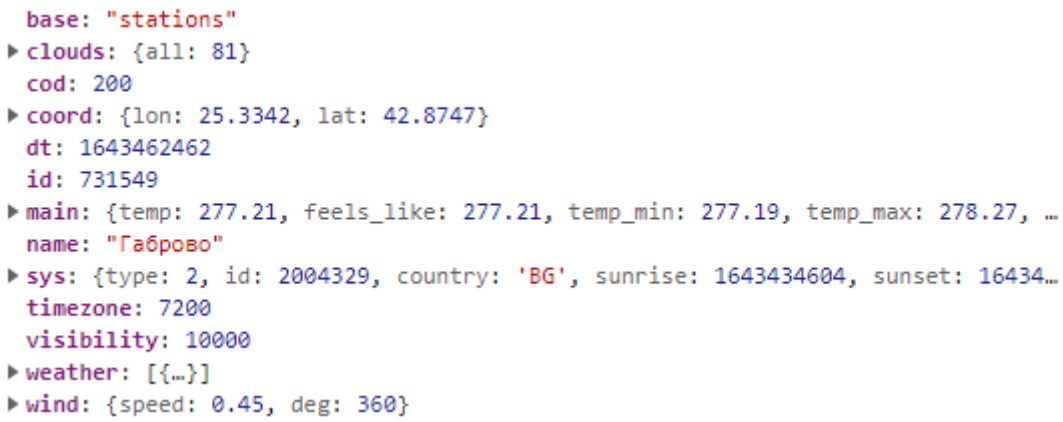
За да използваме One Call API е необходимо да знаем GPS координатите на града за който искаме информация. Ще използваме **Current Weather Data API**, за да получим тази информация. Заявката трябва да бъде до ресурс weather:

https://api.openweathermap.org/data/2.5/weather

Параметрите към ресурса weather са следните:

- **q** - име на града, код на щата (окръга) и код на страната, разделени със запетая. Кодът на окръга и държавата не са задължителни. Те се използват когато има множество градове с едно и също име в различни държави или в рамките на един щат (окръг). Кодовете на държавите са съгласно стандарта ISO 3166.
- **lang** – двубуквен код за езика, чрез който се обменя информация с API (заявки и отговори).
- **units** – мерни единици.
- **appid** - вашият уникален API ключ за достъп до услугата.

Отговорът на тази заявка съдържа информация за времето и GPS координатите на населеното място (виж Фиг. 10-5). Информацията, която ни трябва е стойността на свойство `coord`.



Фиг. 10-5 Отговор при заявка към Current Weather Data API

4.2. Индексен файл

Индексният файл съдържа HTML кода за главния изглед и левия панел на приложението. Левият панел се използва за създаване на форма чрез която се изпраща заявка към Web услугата. Програмният код е показан на Фиг. 10-6. Формата се състои от три компонента:

- Помощна информация за потребителя, който не знае в какъв формат може да опише името на локацията. Тъй като тази информация не е необходимо да се вижда постоянно се използва компонент от тип „акордеон” (Секция 1). При този компонент се вижда само текста от контейнера от клас “item-inner”. Когато потребителят избере този текст се показва текста от контейнер от клас “accordion-item-content”.
- За въвеждане на името на града се използва `input` етикет (виж Секция 2). Използван е атрибут `validate`, за да се активира валидиране на въведения текст. Валидирането се реализира чрез регулярен израз – атрибут `pattern`. Конкретният регулярен израз разрешава името на града да е съставено от малки и главни букви на кирилица и латиница, както и един или повече интервали, запетая, тире и апостроф. Когато потребителят не спазва този формат се визуализира текста от атрибут “data-error-message”.
- Изпращането на заявката се реализира чрез натискане на бутон „Изпрати заявка”. Този бутон по подразбиране е забранен за натискане – атрибут `disabled`. Целта е да се предотврати натискане на бутона, ако въведената информация не е в желания формат. Всеки път, когато полето за въвеждане на име на град получи фокус се изчиства неговото старо съдържание. Целта е потребителят да може да въвежда името на града чрез глас (използва се интегрираната система за разпознаване на глас на Android или iOS). Когато полето загуби фокус се проверява дали въведеното име е валидно - само тогава се разрешава натискането на бутона за изпращане на заявката.

При натискане на бутон „Изпрати заявка” се активира метод **getWeatherInfo**. Веднага след това левият панел трябва да се затвори. Този функционалност се реализира чрез атрибут “panel-close” (виж кода от Секция 3).

```
<div class="panel panel-left panel-reveal panel-init">
 <div class="view view-left">
  <div class="page">
   <div class="navbar">
    <div class="navbar-bg"></div>
    <div class="navbar-inner sliding">
     <div class="title">Направи заявка</div>
    </div>
   </div>
   <div class="page-content menu">
    <div class="block menu-container">
```

1

```
<div class="list">
 <ul>
  <li class="accordion-item">
   <a href="#" class="item-content item-link">
    <div class="item-inner">Информация...</div>
   </a>
   <div class="accordion-item-content">
    <div class="block" id="object-moreinfo">
     <p class="text-color-blue">
      Въведете името на града за който искате да получите информация …
     </p>
    </div>
   </div>
  </li>
 </ul>
</div>
```

2

```
<div class="list">
 <ul>
  <li>
   <div class="item-content item-input">
    <div class="item-inner">
     <div class="item-input-wrap">
      <input type="text" name="city" value=""
             placeholder="Въведете името на град" required validate
             pattern="^[A-Za-zA-Яа-я',.\s-]{2,}$"
             data-error-message="Само букви и интервали!" />
     </div>
    </div>
   </div>
  </li>
 </ul>
</div>
```

3

```
<div class="block">
 <a href="#" class="item-link col button button-fill button-raised
  submit panel-close disabled" onclick="getWeatherInfo()">
   Изпрати заявка
 </a>
</div>
```

```
    </div>
   </div>
  </div>
 </div>
</div>
```

Фиг. 10-6 Индексен файл – програмен код за левия панел

На Фиг. 10-7 е показан програмния код, който визуализира основния изглед на приложението. В даден момент от време се показва съдържанието на един от следните два контейнера:

- Контейнер от клас “warning-container” (Секция 1) – чрез компонент от тип card се визуализира информация когато липсва мрежова свързаност.
- Контейнер от клас “main-container” (Секция 2) – визуализира информация за времето в избрания от потребителя град.

```
<div class="page">
 <div class="page-content results">
  <div class="block main">

   <div class="warning-container">                                    1
    <div class="card demo-warning-card">
     <div class="card-header">
      <p class="text-color-red">
       <b>Разрешете мрежовата свързаност</b>
      </p>
     </div>
     <div class="card-content card-content-padding">
      <p class="text-color-black">
       Приложението визуализира информация за метеорологичното време
       в избран от потребителя град. Данните за времето се получават
       чрез Web услугата Open Weather Map. Поради тази причина е
       необходимо да разрешите достъпа до Internet.
      </p>
     </div>
     <div class="card-footer">
      <p class="text-color-gray">&copy; RS SOFT 2022</p>
     </div>
    </div>
   </div>

   <div class="main-container padding-no">                            2
    <div class="block grid-weather">
     <div class="row align-items-stretch no-gap current-results">
     </div>

     <div class="row align-items-stretch no-gap forecast-results">
     </div>
    </div>

    <div class="block copyright">
     <p class="text-color-gray">
      &copy; Open Weather Map
     </p>
    </div>
   </div>

  </div>
 </div>
</div>
```

Фиг. 10-7 Индексен файл – програмен код за основния изглед

Информацията за времето се вмъква динамично. Данните за времето в текущия момент от време се вмъкват в контейнер от клас “current-results”, а данните за прогнозата – в контейнер от клас “forecast-results”.

4.3. Хранилище

Ще използваме хранилището на приложението за буфериране на следната информация (виж Фиг. 10-8):

- Мрежова свързаност (**networkStatus**) – стойност false означава, че липсва мрежова свързаност. За да се получи тази информация се използва се плъгина “Network Information”.
- Чакаща обработка заявка (**pendingRequest**) – стойност true означава, че има заявка към услугата Open Weather Map, която не е изпълнена (например поради липса на мрежова свързаност).
- Име на последно избрания град (**cityName**) – когато се въведе нов валиден град, неговата стойност се запомна в хранилището на приложението и в *local store* на браузъра. По този начин след стартиране на приложението, то работи с името на последно избрания от клиента град – чете се от local store и се записва в хранилището на приложението.
- Брой дни за предсказаване на времето – **numberOfForecasts**.
- Мерни единици за всяка от мета данните за времето – **weatherUnits**.
- Икони, които показват графично информация за времето – **weatherIcons**. Това поле описва връзката между идентификатора на иконата и името на графичен файл. Тези файлове са в SVG формат (векторна графика).

```
var createStore = Framework7.createStore;
const store = createStore({
 state: {
  networkStatus: false,
  pendingRequest: true,
  numberOfForecasts: 6,
  cityName: 'София',
  weatherUnits: {
   temp: '&deg;C',
   humidity: '%',
   visibility: 'm',
   windSpeed: 'm/s',
   uvi: '',
   pressure: 'hPa',
  },
  weatherIcons: {
   '01d': 'sunny',
   . . .
   '50n': 'fog',
  },
 },
});
```

Фиг. 10-8 Хранилище на приложението – файл store.js

4.4. Маршрутизатор

Информацията за маршрутизатора на Framework7 е показана на Фиг. 10-9.

```
routes = [
  {
    path: '/',
    url: './index.html',
  },
  {
    path: '/menu/',
    url: './pages/menu.html',
  },
  // Default route (404 page). MUST BE THE LAST
  {
    path: '(.*)',
    url: './pages/404.html',
  },
];
```

Фиг. 10-9 Маршрутизация – файл routes.js

4.5. Инициализация

Програмния код за инициализация на приложението, както и необходимите помощни методи са във файл **app.js**. На Фиг. 10-10 е показан програмния код чрез който се създава обект `app` за достъп до рамката Framework7, както и създаването на обектите за главния изглед (mainView) и левия панел (leftView).

```
var $$ = Dom7;
var app = new Framework7({
  el: '#app',                       // App root element
  id: 'bg.tugab.kst.forecast',      // App bundle ID
  name: 'Weather Forecast',         // App name
  theme: 'auto',                    // Automatic theme detection
  store: store,
  routes: routes,
});
// Init/Create main view
var mainView = app.views.create('.view-main', {
  url: '/'
});
var leftView = app.views.create('.view-left', {
  url: '/'
});
```

Фиг. 10-10 Инициализация – създаване на обект app и необходимите изгледи

Приложението получава информация кога се губи мрежова свързаност и кога тя се възстановява. За целта се прехващат събития “offline” и “online” (виж Фиг. 10-11).

```
// Loss of network connectivity
$$(document).on('offline', () => {
  store.state.networkStatus = false;
  if (store.state.pendingRequest) {
    $$('.warning-container').show();
  }
  else {
    notificationOffline.open();
  }
});
// Network connection restored
$$(document).on('online', () => {
  store.state.networkStatus = true;
  if (store.state.pendingRequest) {
    getWeatherInfo();
  }
});
```

Фиг. 10-11 Инициализация – анализ на мрежовата свързаност

При загуба на мрежова свързаност на променлива **networkStatus** се задава стойност false. Ако има необслужена заявка се показва съдържанието на контейнера от клас “warning-container”. В противен случай се показа прозорец с нотификация, че има загуба на мрежова свързаност.

При възобновяване на мрежовата свързаност на променлива networkStatus се задава стойност true и ако има необслужена заявка се вика метод **getWeatherInfo**. Този метод реализира необходимите заявки към Open Weather Map, парсва получените мета данни за времето и визуализира по желания начин тази информация.

За да разберем кога потребителят използва формата за въвеждане на име на град ще прехванем събития “focusin” и “focusout” за обект, свързан с input етикет с атрибут “name” равен на “city” (виж Фиг. 10-12).

```
// Clear current city name
$$('[name="city"]').on('focusin', function () {
  $$('[name="city"]').val('');
});
// Check if a new city name has been entered
$$('[name="city"]').on('focusout', function () {
  let cityName = $$('[name="city"]').val();
  if (cityName.match(/^[A-Za-zA-Яа-я,\s]{2,}$/)) {
    if (cityName !== store.state.cityName) {
      // Enable send request button
      if (store.state.networkStatus) {
        $$('.submit').removeClass('disabled');
      }
      else {
        store.state.pendingRequest = true;
        app.dialog.alert('Моля, разрешете мрежовата свързаност!');
      }
    }
  }
  else {
    // Restore last valid city name
    $$('[name="city"]').val(store.state.cityName);
  }
});
```

Фиг. 10-12 Инициализация – събития от формата за въвеждане на име на град

Когато input полето получи фокус се изчиства въведеното до този момент съдържание. За целта се използва метод `val`. Идеята е да се даде възможност за въвеждане на името на града чрез разпознаване на говор. Когато input полето губи фокус се предполага, че потребителя е въвел името на нов град. За целта се извлича новото съдържание на полето – `cityName`. Следва проверка чрез регулярен израз дали въведеното име отговаря на желания за име на град формат. Продължава се с разрешаване на натискането на бутон “Изпрати заявка” ако: 1) Новото име е с валиден формат; 2) Новото име е различно от старото и 3) Има мрежова свързаност. Ако тези условия не са изпълнение се установява флаг `pendingRequests`. Ако новото име не е валидно се възстановява името на последно избрания град.

Остава да скрием двата контейнера от главния изглед и да проверим дали в текущия момент има налична мрежова свързаност. В зависимост от това дали има мрежова свързаност се симулира генериране на събитие “offline” или “online” чрез използване на метод `trigger` (виж Фиг. 10-13).

```
// Hide containers for result
$$('.warning-container').hide();
$$('.main-container').hide();
// Check network status
var connStatus = navigator.connection.type;
if (connStatus === Connection.NONE) {
  $$(document).trigger('offline');
}
else {
  $$(document).trigger('online');
}
```

Фиг. 10-13 Инициализация – начална логика

Проверката за наличие на мрежова свързаност се реализира чрез получаване на типа на връзката. Мрежовата свързаност не е налична, ако стойността за типа на мрежовата връзка е `Connection.NONE`.

Файлът app.js съдържа и няколко помощни методи. Това са методи за запис на данните от формата (на този етап само името на града) в local store, както и за четене на тези данни (виж Фиг. 10-14).

```
function saveMenuData() {
 let cityName = $$('[name="city"]').val();
 store.state.cityName = cityName;
 var menuData = {
  city: store.state.cityName,
 };
 localStorage.removeItem("menu");
 localStorage.setItem("menu", JSON.stringify(menuData));
}
// ------------------------------------------------------------
function getMenuData() {
 var menuInfo = localStorage.getItem("menu");
 if (menuInfo !== 'undefined' && menuInfo !== null) {
  return JSON.parse(menuInfo);
 }
 else {
  var menuData = {
   city: store.state.cityName,
  };
  return menuData;
 }
}
```

Фиг. 1-14 Инициализация – работа с local store

За да бъде програмният код лесно преизползваем ще запишем данните под формата на JSON обект. На този етап `menuData` ще съдържа само свойство “city”. В бъдеще, ако се наложи да се помнят и други данни, промените ще се реализират много лесно. Записът в local store се реализира чрез метод `setItem`, а четенето – чрез метод `getItem`. Ако в local store няма запис се симулира стойността на `menuData` със стойност за `city`, стойността записана в хранилището на приложението (`store.state.cityName`).

Тъй като Open Weather Map използва Unix timestamp, за да опише датата и часа на генериране на мета данните за времето, се налага тяхното конвертиране до година, месец, ден, час, минути и секунди. Това се реализира чрез метод **getDateFromTimestamp**. Програмният код на този метод е показан на Фиг. 10-15. Стойността на параметър `timestamp` се умножава по 1000, за да се преобразува от секунди до ms. Това е необходимо, за да използваме конструктора на клас `Date`. Чрез него създаваме обект `now`, който описва датата и часа в желания от нас формат. Следва извличане на необходимите параметри чрез методите, които клас `Date` предоставя. Всички те се записват в JSON обект `result`, който функцията връща като отговор. Програмният код, който използва функция getDateFromTimestamp може да извлече необходимите му полета от отговора, който функцията връща, например:

```
let date = getDateFromTimestamp(timestamp);
let year = date.year;
```

```
function getDateFromTimestamp(timestamp) {
  var months = ['януари', 'февруари', 'март', 'април', 'май', 'юни', 'юли',
                'август', 'септември', 'октомври', 'ноември', 'декември'];
  var dayOfWeek = ['Нд', 'Пн', 'Вт', 'Ср', 'Чт', 'Пт', 'Сб'];
  var now = new Date(timestamp * 1000);
  var day = now.getDate();
  var dayName = dayOfWeek[now.getDay()];
  var month = months[now.getMonth()];
  var year = now.getFullYear();
  var hours = now.getHours().toString().padStart(2, '0');;
  var minutes = now.getMinutes().toString().padStart(2, '0');;
  var result = {
    year: year,
    month: month,
    day: day,
    dayName: dayName,
    hours: hours,
    minutes: minutes,
  };
  return result;
}
```

Фиг. 10-15 Инициализация – метод getDateFromTimestamp

4.6. Комуникация с Open Weather Map

Всички функции, чрез които се реализира извличане на информация за времето от Open Weather Map, парсване на съдържанието на отговора и визуализиране на мета данните за времето се намират във файл **openweathrmap.js**. Да припомним, че ще използваме два приложни програмни интерфейса от Open Weather Map, за да реализираме приложението. *Първо*, чрез Current Weather Data API ще получим GPS координатите на града по зададено име на града. *Второ*, чрез One Call API ще получим всички необходими мета данни за времето при зададени GPS координати на населеното място. На Фиг. 10-16 е показан програмния код на функция **getCityLocation** чрез която се получават GPS координатите на града.

```
function getCityLocation() {
  var cityName = $$('[name="city"]').val();
  var promise = new Promise(function (resolve, reject) {
    let url = 'https://api.openweathermap.org/data/2.5/weather';
    app.request({
      url: url,
      type: 'GET',
      timeout: 6000,
      dataType: 'json',
      data: {
        appid: 'Вашият ключ за достъп до услугата',
        q: cityName,
        lang: 'bg',
      },
      success: function (data) {
        resolve(data);
      },
      error: function (error) {
        reject(error);
      }
    });
  });
  return promise;
}
```

Фиг. 10-16 Функция getCityLocation

За да използвате услугата, трябва да получите ключ за достъп. Задайте като стойност на поле `appid` така получения ключ.

Извличането на данните за времето се реализира чрез функция **readWeatherInfo** (виж Фиг. 10-17). Като входни данни се задава GPS локацията на града. За да филтрираме отговора, използваме поле exclude чрез което описваме кои части от отговора не желаем да получаваме. В случая не се нуждаем от данните от секции minutely, hourly и alerts.

```
function readWeaterInfo(lat, lon) {
 var promise = new Promise(function (resolve, reject) {
  let url = 'https://api.openweathermap.org/data/2.5/onecall';
  app.request({
   url: url,
   type: 'GET',
   timeout: 6000,
   dataType: 'json',
   data: {
    appid: 'Вашият ключ за достъп до услугата',
    lon: lon,
    lat: lat,
    lang: 'bg',
    units: 'metric',
    exclude: 'minutely,hourly,alerts'
   },
   success: function (data) {
    resolve(data);
   },
   error: function (error) {
    reject(error);
   }
  });
 });
 return promise;
}
```

Фиг. 10-17 Функция readWeatherInfo

Базовата функционалност на приложението се реализира чрез програмния код от функция **getWeatherInfo**. Тази функция изпълнява следната последователност от действия:

- Получава името на града, кода на държавата (два символа) и GPS локацията на града чрез функция getCityLocation.
- Получава мета данните за времето чрез функция readWeatherInfo.
- Вмъква динамично информация за текущото време в контейнер от клас “current-results”.
- Вмъква динамично информация за прогнозата за времето в контейнер от клас “forecast-results”.
- Визуализира динамично генерираните данни (контейнер от клас “main-container”).

Мета данните за текущото време се получават от секция `current` от отговора, който функция readWeatherInfo връща (обект `data`). Програмният код (HTML), който визуализира тези данни се генерира динамично и записва в променлива `result`. Този код е показан на Фиг. 10-18.

Мета данните за прогнозата за времето се получават от секция `dayly` от отговора, който функция readWeatherInfo връща. Програмният код (HTML), който визуализира тези данни се генерира динамично и записва в променлива result. Този код е показан на Фиг. 10-19.

```
function getWeatherInfo() {
 $$('.submit').addClass('disabled');          // Disable send request button
 getCityLocation().then(function (data) {     // Send a new request
  saveMenuData();                             // Save new city into local store
  let cityName = data.name;
  let country = data.sys.country;
  let location = data.coord;
  let lat = location.lat;
  let lon = location.lon;

  readWeaterInfo(lat, lon).then(function (data) {
   // Get metadata for current weather
   let current = data.current;
   let temperature = round(current.temp);
   let temperatureFeelsLike = round(current.feels_like);
   let pressure = current.pressure;
   let humidity = current.humidity;
   let uvi = round(current.uvi);
   let windSpeed = round(current.wind_speed);
   let weatherDesc = current.weather[0].description;
   let iconId = current.weather[0].icon;
   let timestamp = current.dt;
   let now = getDateFromTimestamp(timestamp);
   let date=`${now.dayName} ${now.day} ${now.month}, ${now.hours}:${now.minutes}`;
   let iconName = store.state.weatherIcons[iconId];
   if (iconName === undefined) {
    iconName = 'none';
   }
   let result = `Динамично генериране на информация за текущото време`;
   $$('.current-results').html(result);
   // Get metadata for weather forecast
   $$('.forecast-results').html('');
   for (let i = 0; i < store.state.numberOfForecasts; i++) {
    let daily = data.daily[i];
    let tempMin = round(daily.temp.min, 1);
    let tempMax = round(daily.temp.max, 1);
    let description = daily.weather[0].description;
    let iconId = daily.weather[0].icon;
    let timestamp = daily.dt;
    let now = getDateFromTimestamp(timestamp);
    let date = `${now.dayName} ${now.day} ${now.month}`;
    let iconName = store.state.weatherIcons[iconId];
    let temperature = `${tempMin} / ${tempMax} ${store.state.weatherUnits.temp}`;
    let iconPath = `images/${iconName}.svg`;
    let result = `Динамично генериране на информация за прогнозата`;
    $$('.forecast-results').append(result);
    store.state.pendingRequest = false;
   }
   // Show container with results and hide warning container
   $$('.warning-container').hide();
   $$('.main-container').show();
  }).catch(function (error) {
   app.dialog.alert(error);
  });
 }).catch(function (error) {
  if (error.status === 404) {
   // Restore last valid city name
   $$('[name="city"]').val(store.state.cityName);
   app.dialog.alert('Името на града не е в базата данни!');
  }
  else {
   app.dialog.alert('Не може да се получи информация за времето.
        Опитайте отново по-късно.');
  }
 });
}
```

Фиг. 10-18 Функция getWeatherInfo

```
<div class="col-75 weather col-center-content">
 <div class="block no-margin">
  <p class="text-color-white city-name"><b>${cityName}</b>(${country})</p>
  <p class="text-color-white current-date">${date}</p>
  <p class="text-color-white temperature">
   ${temperature} ${store.state.weatherUnits.temp}
  </p>
  <p class="text-color-white temperature-feels-like">
   Усеща се като: ${temperatureFeelsLike} ${store.state.weatherUnits.temp}
  </p>
  <hr>
  <p class="text-color-white current-uvi">
   Атм. налягане: ${pressure} ${store.state.weatherUnits.pressure}
  </p>
  <p class="text-color-white current-uvi">
   Влажност: ${humidity}${store.state.weatherUnits.humidity}
  </p>
  <p class="text-color-white current-uvi">
   Вятър: ${windSpeed} ${store.state.weatherUnits.windSpeed}
  </p>
  <p class="text-color-white current-uvi">UV индекс: ${uvi}</p>
  <hr>
  <p class="text-color-white description">${weatherDesc}</p>
 </div>
</div>

<div class="col-25 weather col-center-content">
 <div class="block">
  <img src="images/${iconName}.svg" class="current-weater-icon" />
 </div>
</div>
```

Фиг. 10-19 HTML код за текущото време

```
<div class="col-33 medium-15 forecast">
 <div class="block forecast-container no-margin">
  <p class="text-color-black forecast-date">${date}</p>
  <img src="${iconPath}" class="forecast-icon" />
  <p class="text-color-black forecast-temp">${temperature}</p>
  <p class="text-color-black forecast-description">${description}</p>
 </div>
</div>
```

Фиг. 10-20 HTML код за прогнозата за времето

Използвани са възможностите на Framework7 за responsive дизайн с цел визуализиране на данните, свързани с прогнозата на времето (виж Фиг. 10-20). Тъй като информацията се визуализира в редове и колони (grid) използваме атрибути “col-33” и “medium-15”, за да реализираме желания адаптивен към ширината на екрана дизайн. При ширина на екрана до 768 пиксела на един ред се визуализира информация за 3 дни (col-33). В противен случай на един ред се визуализира информацията за 6 дни (medium-15).

4.7. Стилово форматиране

Стиловото форматиране се използва основно с цел позициониране и задаване на цветовете чрез които се визуализира текущото време и прогнозата за времето. Всички CSS правила са във файл **app.css** от папка css (виж Фиг. 10-21).

За да направим така, че височината на всички колони от един ред да са еднакви се използва атрибут “align-items-stretch” за редовете и следното CSS правило за колоните:

```
.grid-weather .col-center-content {
   display: flex;
   align-items: center;
   justify-content: center;
}
```

Информацията за текущото време се визуализира с бели символи на син фон, а информацията за прогнозата – с черни символи на светло син фон.

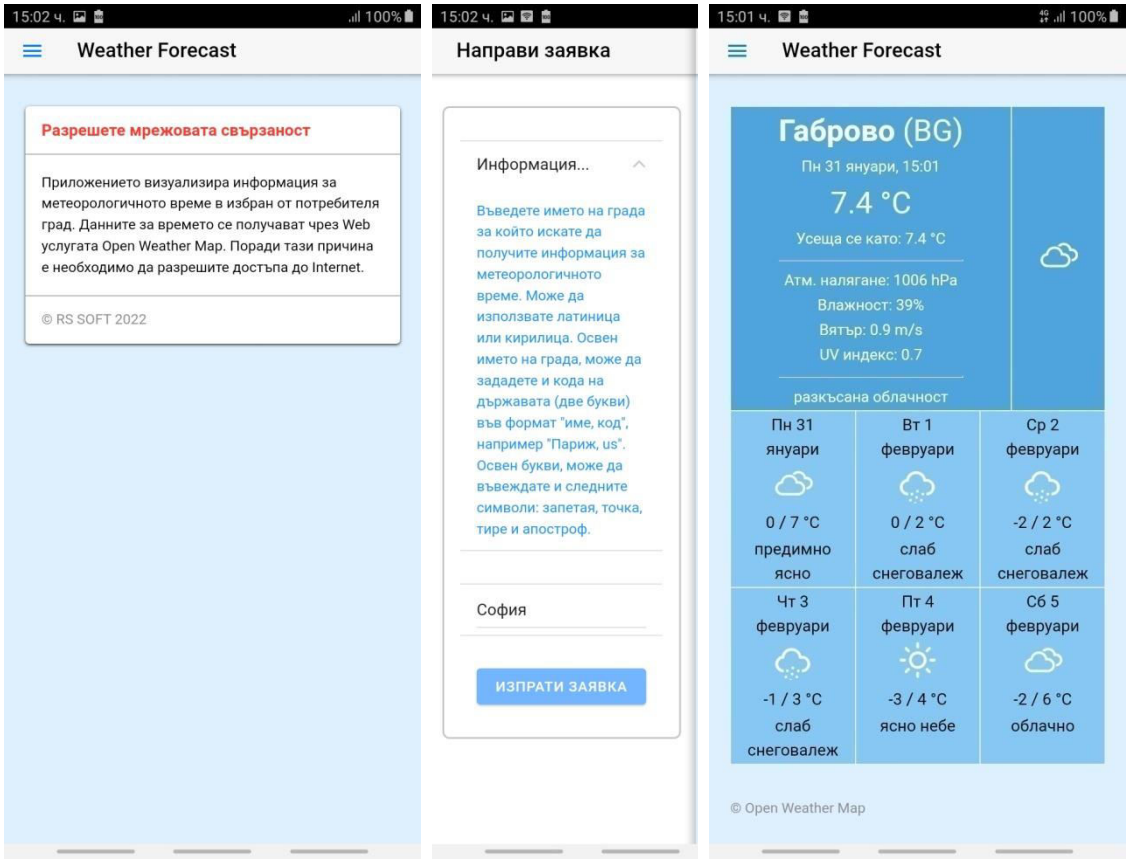
```
p {
   margin: 0;
   padding: 1px 0;
}
.city-name, .temperature {
   font-size: 200%;
}
div[class*="col-"] {
   background: #4fa4dd;
   text-align: center;
   border: 1px solid rgb(240, 240, 240);
   font-size: 15px;
}
div[class*="forecast"] {
   background: #86c8f3;
   font-size: 16px;
}
.grid-weather .col-center-content {
   display: flex;
   align-items: center;
   justify-content: center;
}
img.current-weater-icon {
   width: 100%;
}
.page-content.menu {
   margin-left: 10px;
   margin-right: 15px;
}
.menu-container {
   border: 2px solid rgb(196, 196, 196);
   border-radius: 7px;
}
.block.main {
   padding: 0 5px;
}
.page-content.results {
   background: #def0fd;
```

Фиг. 10-21 Стилово форматиране – файл app.css

4.8. Изграждане и тестване на приложението

Преди изграждането на приложението трябва да добавите плъгин “Network Information” към проекта. Чрез този плъгин приложението ще получава информация кога липсва мрежова свързаност и кога тя е налична.

На Фиг. 10-22 са показани екрани, които показват как функционира приложението при липса и наличие на мрежова свързаност.



a) b) c)

Фиг. 10-22 Тестване на приложението: a) Лисва мрежова свързаност; b) Ляв панел за въвеждане на име на града и c) Информация за текущото време и прогноза за 6 дни

Заключение

В тази глава се запознахте с това как може да получите информация за метеорологичното време от облачно базирани услуги. Проектът от тази глава използва услуга Open Weather Map с цел получаване на моментните параметри на времето, както и прогноза за следващите 6 дни. Проектът е пример за използване на асинхронни комуникации с услуги, както и как потребителският интерфейс да се адаптира към наличието или липсата на мрежова свързаност. Разбрахте и как се реализира responsive дизайн чрез използване на компонент grid от Framework7.

Глава 11

Прогресивни Web приложения

I. Въведение

Прогресивните Web приложения (PWA) са вид приложен софтуер, създаден с помощта на Web технологии като HTML5, CSS3, JavaScript и WebAssembly. Идеята е приложението да може да бъде инсталирано и да работи на всяка платформа, която използва съвместим със стандартите браузър, включително настолни и мобилни устройства. Прогресивните приложения не изискват пакетиране или публикуване в магазини за приложения като App Store на Apple или Google Play. Прогресивното приложение може да бъде свалено след първоначално адресиране на индексния файл. Въпреки, че PWA са Web приложения, те могат да предоставят функционалност, специфична за нативните мобилни приложения, като достъп до апаратните модули на устройството, получаване на push известия, работа в офлайн режим и др. Друго важно предимство на прогресивните Web приложения е краткото време за тяхното зареждане. Това е възможно благодарение на локалното кеширане на файловете, които изграждат приложението.

Прогресивните Web приложения работят под управлението на браузъри, подобно на уебсайтове. Следователно, за тях са в сила предимствата, характерни за Web приложенията, например:

- Могат да бъдат индексирани от търсачките на Web съдържание.
- Могат да работят на множество устройства, без да се налага пренаписване на програмния код.
- Разходите за разработката им са много по-ниски в сравнение с алтернативните нативни приложения.

Прогресивните Web приложения имат и функционалност, специфична за нативните приложения:

- Могат да работят и когато устройството е офлайн. Това означава, че PWA по подразбиране са и “Offline first” приложения.
- Могат да бъдат инсталирани и имат собствена икона чрез която бързо могат да бъдат стартирани.
- Поддържат push нотификации (известия) и актуализации.
- Имат достъп до по-голяма част от апаратните модули.

Основните характеристики на PWA са следните:

- **Откриваемост.** Приложението е откриваемо от резултатите от Web търсене и чрез магазините за приложения.
- **Инсталируемо.** Приложението може да бъде инсталирано както на мобилни платформи, така и на настолни компютри. То може да бъде стартирано чрез икона от началния екран, или да бъде прикачено към "Старт" менюто и лентата със задачи.
- **Прогресивно.** Потребителското изживяване на приложението се увеличава или намалява в зависимост от възможностите на устройството.
- **Безопасно.** Приложението използва сигурни HTTPS канали за комуникация.

- **Re-engageable.** Приложението може да получава push известия, дори когато не е активно и не е на фокус.
- **Независимо от наличието на мрежова свързаност**. Приложението работи офлайн и в условия на слаба мрежова свързаност. За да запази своята функционалност в офлайн режим, прогресивните Web приложения най-често използват собствена *локална* база данни. При възстановяване на мрежовата връзка, локалната база данни се синхронизира с отдалечената база данни (Web, облак), за да се получат най-новите данни. Програмният код на приложението не е необходимо да се сваля всеки път от сървър, тъй като може да се кешира локално.
- **Responsive.** Приложението се адаптира към размера или ориентацията на екрана на потребителя.
- **Linkable.** Приложението лесно се сваля и стартира чрез стандартна Web връзка. Единственото което трябва да знаем е URL адреса на индексния файл на приложението.

За значимостта на прогресивните Web приложения за бизнеса ще разгледаме част от фирмите, които предпочитат услугите им да са достъпни предимно чрез PWA, например:

- **Google**. Повече от половината потребители, които са избрали да инсталират нативно приложение чрез използване на банери, не успяват да завършат инсталирането на приложението, докато инсталирането на PWA е почти незабавно.
- **Starbucks**. Фирмата увеличава дневните активни потребители 2 пъти след предоставяне на достъп чрез PWA. Броят на поръчките, реализирани чрез PWA и тези чрез нативни приложения са почти равни.
- **Pinterest**. След като Pinterest преустрои мобилния си сайт като PWA достъпът до услугата се увеличават с 60%. Те също така отбелязват и 44% увеличение на приходите от реклами, генерирани от PWA потребителите, а времето, прекарано в сайта, се е увеличило с 40%.
- **Uber**. PWA на Uber е проектирано така, че да бъде бързо дори при 2G свързаност. Основното приложение се зарежда за по-малко от 3 секунди в 2G мрежи.
- **Tinder**. Tinder намали времето за зареждане от 12 секунди на 4.7 секунди с PWA. Прогресивното Web приложение е с 90% по-малък размер от оригиналното приложение за Android. Фирмата отчита, че ангажираността на потребителите, които използват PWA се е повишила.
- **OLX**. OLX декларират 250% повече повторно ангажиране на потребителите с помощта на push известия и 146% по-висока честота на кликване върху рекламите. По-добрата интерактивност на PWA приложението отнема 23% по-малко време, отколкото използването на нативното приложение.
- **Flipkart**. PWA на Flipkart е източник на 50% от придобиването на нови клиенти. Близо 60% от посетителите на PWA преди това са деинсталирали нативното приложение главно за да пестят място.
- **Selio**. Средните разходи на Selio за придобиване на клиенти чрез PWA са 10 пъти по-ниски, отколкото при използване на нативни приложения.

II. Манифест файл

Манифестът на едно прогресивно Web приложение е текстов файл, кото съсържа необходима информация в JSON формат за изтеглянето на приложението и

представянето му на потребителя като нативно приложение. PWA манифестите включват името на приложението, автор, икони, версия, описание, както и списък на всички необходими ресурси. Локацията на манифест файла в папка www, а името на файла - **manifest.json**. Развойната среда Monaca IDE създава автоматично съдържанието на манифест файла, в зависимост от напревените настройки за приложението. На Фиг. 11-1 е показано примерно съдържание на манифест файл на прогресивно приложение.

```
{
 "name": "Име на приложението",
 "short_name": "Кратко име на приложението",
 "description": "Текстово кратко описание на приложението",
 "icons": [
    {
     "src": "images/icons/icon-48x48.png",
     "sizes": "48x48",
     "type": "image/png"
    },
    // следва описание за останалите икони:
    // 72x72, 96x96, 128x128, 144x144, 152x152, 168x168, 192x192, 256x256
 ],
 "lang": "en",
 "dir": "ltr",
 "scope": "./",
 "start_url": "index.html",
 "display": "standalone",
 "orientation": "any",
 "background_color": "#00629d",
 "theme_color": "#00629d"
}
```

Фиг. 11-1 Съдържание на файл manifest.json

Предназначението на полетата от мнифест файла е следното:

- **name** - Заглавието на приложението. То се използва, когато потребителят е подканен да инсталира приложението. Това трябва да бъде пълното заглавие на приложението.
- **short_name** - Това е името на приложението, както ще се появи под иконата на приложението. То трябва да е кратко и съдържателно.
- **description** – Това поле е низ, чрез който разработчиците могат да обяснят какво прави приложението.
- **lang** – Полето задава основния език за стойностите на полетата от манифеста, визуализацията на които може да се насочва, в зависимост от стойността на поле dir. При конкретният пример избраният език е английски.
- **dir** – Задава посоката в която се изписва текста от полета като description. Визуализирането на текста може да бъде отляво-надясно (стойност ltr) и отдясно-наляво (стойност rtl). При стойност на полето “auto” посоката на текста се определя автоматично.
- **start_url** - Указва на браузъра коя страница да се зареди при стартиране на приложението. Обикновено това е index.html, но не е задължително.
- **icons** - Полето icons е масив от обекти, описващи URL на файлове с изображения, които могат да служат като икони на приложението.
- **orientation** – Полето определя ориентацията по подразбиране. Може да приеме една от следните стойности: any, natural, landscape, landscape-primary, landscape-secondary, portrait, portrait-primary и portrait-secondary.

- **scope** – Стойността на полето е низ, който определя навигационния обхват на контекста на приложението. Ако потребителят премине извън зададения обхват, той се връща към Web страница в раздела или прозореца на браузъра.
- **display** – Задава видът на изгледа, в който трябва да се появи приложението. Възможните стойности са следните:
  - *fullscreen* - Използва се цялата налична площ на дисплея.
  - *standalone* - Приложението ще изглежда като нативно приложение за съответната платформа.
  - *minimal-ui* - Приложението ще изглежда и ще се усеща като нативно приложение, но ще има минимален набор от елементи на потребителския интерфейс за управление на навигацията. Елементите ще се различават в зависимост от браузъра.
  - *browser* - Приложението се отваря в отделен раздел на браузъра или в нов прозорец, в зависимост от браузъра и платформата. Това е настройката по подразбиране.
- **background_color** – Дефинира заместващ цвят на фона, който страницата на приложението ще показва, преди да бъде зареден използвания набор от стилове.
- **theme_color** - Задава цвета на темата по подразбиране за приложението. Понякога това влияе на начина, по който операционната система показва приложението. Например, при Android цветът на темата определя цвета на рамката.

III. Библиотека PWACompact

Макар че много браузъри ще се съобразят със съдържанието на манифеста на приложението, не всеки браузър ще зареди или ще се съобрази с всяка стойност, която посочите. Боблиотеката PWACompat анализира манифест файла на приложението и автоматично добавя стандартни и нестандартни етикети (meta, link) за описание на връзките към иконите с различни размери, favicon, режима на стартиране, цветове и др. Това означава, че вече не е необходимо да добавяте редица стандартни и нестандартни етикети като

```
<link rel="icon" ... />
<meta name="orientation" ... />
```

към своите страници.

Библиотеката PWACompat ще зареди манифеста на вашето прогресивно приложение и ще реализира следната последователност от действия:

- Създава мета етикети за всяка икона, описана в манифеста, включително и за fav-icon.
- Създава специфични мета етикети за различните браузъри.
- Задава цвета на темата.
- За браузър Safari, PWACompat задава apple-mobile-web-app-capable (отваряне без браузър Chrome) за режими на визуализиране standalone, fullscreen или minimal-ui.
- Създава apple-touch-icon изображения, като добавя фона на манифеста към прозрачни икони - в противен случай iOS замества прозрачността с черно.
- Създава динамични заставки, които съответстват на заставките, генерирани за браузъри, базирани на Chromium.

IV. Service worker

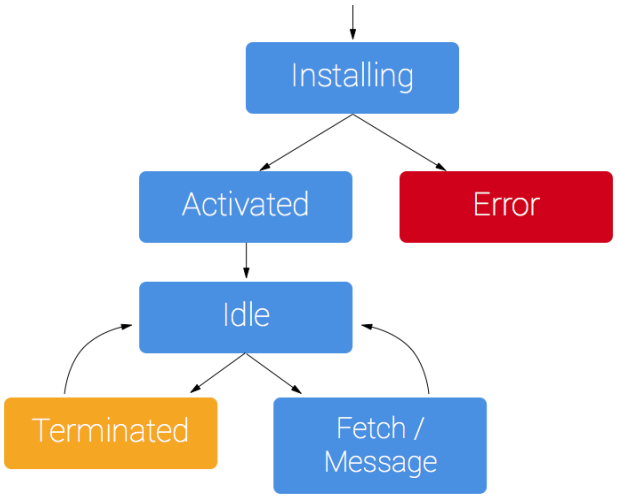
Функционалността на прогресивните Web приложения се базира на обекти от тип *service worker*. Те се базират на JavaScript Web worker и следователно се пишат на JavaScript. По същество те действат като *прокси сървъри*, които се намират между прогресивното приложение, браузъра и мрежата. Позволяват създаването на ефективни Offline-first приложения. Могат да прихващат мрежови заявки и да предприемат подходящи действия въз основа на това дали мрежата е налична или не е налична, както и да актуализират съдържание, хоствано на сървъра. Те позволяват и достъпа до API за работа с push известия. Това са обекти, които се управляват от събития.

Както Web worker, така и service worker се изпълнява в своя контекст. Следователно, те нямат достъп до DOM и се изпълняват чрез собствена нишка, различна от основната нишка за JavaScript кода. Всеки service worker работи във фонов режим и не блокира основния JavaScript код. Тези обекти са напълно асинхронни. В резултат на това, заявки към синхронни API не могат да се използват от service worker.

От съображения за сигурност service worker работят само с протокол HTTPS. Трябва да се има предвид, че при Firefox приложните програмни интерфейси на service worker не могат да се използват, когато потребителят е в режим на поверително сърфиране.

4.1 Жизнен цикъл на Service worker

За ефективното използване на service worker разбирането на жизнения цикъл на услугата е от съществено значение (виж Фиг. 11-2).



Фиг. 11-2 Жизнен цикъл на service worker

Жизненият цикъл на service worker се състои основно от три фази, които са:

- Регистрация.
- Инсталиране.
- Активиране.

4.1.1. Регистрация

Преди да се използва, service worker трябва да се регистрира като фонов процес. Това е първата фаза от неговия жизнения цикъл. *Тъй като service workers все още не се поддържат от всички браузъри, първо трябва да проверим дали браузърът поддържа*

service workers. По-долу е представен код, който можем да използваме, за да регистрираме service worker:

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/service-worker.js')
    .then(function (registration) {
       console.log('Service worker registered!');
    })
    .catch(function (error) {
       console.log('Регистрацията се провали!');
    })
}
```

Първо се проверява дали браузърът поддържа service workers, т.е. дали обектът navigator има свойство serviceWorker. Ако браузърът поддържа service worker следва регистриране чрез метод `register`. Този метод получава пътя до файла, съдържащ кода на service worker, в случая service-worker.js.

4.1.2. Инсталация

Фактът, че service worker е бил успешно регистриран, не означава, че той е инсталиран. Следващата фаза от жизнения цикъл е инсталиране на service worker. При успешна регистрация скриптът се изтегля и след това браузърът ще се опита да инсталира service worker. Това ще се случи само в един от следните случаи:

- Service worker не е бил регистриран преди това.
- Скриптът на service worker се е променил.

След като service-worker бъде инсталиран, се генерира събитие “install”. Можем да подслушваме това събитие и да изпълняваме някои специфични за приложението задачи. Например, в този момент можем да кешираме статичните ресурси на приложението:

```
var CACHE_NAME = 'my-cache-v1';
var urlsToCache = [
  '/',
  '/css/app.css',
  '/css/icons.css',
  '/js/app.js',
  '/js/routes.js',
  '/js/store.js'
];

self.addEventListener('install', (event) => {
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then( (cache) => {
        console.log('Opened cache');
        return cache.addAll(urlsToCache);
      })
    );
});
```

Чрез метод **open** (cache API), който получава като аргумент името на локалния кеш на service worker се реализира създаване на кеша, ако той не съществува. Ако съществува – само се отваря за използване. Чрез метод **addAll** (cache API) се кешира съдържанието на всички ресурси, URL адресите на които метод addAll получава като аргумент (масив). Тъй като методът open връща обещание се използва метод waitUntil, за да се забави инсталирането на service worker, докато обещанието не бъде обработено успешно.

При наличие на нова версия на service worker, новата версия се инсталира във фонов режим. Тя се активира едва когато вече няма заредени страници, които все още използват стария service worker. Веднага щом няма повече такива страници, които все още са заредени, новият service worker се активира. За целта ще трябва да актуализирате слушателя на събитието "install" в новия service worker, като зададете новата версия, например:

```
var CACHE_NAME = 'my-cache-v2';
```

Докато това се случва, предишната версия все още отговаря за извличанията.

4.1.3. Активиране

Ако инсталацията е била успешна, service worker вече е в инсталирано състояние. След това той преминава към следващата фаза от жизнения цикъл, която е фазата на активиране. Активирането на service worker не се реализира веднага след инсталирането. Активирането ще се реализира само в следните случаи:

- Ако в момента няма активен service worker.
- Ако е извикана функцията self.skipWaiting в обработчика на събитието “install” от скрипта на service worker.
- Ако потребителят опресни страницата.

Можем да подслушваме събитие “activate”, за да изпълняваме някои специфични за приложението задачи. Например, в този момент можем да отстраним от кеша всички ресурси, които не са част от кеш с указано име:

```
self.addEventListener('activate', (event) => {
  var cacheWhitelist = [CACHE_NAME];
  event.waitUntil(
    caches.keys().then((cacheNames) => {
      return Promise.all(
        cacheNames.map((cacheName) => {
          if (cacheWhitelist.indexOf(cacheName) === -1) {
            return caches.delete(cacheName);
          }
        })
      );
    })
  );
});
```

Това е полезно и за премахване на данни, които вече не са необходими, за да се избегне запълването на твърде много дисково пространство - всеки браузър има твърд лимит на количеството кеш памет, което даден service worker може да използва. Обещанията, предадени в waitUntil, ще блокират други събития до завършването им, така че можете да сте сигурни, че операцията по почистване на кеша ще завърши в момента, в който получите първото събитие за извличане на данни от новия service worker.

Service worker няма да получава събития като fetch и push, докато не завърши успешно инсталацията и не стане „активен”.

4.1.3. Събитие fetch

След като вече сте кеширали ресурсите на сайта, може да „кажете” на service worker да направи нещо с кешираното съдържание. Това се реализира лесно чрез събитието fetch. Събитието се задейства всеки път, когато се извлича ресурс, контролиран от service worker, което включва документите в посочения обхват и всички ресурси, към които има

препратки в тези документи. Например, ако index.html направи кръстосана заявка за вграждане на изображение, тя ще премине през service worker. Можете да прикачите слушател на събитието fetch, след което да извикате метода **respondWith**, за да прехванете HTTP отговорите и да ги актуализирате със собствено съдържание.

```
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request)
  );
});
```

В конкретният пример чрез метод **match** можем да съпоставяме всеки ресурс, заявен от мрежата, с еквивалентния ресурс, наличен в кеша (ако има наличен такъв). Съпоставянето се извършва чрез URL, както при нормалните HTTP заявки. Ако в кеша не е намерено съвпадение, можете да кажете на браузъра да направи fetch заявка по подразбиране за този ресурс, за да получи новия ресурс от мрежата, например:

```
fetch(event.request);
```

Ако не е намерено съвпадение в кеша и мрежата не е налична, можете просто да върнете някакъв вид резервна страница по подразбиране като отговор, използвайки метод match, например:

```
caches.match('./warning.html');
```

Можете да извлечете информация за всяка заявка, като извикате параметрите на обекта request, върнат от събитие fetch:

- event.request.url
- event.request.method
- event.request.headers
- event.request.body

Трябва да се предвиди обработка на всички неуспешни заявки. Можем да направим така, че ако ресурсите не се намират в кеша, те се заявяват от мрежата:

```
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request).then((response) => {
      return response || fetch(event.request);
    })
  );
});
```

При адресиране на липсващ в кеша ресурс трябва да го извлечем по мрежата и след това да го запишем в кеша:

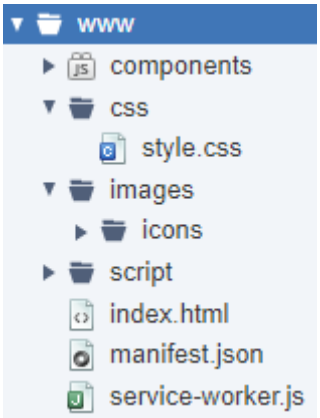
```
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request).then((resp) => {
      return resp || fetch(event.request).then((response) => {
        return caches.open(CACHE_NAME).then((cache) => {
          cache.put(event.request, response.clone());
          return response;
        });
      });
    })
  );
});
```

По този начин, следващите заявки към този ресурс ще могат да бъдат изтеглени от кеша при липса на мрежова свързаност. Тук връщаме мрежовата заявка по подразбиране с return fetch(event.request), която връща обещание. Когато това обещание бъде разрешено, ние отговаряме, като изпълняваме функция, която получава достъп до кеша с помощта на caches.open(`CACHE_NAME`); това също връща обещание. Когато това обещание се разреши, се използва cache.put(), за да се добави ресурсът в кеша. Ресурсът се взема от event.request, а след това отговорът се клонира с response.clone() и се добавя в кеша. Клонингът се поставя в кеша, а оригиналният отговор се връща на браузъра, за да бъде предоставен на страницата, която го е извикала. Клонирането на отговора е необходимо, тъй като потоците от заявки и отговори могат да се четат само веднъж. За да върнем отговора на браузъра и да го поставим в кеша, трябва да го клонираме. Така оригиналът се връща на браузъра, а клонингът се изпраща в кеша. Единственият проблем, който имаме сега е, че ако заявката не съвпада с нищо в кеша и мрежата не е достъпна, заявката ще се провали. За да не стане това, трябва да осигурим резервен вариант по подразбиране, така че каквото и да се случи, потребителят поне ще получи резултат:

```
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request).then((resp) => {
      return resp || fetch(event.request).then((response) => {
        let responseClone = response.clone();
        caches.open(CACHE_NAME).then((cache) => {
          cache.put(event.request, responseClone);
        });
        return response;
      }).catch(() => {
        return caches.match('./ресурс който се връща при изключение');
      })
    })
  );
});
```

V. Разработване на прогресивно Web приложение

Ще реализираме приложението “Weather Forecast”, описано в Глава 10 „Работа с метеорологични данни”, като прогресивно Web приложение. Нека името на новото приложение е “Weather Forecast PWA”. За да ускорим разработката на приложението ще използваме шаблон за създаване на PWA, който Monaca IDE предоставя. От Monaca Dashboard изберете “Create New Project”. За Project Type изберете “Simple Applications”, а след това “Blank PWA”. Структурата на проекта е следната:



Виждаме, че по структура това е Web приложение, като основната разлика е наличието на файл service-worker.js. Съдържанието на индексния файл е показани на Фиг. 11-3.

```
<!DOCTYPE HTML>
<html>
<head>
 <meta charset="utf-8">
 <meta name="viewport" content="width=device-width, initial-scale=1,
     maximum-scale=1, user-scalable=no">
 <meta http-equiv="Content-Security-Policy" content="default-src * data: gap:
  content: https://ssl.gstatic.com; style-src * 'unsafe-inline';
  script-src * 'unsafe-inline' 'unsafe-eval'">
 <script src="components/loader.js"></script>
 <link rel="stylesheet" href="components/loader.css">
 <link rel="stylesheet" href="css/style.css">
 <link rel="manifest" href="/manifest.json">
 <script async src="https://cdn.jsdelivr.net/npm/pwacompat@2.0.7/pwacompat.min.js">
 </script>
 <title>PWA Minimum</title>
```

1

```
<script>
 if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/service-worker.js')
   .then(function(registration) {
    console.log('ServiceWorker registration successful with scope: ',
     registration.scope);
   }).catch(function(error) {
    console.log('ServiceWorker registration failed: ', error);
   });
  }
</script>
```

```
</head>

<body>
    This is a PWA template for Monaca app.
</body>

</html>
```

Фиг. 11-3 Съдържание на файл index.html

Основните разлики с досега разработваните проекти са две:

- Асинхронно сваляне на библиотека PWACompact чрез етикет script.
- Регистрация на service worker.

За да регистрираме service-worker използваме метод **register** (виж Секция 1). На метод register трябва да се подаде като аргумент пътят до JavaScript файла, съдържащ кода на service worker.

Програмният код от файл service-worker.js съдържа обработка на събития “Install”, “activate” и “fetch”. Този програмен код е показан на Фиг. 11-4. Няма съществена разлика в кода за обработка на тези събития във файл service-worker.js и разгледания в Секция 4.1 код. Когато създаваме новия проект, програмният код на service worker няма да се променя.

Манифест файлът на прогресивното Web приложение съдържа информация за използваните икони, цветова схема, тип и ориентация на дисплея и др. За повече информация виж Фиг. 11-1.

```
var CACHE_NAME = 'my-cache-v1';
var urlsToCache = [
  '/',
  '/css/style.css',
  '/script/main.js'
];

self.addEventListener('install', function (event) {
 event.waitUntil(
   caches.open(CACHE_NAME)
    .then(function (cache) {
     console.log('Opened cache');
     return cache.addAll(urlsToCache);
    })
  );
});

self.addEventListener('activate', function (event) {
 var cacheWhitelist = [CACHE_NAME];
 event.waitUntil(
   caches.keys().then(function (cacheNames) {
    return Promise.all(
     cacheNames.map(function (cacheName) {
      if (cacheWhitelist.indexOf(cacheName) === -1) {
       return caches.delete(cacheName);
      }
     })
    );
   })
 );
});

self.addEventListener('fetch', function (event) {
 event.respondWith(
   caches.match(event.request)
    .then(function (response) {
     if (response) {
      return response;
     }
     var fetchRequest = event.request.clone();
     return fetch(fetchRequest).then(
      function (response) {
       if (!response || response.status !== 200 || response.type !== 'basic') {
        return response;
       }
       var responseToCache = response.clone();
       caches.open(CACHE_NAME)
        .then(function (cache) {
         cache.put(event.request, responseToCache);
        });
       return response;
      }
     );
    })
  );
});
```

Фиг. 11-4 Съдържание на файл service-worker.js

5.1 Конфигуриране на проекта

Можете да конфигурирате прогресивното Web приложение чрез Monaca IDE. За целта изберете “Configure” -> “App Settings for PWA…”. Трябва да въведете следната информация за вашето приложение:

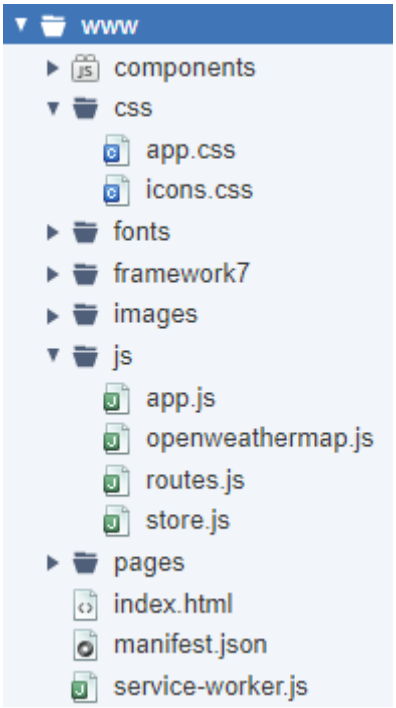
- Пълно и кратко име на приложението.
- Описание на приложението.

- Използван език.
- Ориентация на текста.
- Навигационният обхват на контекста на приложението (scope).
- Начален URL адрес.
- Тип и ориентация на дисплея.
- Цветови теми.
- Икони на приложението.

Въз основа на тази информация ще се актуализира съдържанието на файл manifest.json.

5.2 Програмен код

Единственото, което трябва да направите е да копирате програмния код на приложение “Weather Forecast” в папка www на проекта “Weather Forecast PWA” (виж Фиг. 11-5).



Фиг. 11-5 Структура на проект “Weather Forecast PWA”

В папка images трябва да остане папка icons от базовия PWA проект. Съдържанието на файлове service-worker.js и manifest.json не се променя. Промяна трябва да направите само в индексния файл – зареждане на библиотека PWACompact и регистрация на service worker:

```
<link rel="manifest" href="/manifest.json">
<script async src="https://cdn.jsdelivr.net/npm/pwacompat@2.0.7/pwacompat.min.js">
</script>
<script>
  if ('serviceWorker' in navigator) {
    navigator.serviceWorker.register('/service-worker.js')
      .then(function(registration) {
        console.log('ServiceWorker registration, scope: ', registration.scope);
      }).catch(function(err) {
        console.log('ServiceWorker registration failed: ', err);
      });
    }
</script>
```

5.3 Изграждане на проекта

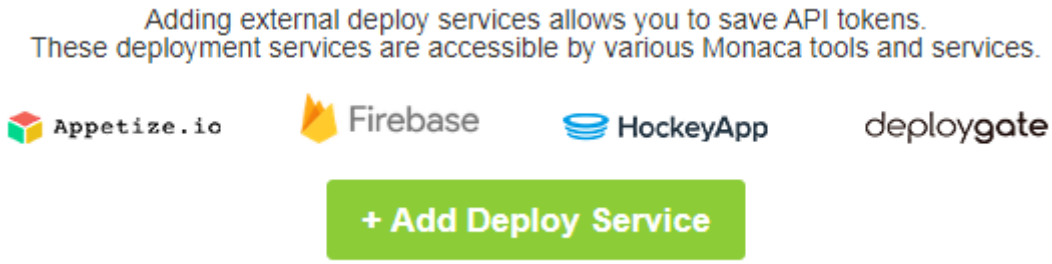
За да изградите проекта изберете “Build” -> “Build App for PWA…” и натиснете бутон “Start Build”. В резултат на изграждането ще получите ZIP архив, който съдържа всички необходими конфигурационни файлове, програмния код от папка www, както и папка res с ресурсите на приложението за 4 платформи в 4 папки:

- android – ресурси за ОС Android.
- ios – ресурси за ОС iOS.
- winrt – ресурси за ОС Windows.
- electron – ресурси за създаване на десктоп приложение.

По подразбиране, ресурсите на приложението са иконите (папка icon) и началните екрани (папка screen) в PNG формат.

5.4 Разгръщане на проекта

Така полученият програмен код трябва да се хоства на сървър, за да може прогресивното приложение да бъде инсталирано чрез зададен URL адрес. За целта може да използвате коя да е услуга за хостване на Web проекти. Развойната среда Monaca IDE позволява да хоствате своите прогресивни проекти автоматично като използвате няколко услуги, специализирани за тази цел. От менюто изберете “Configure” -> “Deploy Service…”. На този етап можете да избирате едно от 4 възможни хранилища на проекти (виж Фиг. 11-6).



Фиг. 11-6 Избор на хранилище за прогресивното Web приложение

Ще използване Google Firebase като хранилище на проекта. За целта натиснете бутон “+Add Deploy Service” и от падащото меню изберете Firebase. Трябва да въведете следната информация, за да може Monaca IDE да комуникира с Firebase:

- Config Alias - Псевдоним на текущата конфигурация. Тъй като може да използвате множество конфигурации за Deploy Service в това поле трябва да въведете свое уникално име за тази конфигурация.
- API Token – Уникален низ (токен), който се използва за идентификатор на даден клиент на услугата Firebase. Този низ ще получите на вашата електронна поща при първоначална регистрация за използване на Firebase. От конзолата на компютъра трябва да изпълните следната команда:

```
firebase deploy --token "FIREBASE_TOKEN"
```

  където FIREBASE_TOKEN трябва да заместите с получения токен. За да може да изпълните тази конзолна команда трябва да инсталирате Firebase CLI – интерфейсът за достъп до услуга Firebase от ниво конзола. Свалете инсталационния файл от следния адрес:

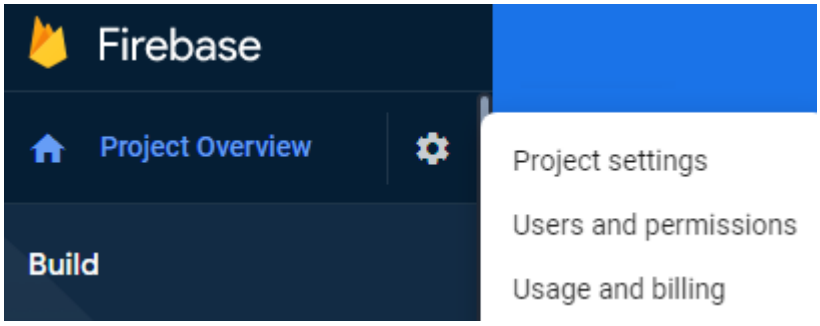
https://firebase.google.com/docs/cli#windows-standalone-binary

След като регистрирате своя токен, Monaca IDE ще може да се свързва с услуга Firebase.

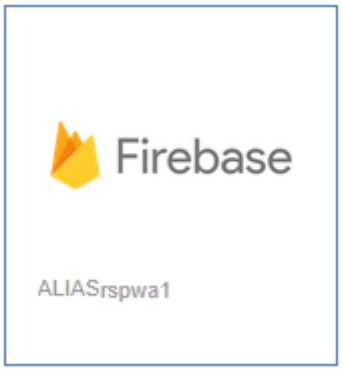
- Project ID – Идентификатор на Firebase проекта с който се асоциира вашето приложение. За да получите този идентификатор е необходимо да създадете проект. Ще реализирате това през конзолата на Firebase:

  https://console.firebase.google.com/

  Изберете “Add Project” и задайте име на проекта. При създаване на проекта ще получите и необходимата стойност за Project ID. Може да видите тази стойност като изберете “Project Settings” (използвайте иконата зъбно колело):



След успешно описание на хранилището за вашия проект ще може да го използвате за хостване на приложението след успешно изграждане. За целта трябва да изберете следната икона:



След като приложението вече е разгърнато успешно може да го адресирате с цел сваляне и инсталиране на коя да е платформа. Адресът, от който може да реализирате достъп до проекта си се получава по следния начин:

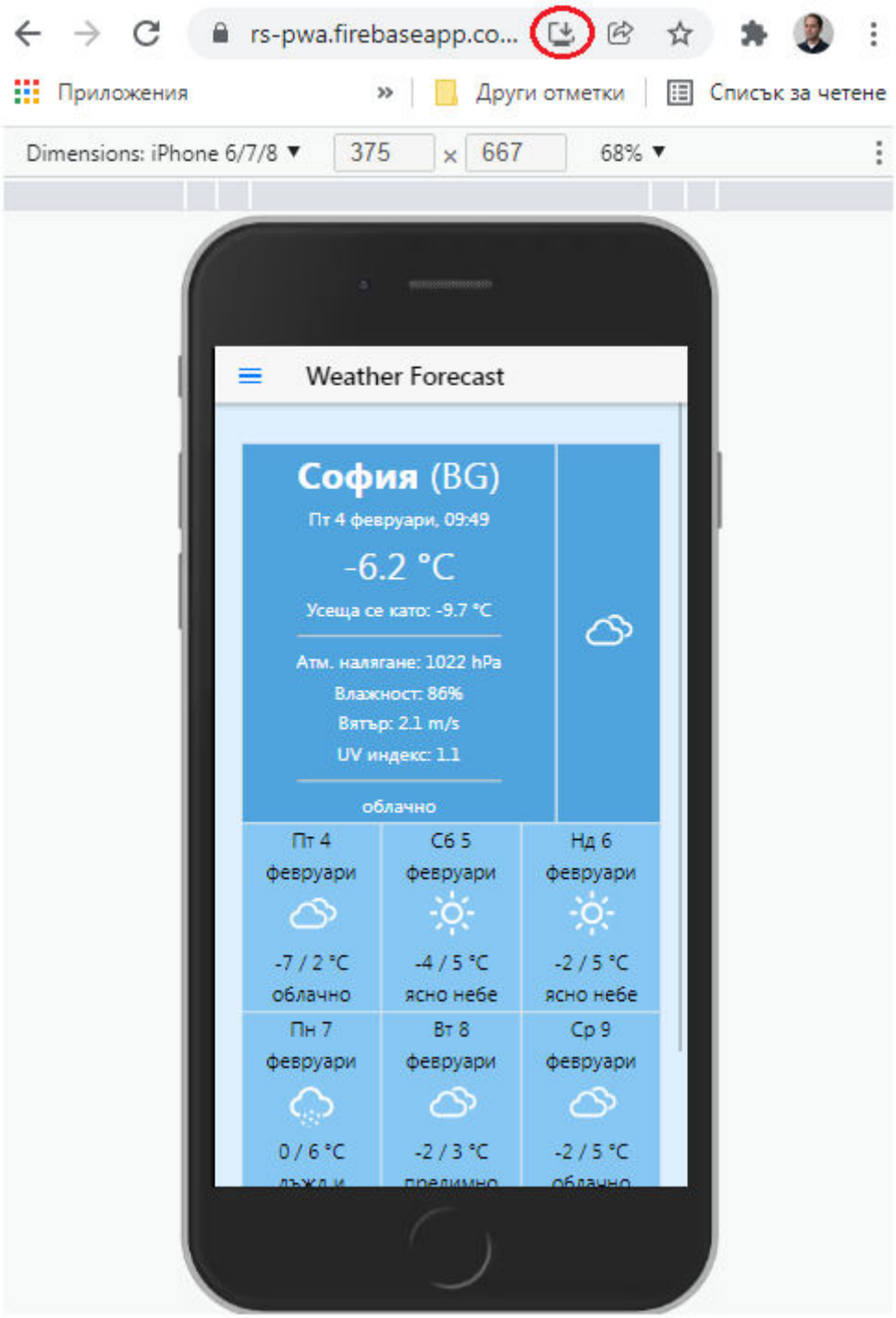
https://project-name.firebaseapp.com/index.html

Сменете “project-name” с името на вашия Firebase проект.

5.5 Тестване на приложението

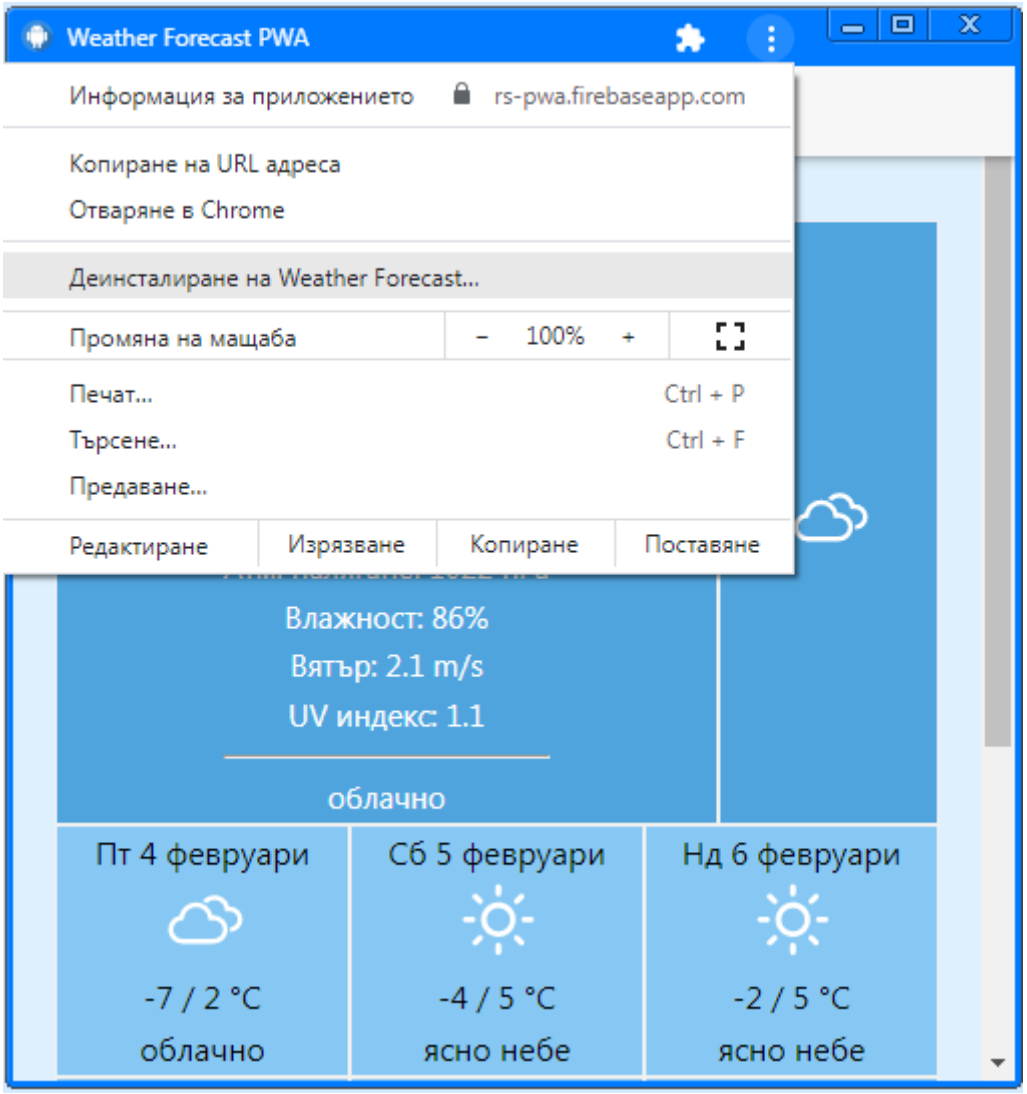
След като се уверите чрез Monaca Debugger, че приложението функционира нормално може да преминете към неговата инсталация. Нека първо да инсталираме приложението на персонален компютър. Отворете браузъра (Chrome, Opera, MS Edge, Vivaldi или Safari за iOS и macOS) с който работите и въведете адреса на вашия Firebase проект. На Фиг. 11-7 е показан получения резултат при използване на браузър Chrome. Чрез иконата,

заградена в червено, може да инсталирате приложението при което ще получите икона за стартирането му от десктопа.



Фиг. 11-7 Инсталиране на прогресивното Web приложение

По аналогичен начин може да инсталирате приложението при мобилни устройства с ОС Android, iOS и Windows. Както всяко приложение, така и прогресивните приложения могат да бъдат деинсталирани. За целта след като стартирате приложението от менюто изберете „Деинсталиране на Weather Forecast…”, както е показано на Фиг. 11-8.



Фиг. 11-8 Деинсталиране на прогресивното приложение

Заключение

В тази глава се запознахте с предназначението и начина на функциониране на прогресивните Web приложения. Научихте каква е ролята на обектите от тип service worker и какъв е техния жизнен цикъл. Приложението от тази глава е пример за PWA, което може да бъде инсталирано на мобилни устройства с ОС Android, iOS, както и на персонални компютри. В тази глава научихте и как може да хоствате вашето прогресивно приложения в облака на Firebase.

Глава 12

Push известяване

I. Въведение

Комуникациите в Интернет среда, в зависимост от инициатора на заявките, се делят на *pull* (get) и *push* комуникации. При pull комуникациите инициаторът е програмното осигуряване от страна на клиента. На този етап, това е преобладаващия тип комуникации. Все по-голям става дялат на push комуникациите. При push комуникациите инициаторът на обмена е програмното осигуряване от страна на сървър. Услугите, базирани на push комуникации се основават на предварително дефинирани предпочитания за получаваната информация. Това се нарича модел “publish-subscribe”. Клиентът се "абонира" за различни информационни канали, предоставени от сървъра. Когато в някой от тези канали е налично ново съдържание, сървърът изпраща тази информация на клиента. Информационните канали могат да се базират на използването на различни технологии, например:

- Кратки съобщения – push SMS. Типичен пример за този информационен канал е изпращане на push кратко съобщение на всеки нов потребител на даден мобилен оператор. Този SMS е бинарен и съдържа информация за конфигуриране на достъпа до различни услуги на оператора.
- Електронна поща – push email. Push email е система за електронна поща, която осигурява възможност новите писма да се предават веднага на клиента, без те да са поискани. Клиентите за електронна поща включват смартфони и IMAP4 приложения за електронна поща, инсталирани на персонални компютри. Фирмата Apple поддържа за iPhone и iPad технология push имейл чрез Google Sync и платформата Exchange ActiveSync на Microsoft. При ОС Android има вграден клиент на Gmail в Android, който използва Google Cloud Messaging за изпращане на писма, настроени да се синхронизират с телефона.
- Известия – push notifications. Този тип съобщения се изпращат до група получатели при възникване на определено събитие. Получателите могат да бъдат мобилни клиенти или Web клиенти.

Най-голяма печалба може да се получи при използване на push известяване, тъй като потребителите на тази услуга са най-многобройни. Това са клиенти с мобилни устройства (in-app push notification) и клиенти с персонални компютри (Web push notification). За изпращане на Web известия е необходим Web сайт, на който е инсталиран код за Web известяване. Това означава, че фирмите, които нямат мобилни приложения, могат да се възползват от предимствата на push известията, например персонализирани комуникации в реално време със своите клиенти. За да получавате push известия на своя мобилен телефон трябва да инсталирате приложение, което дава тази функционалност. Може да използвате и прогресивно Web приложение, което да предоставя push известяване както за мобилните потребители, така и за потребителите с персонални компютри.

Чрез push известяване може да доставите разнообразна информация за своите потребители, например:

- Заглавие на съобщението. Заглавието трябва да е кратко, информативно и запомнящо се.
- Описание - текст, който трябва да информира клиента за конкретно събитие.
- Икона – най-често това е логото на компанията, изпратила известяването.
- Банер – изображение с размер на банер. Банерите служат за привличане на потенциални клиенти и служат най-често за хипервръзка към рекламна информация.
- Допълнителна информация във формат „име на поле: стойност”.
- Адрес на сайт, който трябва да се отвори при получаване на известието.
- Бутони, които реализират желани препратки.
- Изображение, което да бъде визуализирано при получаване на известяване.
- Звуков файл, който да замести звуковия сигнал, който е активен по подразбиране за съответната мобилна ОС.

Тази информация прави push известяването мощно средство за доставяне на съдържание на потенциалните клиенти на мобилни и Web услуги. Известията могат да бъдат доставени до всички потребители, инсталирали приложението, или персонализирано - до група от потребители или хипер-персонализирано – до определен потребител на услугата. Това прави технологията push известяване идеална за разработване на услуги, базирани на местоположението на клиентите – Location Based Services (LBS). При този тип услуги клиентите получават специфична информация, която зависи от тяхната локация. Например, когато потребителят на услугата преминава в близост до определен магазин, той получава информация за новите стоки и отстъпките за деня. Услугите, базирани на локацията на потребителите могат да се използват както на открито, така и на закрито. Местоположението на потребителите на открито се получава чрез GPS приемника, интегриран в мобилните устройства, а местоположението в сгради - чрез различни технологии като Near Field Communications (NFC) и iBeacons.

Един пример за услуги, основани на местоположението на клиентите са тези, които използват *геофенсинг*. Терминът geofence (geo defence) дефинира виртуален периметър за реална географска област. Това може да бъде окръжност с определен радиус или да съответства на предварително определен граници, най-често полигон, който описва контурите на квартал, сграда или специфично пространство между сгради. Използването на geofence за създаване на LBS услуги се нарича *geofencing* (геофенсинг). По този начин услугите могат да определят кога клиентите им *влизат* или *излизат* от така дефинираната защитена зона. В зависимост от това, те могат да получават специфични push съобщения.

Основните предимства на геофенсинга са следните:

- Възможност за много добро таргетиране на клиентите на услугата. Можете да ангажирате своите клиенти с персонализирани съобщения, които са подходящи точно за тях.
- Филтриране на потребителите на дадена услуга в зависимост от географския район в който те се намират.
- Създаване на хипер-персонализирана реклама чрез която броят на ангажираните с вашата услуга потребители лесно се увеличава.
- Възможност за персонализирано доставяне на съдържание. Например, на посетителите на мол може да се предложи да посетят специфичен магазин, в

зависимост от това къде посетителят си е задържал вниманието си за определен минал интервал от време.

- Чрез комбиниране на информация за предвижването на посетителите в дадена сграда, онлайн активността им, информацията за покупки и поведение при сърфиране в Интернет, бизнесът може да подобри потребителското изживяване, да увеличи ангажираността на клиентите и да разбере по-добре тяхното поведение. Същата тази информация може да се използва и за насочване на реклама и услуги към хора, които преди това са посещавали определени места.

По-долу са представени статистически данни, свързани с геофенсинг маркетинга за последната година:

- Геофенсингът е съвместим с 92% от смартфоните.
- Мобилните реклами с геофенсинг са с двойно по-висока честота на кликванията.
- 53% от купувачите са посетили търговец на дребно, след като са получили съобщение, базирано на местоположението им.
- 90% от push кратките съобщения се прочитат в рамките на 3 минути.

Очаква се глобалният пазарен дял на услугите, използващи геофенсинг да се увеличи с 23,96% (2.21 милиарда. долара) за периода 2022-2025 г.

II. Услуги за push известяване

Съществуват множество услуги за предоставяне на възможност за получаване и изпращане на push известия. Ще разгледаме две от тях, които предлагат безплатен план на използване – Google Firebase и OneSignal.

Firebase е облачна платформа, собственост на Google, за хостване на мобилни и Web приложения. Първият продукт на Firebase беше приложения програмен интерфейс за базата данни Firebase Realtime Database, който синхронизира данните на приложенията в iOS, Android и Web устройства и ги съхранява в облака на Firebase. На този етап услугата за изпращане на push известия Google Cloud Messaging за устройства с ОС Android, е заменена от Firebase Cloud Messaging чрез която могат да се изпращат push известия и до устройства с iOS и Web.

Услугата **OneSignal** е водеща на пазара, свързан с изпращане на push известия (in-app и Web), както и push SMS и push email. OneSignal е мултиплатформена услуга за известяване и е безплатна за използване. Може да използвате плъгините на OneSignal чрез които се намалява времето за създаване на мобилни и Web приложения, които използват push технологиите на компанията. Основните предимства на тази услуга са:

- Поддържа неограничен брой устройства и известия без такси.
- Настройката за използване на услугата и създаване на push известие в OneSignal е проста и лесна за изпълнение.
- Мултиплатформена услуга - предоставя единен потребителски интерфейс и API за доставка на съобщения за различни платформи, като iOS, Android, Amazon Fire, Windows Phone, Chrome Apps и др.
- Предоставя SDK за почти всяка хибридна среда за мобилна разработка, включително Cordova, PhoneGap, React Native, Intel XDK, Unity и др.

2.1 Създаване на Firebase проект

Услуга OneSignal използва Firebase платформата за изпращане и получаване на push известия. Това означава, че първо трябва да направите регистрация за Firebase. След като се регистрирате от конзолата създайте нов проект:

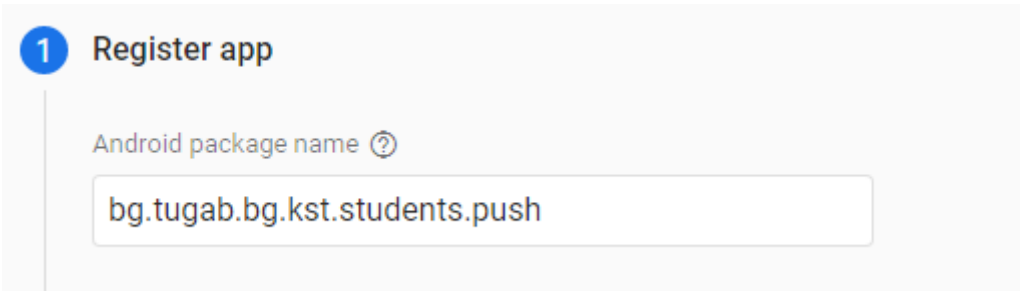
https://console.firebase.google.com/

Изберете “Add project” и задайте име на вашия проект. Разрешете Geegle Analytics за вашия проект. След създаване на проекта ще бъдете препратени до страница чрез която ще можете да конфигурирате проекта (виж Фиг. 12-1).

Фиг. 12-1 Firebase проект

От менюто изберете “Project Settings”. От тази страница ще получите информация за “Project ID” и “Project number”. От секция “Your apps” изберете иконата за Android:

От формата “Add Firebase to your Android app” трябва да опишете за кое Android приложение се отнася този проект. В поле “Android package name” въведете името на пакета за вашето мобилно приложение, което ще използва push известяване (виж Фиг. 12-2).



Фиг. 12-2 Асоциирене на Firebase проекта с конкретно Android приложение

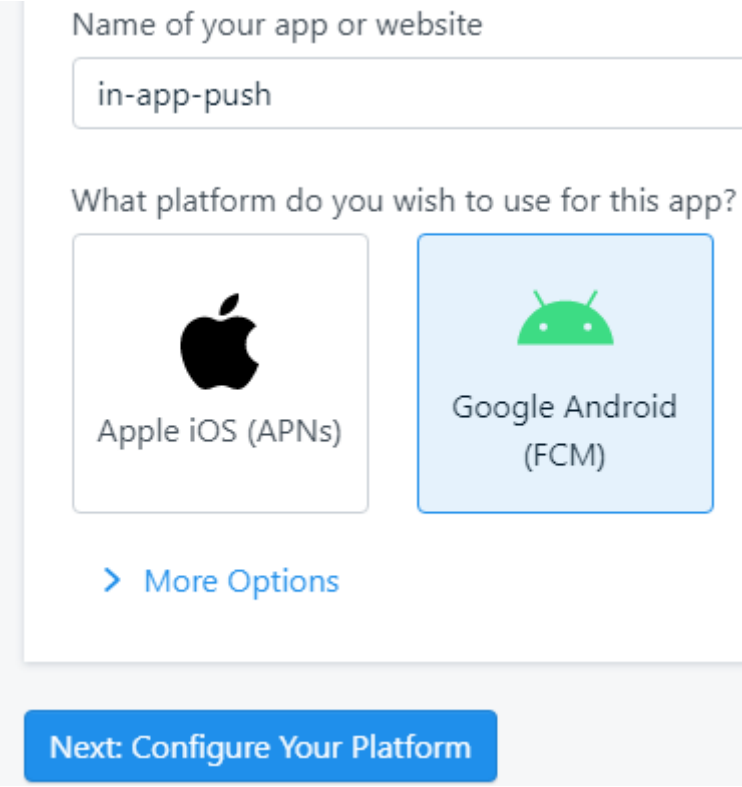
Друга информация не е необходимо да въвеждате. Чрез бутон “Next” се върнете на ниво конзола на Firebase. От “Project Settings” изберете “Cloud Messaging”. В секция “Project credentials” ще получите информация за “Server key” и “Sender ID”. Копирайте тази информация, тъй като тя ще е необходима на услуга OneSignal.

2.2 Създаване на OneSignal проект

Направете регистрация за услуга OneSignal:

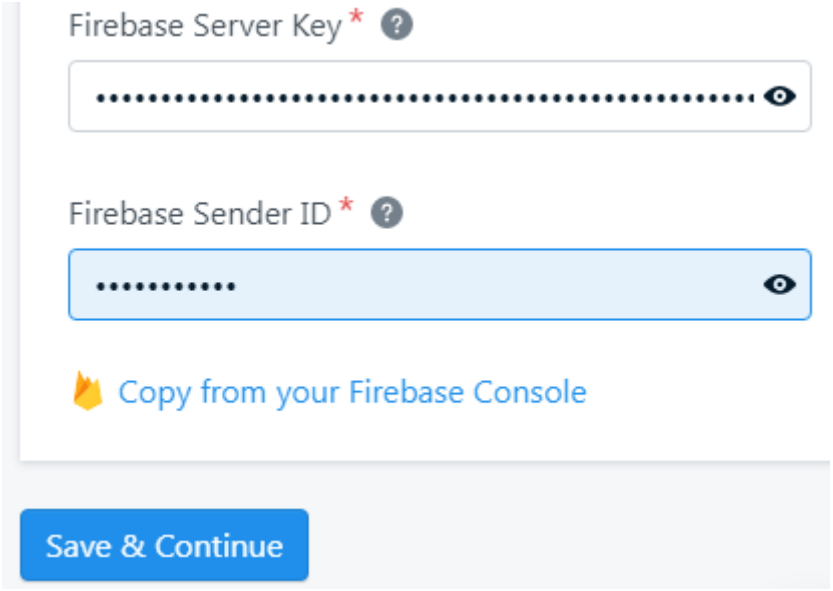
https://app.onesignal.com/

След успешна регистрация ще бъдете пренасочени към Dashboard на OneSignal. Изберете създаване на нов проект. От “New App / Website” задайте име на приложението и изберете платформа Android (виж Фиг. 12-3).



Фиг. 12-3 Създаване на OneSignal проект

Натиснете бутон “Next: Configure Your Platform”. От секция “Google Android (FCM) Configuration” задайте стойност за “Firebase Server Key” и “Firebase Sender ID”, които получихте при създаване на Firebase проекта. След като копирате стойностите на тези полета и ги прехвърлите в OneSignal натиснете бутон “Save & Continue” (виж Фиг. 12-4).



Фиг. 12-4 Задаване на стойности за Firebase “Server key” и “Sender ID”

След това трябва да изберете SDK за платформата с която ще работите. Тъй като ще създадем Cordova мобилно хибридно приложение трябва да изберете Cordova. Потвърдете своя избор чрез бутон “Save & Continue”. От следващата страница ще получите стойността на “App ID”, която трябва да запомните. Тъй като на този етап няма как да имате абонирани потребители, то няма смисъл да тествате това чрез бутон “Check Subscribed Users”. Натиснете бутон “Done”.

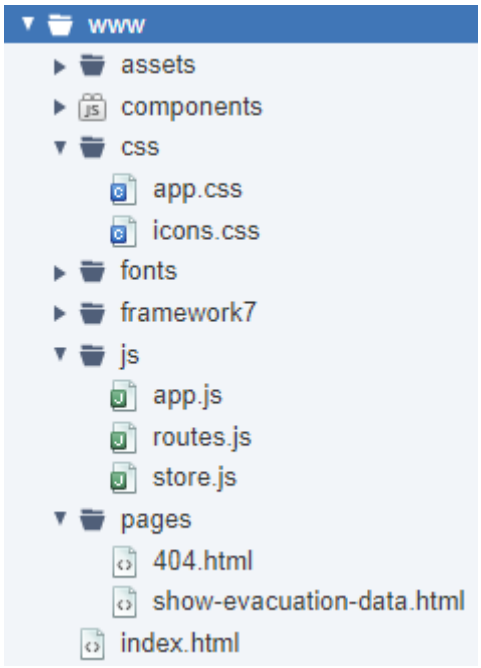
III. Мобилно приложение

Ще създадем мобилно хибридно приложение, което ще използва услуга OneSignal, за да получава push известия. Приложението ще уведомява своите клиенти чрез push известие при активиране на план за евакуация от сградата в която те се намират. Ще използваме конзолата на OneSignal, за да изпращаме известия на клиентите.

На практика, реализацията на така описаната функционалност изисква създаването както на мобилно приложение, така и на сървърен код, който да изпраща push известията. Услугата OneSignal предлага SDK за множество програмни езици чрез които можете да реализирате сървърния код. Трябва да се реализира и филтриране на клиентите на услугата в зависимост от тяхното местоположение. Трябва да се изпрати известие само на клиентите, които са в сградата, за която е активиран евакуационен план. За целта сървърният код трябва да следи местоположението на всеки клиент. Това се реализира чрез GPS и технологии като NFC и iBeacon. Услугата трябва да има и достъп до всички сензори в сградата (температура, дим, изтичане на газ и др.). Въз основа на данните от сензорите тя трябва да се реши дали да се активира евакуационен план и какъв точно да е той (пожар, обгазяване, заметресение). Следва уведомяване на клиентите на услугата, че е активиран евакуационен план и какво трябва да предприемат те като действие. Би трябвало услугата да доставя персонализирано съдържание - всеки посетител трябва да получи евакуационен план за етажа на който той се намира, както и маршрута, който трябва да следва, за да напусне безопасно сградата.

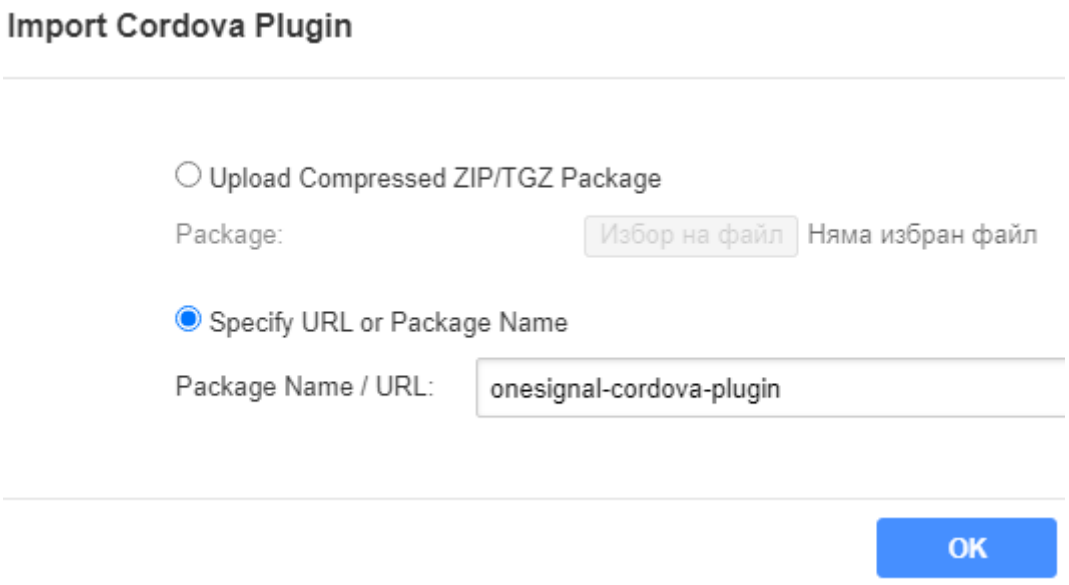
3.1 Структура на приложението

Ще създадем хибридно мобилно приложение чрез програмна рамка Framework7 v6.x. Структурата на приложението е показана на Фиг. 12-5.



Фиг. 12-5 Структура на приложението

За да получим достъп до услуга OneSignal ще използваме плъгина “onesignal-cordova-plugin”. Този плъгин трябва да се импортира към проекта, тъй като не е част от интегрираните в Monaca IDE плъгини. Това означава, че трябва да имате *платен план* за използване на Monaca IDE. Импортирайте плъгина от Settings -> “Cordova Plugin Settings…”. Натиснете бутон <Import Cordova Plugin>, изберете “Specify URL or Package Name” и в поле “Package Name / URL” въведете onesignal-cordova-plugin (виж Фиг. 12-6).



Фиг. 12-6 Включване на плъгина “onesignal-cordova-plugin” към проекта

3.2 Потребителски интерфейс

Приложението трябва да има изчистен потребителски интерфейс. Това произтича от неговото предназначение. То трябва да се стартира еднократно, за да може да се реализира абонирането за услуга OneSignal. След стартиране, потребителят трябва да получи информация какво представлява и как се използва приложението. При успешно абониране, услугата OneSignal ще може да изпраща push известия до приложението, независимо дали то е стартирано или не. Ако потребителят докосне прозореца на известието, приложението ще визуализира документ-шаблон, който ще покаже в компонент `card` информацията, която сървърът е изпратил. При конкретното приложение с всяко push известие се изпраща следната информация:

- Заглавие (title).
- Съдържание (body).
- Потребителско поле type – причина за евакуация, например пожар или изтичане на газ.
- Потребителски тип поле floor – на кой етаж е проблема.
- Изображение с евакуационния план на етажа на който е потребителя (bicon).
- Кога е изпратено известието в Unix timestamp формат (google.sent_time).

3.3 Програмен код

Потребителският програмен код е съсредоточен в следните файлове:

- Индексен файл **index.html** – реализира зареждане на необходимите библиотеки с CSS и JavaScript код, както и показване на текст, който описва предназначението на приложението.
- Файл **app.js** – инициализация на приложението, включително и абонамент за услугата OneSignal.
- Файл **routes.js** – описва всички необходими маршрути.
- Файл **store.js** – локално хранилище на приложението. Използва се с цел запомняне на мета данните на последното push известие.
- Файл **app.css** – стилово форматиране на приложението.
- Файл **show-evacuation-data.html** – документ-шаблон, който визуализира мета данните от push известието.

3.3.1 Инициализация на приложението

Инициализацията на приложението се реализира чрез програмния код от файл app.js (виж Фиг. 12-7).

```
// Dom7
var $$ = Dom7;

// Framework7 App main instance
var app = new Framework7({
  el: '#app',                       // App root element
  id: 'bg.tugab.kst.push',          // App bundle ID
  name: 'Evacuation',               // App name
  theme: 'auto',                    // Automatic theme detection
  store: store,
  routes: routes,
});

// Init/Create main view
var mainView = app.views.create('.view-main', {
    url: '/'
});
// ------------------------------------------------------------------------
$$(document).on('deviceready', function () {
  window.plugins.OneSignal.setAppId("a609e9b9-ea12-4e52-bf71-2a99ef127cbf");
  window.plugins.OneSignal.setNotificationOpenedHandler(function (data) {
    var title = data.notification.title;
    var body = data.notification.body;
    var bicon = data.notification.rawPayload.bicon;
    var floor = '';
    var type = '';
    var additionalData = data.notification.additionalData;
    if (additionalData != undefined) {
      floor = additionalData.floor;
      type = additionalData.type;
    }
    var timestamp = data.notification.rawPayload['google.sent_time'];
    var now = getDateFromTimestamp(timestamp);
    var sentDate =
       `${now.day} ${now.month} ${now.year}, ${now.hours}:${now.minutes}`;
    var evacuation = {
      title: title,
      body: body,
      type: type,
      location: floor,
      bicon: bicon,
      timestamp: sentDate,
    };
    store.state.evacuation = evacuation;
    app.views.main.router.navigate('/show/evacuation/');
  });
});
```

Фиг. 12-7 Инициализация на приложението

След създаване на обектите за достъп до Framework7 и DOM7 се изгражда обект за главния изглед на приложението. Когато библиотеката Cordova се зареди в паметта (събитие “deviceready”), приложението се абонира да използва услуга OneSignal. Това се реализира чрез метод `setAppId` чрез който се задава идентификатора на вашия OneSignal проект. Ако сте забравили стойността на “App ID” може да го получите от OneSignal dashboard. От Settings изберете “Keys & IDs” и копирайте стойността на полето “OneSignal App ID”. Чрез метод `setNotificationOpenedHandler` се стартира наблюдател за следене на получаването на push известия. Анонимната callback функция получава обекта `data`, който съдържа данните, изпратени от OneSignal. Това е JSON обект със съдържание, показано на Фиг. 12-8.

Фиг. 12-8 Съдържание на push известие, изпратено от OneSignal

Данните, които се извличат от обект `data` са следните:

- Заглавие на известието (title).
- Основен текст (body).
- URL към изображение с плана за евакуация (bicon).
- На кой етаж е проблемът (floor).
- Какъв тип евакуационен план е задействан (type).
- Дата и час на генериране на push известието (goole.sent_time).

Всички тези данни се обединяват до един JSON обект с име `evacuation`. Стойността му се запомня в локалното хранилище. Веднага след това, чрез метод `navigate` се предизвиква програмно предаване на управлението на ресурс “/show/evacuation/”. Чрез този ресурс се адресира документа-шаблон “show-evacuation-data.html”.

3.3.2 Индексен файл

Индексният файл съдържа етикетите за зареждане на CSS кода и необходимите JavaScript библиотеки (виж Фиг. 12-9).

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    // Други мета тагове
    <title>Evacuation</title>
    <link rel="apple-touch-icon" href="assets/icons/apple-touch-icon.png">
    <link rel="stylesheet" href="framework7/framework7-bundle.min.css">
    <link rel="stylesheet" href="css/icons.css">
    <link rel="stylesheet" href="css/app.css">
    <link rel="stylesheet" href="components/loader.css">
</head>

<body>
 <div id="app">
  <!-- Status bar overlay for fullscreen mode-->
  <div class="statusbar"></div>
  <!-- Your main view, should have "view-main" class -->
  <div class="view view-main">
   <!-- Page, data-name contains page name which can be used in callbacks -->
   <div class="page" data-name="home">
    <!-- Top Navbar -->
    <div class="navbar">
     <div class="navbar-bg"></div>
     <div class="navbar-inner">
      <div class="title">Evacuation</div>
     </div>
    </div>
    <!-- Toolbar-->
    <!-- Scrollable page content-->
    <div class="page">
     <div class="page-content main">
      <div class="block">
       <p class="text-color-blue">
        <b>Пример за хибридно мобилно приложерние което получава push известяване
        при активиране на евакуация от сгради. Приложението използва услуги One
        Signal и Firebase, за да реализира получаване на push нотификации.</b>
       </p>
       <p class="text-color-black">
        Не е необходимо приложението да е стартирано, за да получавате известия!
       </p>
       <p class="text-color-gray">
        &copy; RS-SOFT 2022
       </p>
      </div>
     </div>
    </div>
   </div>
  </div>
 </div>
 <!-- Framework7 library -->
 <script src="framework7/framework7-bundle.min.js"></script>
 <!-- App routes -->
 <script src="js/routes.js"></script>
 <!-- App store -->
 <script src="js/store.js"></script>
 <!-- App scripts -->
 <script src="js/app.js"></script>
 <script src="components/loader.js"></script>
</body>

</html>
```

Фиг. 12-9 Индексен файл - index.html

Главният изглед на приложението съдържа само навигационна лента и съдържанието на страницата. Страницата информира потребителите за какво служи приложението.

3.3.3 Визуализиране на данните за евакуация

Страницата за визуализиране на данните от push известията се изгражда динамично. Използва се компонент `card` в който се показват данните от известието (виж Фиг. 12-10). Тези данни се получават след прочитане на предаваните към ресурса параметри – `props.evacuation`.

```
<template>
<div class="page">
 <div class="navbar">
  <div class="navbar-bg"></div>
   <div class="navbar-inner sliding">
    <div class="title">ЕВАКУАЦИЯ ОТ СГРАДАТА</div>
   </div>
  </div>
  <div class="page-content evacuation">
   <div class="evacuation-container">
    <div class="card demo-news-card">
     <div class="card-header">
      <p class="text-color-red">
       <b>${evacuation.title}. ${evacuation.type} - ${evacuation.location}</b>
      </p>
     </div>
     <div class="card-content card-content-padding">
      <p>${evacuation.body}</p>
      <img src="${evacuation.bicon}" style="max-width:100%" />
     </div>
     <div class="card-footer">
      <p class="text-color-gray">${evacuation.timestamp}</p>
     </div>
    </div>
   </div>
  </div>
</div>
</template>
<script>
  export default (props) => {
    const evacuation = props.evacuation;
    return $render;
  }
</script>
```

Фиг. 12-10 Визуализация на данните за евакуация – файл show-evacuation-data.html

3.3.5 Локално хранилище

В локалното хранилище се помнят само данните от последно полученото push известие – свойство `evacuation` (виж Фиг. 12-11).

```
var createStore = Framework7.createStore;
const store = createStore({
 state: {
  evacuation: {}
 },
});
```

Фиг. 12-11 Локално хранилище – файл store.js

3.3.4 Маршрутизация

Използват се три маршрута – към индексния файл, към ресурса “show/evacuation/” и страницата 404.html (виж Фиг. 12-12).

```
var routes = [
  {
    path: '/',
    url: './index.html',
  },
  {
    path: '/show/evacuation/',
    async: function ({ resolve }) {
      var evacuationData = store.state.evacuation;
      resolve(
        {
          componentUrl: './pages/show-evacuation-data.html',
        },
        {
          props: {
            evacuation: evacuationData,
          }
        }
      );
    },
  },
  // Default route (404 page). MUST BE THE LAST
  {
    path: '(.*)',
    url: './pages/404.html',
  },
];
```

Фиг. 12-12 Маршрутизация – файл routes.js

Преди управлението да се предаде на ресурса “show-evacuation-data.html” се изпълнява анонимна функция, която извлича от локалното хранилище данните за евакуация (`evacuationData`). Тези данни се предават като аргумент на ресурса чрез свойство `props`.

3.3.6 Стилово форматиране

Стиловото форматиране се свежда само до промяна на цвета на фона на навигационната лента и цвета на фона на страницата, която показва данните за евакуация (виж Фиг. 12-13). Забележете, че цветът на фона на навигационната лента се променя чрез root селектора `--f7-bars-bg-color`.

```
:root {
    --f7-bars-bg-color: rgba(250, 218, 218, 0.95);
}
.evacuation {
    background-color: rgb(220, 220, 220);
}
```

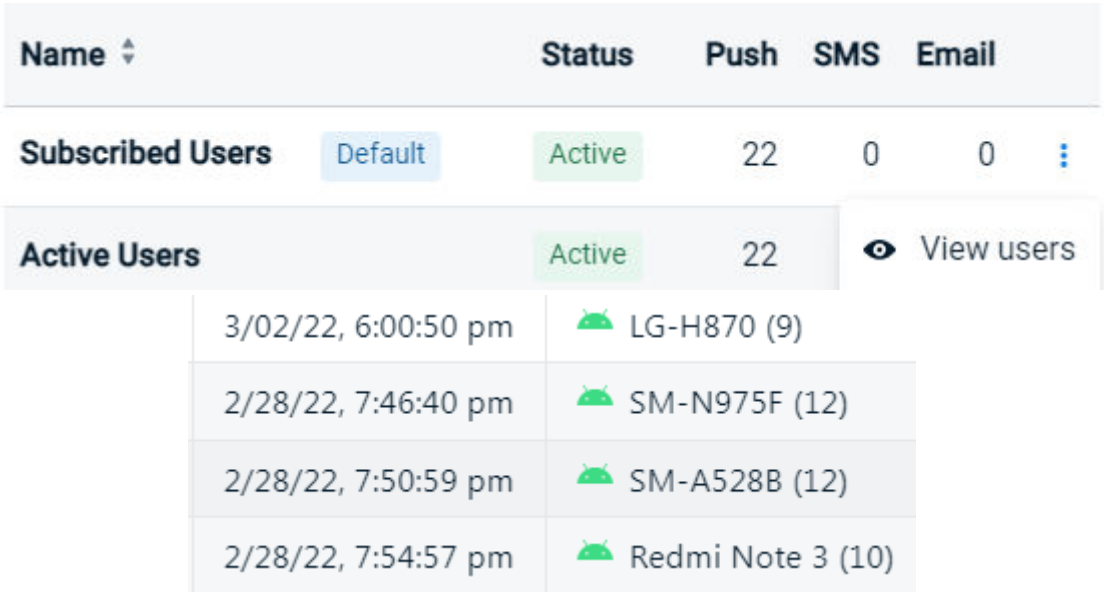
Фиг. 12-13 Стилово форматиране – файл app.css

IV. Изграждане и тестване на приложението

Приложението може да бъде тествано само ако се стартира на *реално* устройство. Причината за това е, че заради плъгина “onesignal-cordova-plugin” приложението не може да се изгради в режим Custom Build Debugger и следователно то не може да бъде тествано с Monaca Debugger. Преди да изградите приложението, реализирате необходимите конфигурации. За целта от Configure изберете “App settings for Android…”. Задайте име на приложението (Evacuation) и име на пакета. Името на пакета трябва да съвпада с името на пакета, зададено при конфигуриране на проекта за Firebase (виж Фиг.

12-2). Задължително трябва да разрешите използването на библиотека AndroidX. Това се изисква от кода на използвания плъгин. Следва избор на икони на приложението и начален екран. Остава да изградите приложението чрез Build -> “Build App for Android…”.

След инсталиране и стартиране на приложението трябва да проверите дали има абонирани потребители за услуга OneSignal. Може да направите това от Dashboard на OneSignal. Изберете името на вашия проект, а от менюто изберете Audience -> Segments. От списъка с потребители от менюто за “Subscribed Users” изберете “View users”. Ще получите списък на всички абонирани потребители (виж Фиг. 12-14).



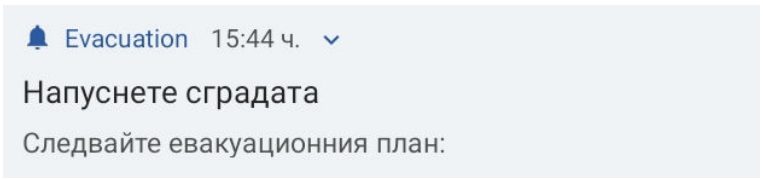
Фиг. 12-14 Проверка за наличие на абонирани потребители

Ако приложението е част от “Subscribed Users” можете да тествате програмния код за показване на данните от push известията. За целта от Messages натиснете бутон “New Push”. Ще получите достъп до форма от която може да изпратите push известие до вашите абонати. Задайте следната информация:

- Име на известието (произволно избрано от вас).
- От Audience изберете “Send to Subscribed Users”.
- От Message задайте стойност за полета Title и Message.
- За Image изберете графичен файл, който съдържа евакуационен план.
- От “Additional Data” създайте две полета - floor и type.

Остава да натиснете бутона “Review and Send”. След като се уверите, че всичко е зададено коректно натиснете бутон “Send Message”. Времето за доставяне на известието е няколко секунди.

В зависимост от версията на ОС ще получите прозорец в който са част от данните от push известието (виж Фиг. 12-15)



Фиг. 12-15 Прозорец на полученото push известие

Изберете с пръст известието. Това трябва да стартира приложението (ако вече не е стартирано) и да се отвори изгледа на документа-шаблон show-evacuation-data.html. на Фиг. 12-16 е показано как се визуализират данните от конкретно push известие.

Фиг. 12-16 Визуализиране на данните от push известие

V. Програмно изпращане на push известия

Ако разработвате реална услуга ще ви трябва API за програмно изпращане на push известия. В случая трябва да използвате OneSignal REST API, за да реализирате програмно пълната функционалност от действия, които услугата предлага:

https://documentation.onesignal.com/docs/onesignal-api

Поддържа се голямо разнообразие на програмни езици чрез които може да реализирате своя сървърен код: PHP, C#.NET, Ruby, Python, Node.js, Perl, Java и др. Ще използваме Node.js, за да реализираме сървърния код чрез който ще изпращаме push известия до мобилното приложение с име Evacuation.

Създайте папка за проекта, например **sendnotification** и следвайте следната последователност от действия.

5.1 Инсталиране на Node.js

Може да свалите необходимия инсталатор за Node.js от следния адрес:

https://nodejs.org/en/download/

Изберете инсталатор за вашата операционна система. При ОС Windows е най-добре да свалите съответния msi инсталатор, например:

node-v14.17.3-x86.msi

С тази инсталация ще получите съответната версия на Node.js, както и версия на Node Package Manager (NMP). Приложението NPM е по подразбиране мениджър за инсталиране на Node.js програмни пакети. Състои се от клиент за командния ред, също наречен NPM, и онлайн база данни с публични и платени частни пакети, наречена регистър NPM. Достъпът до регистъра се осъществява чрез клиента, а наличните пакети могат да бъдат разглеждани и търсени чрез уебсайта на NPM Inc. Направете проверка дали инсталацията е успешна. За целта от командния ред изпълнете следните команди:

```
node –v
npm –v
```

Като резултат трябва да получите версиите на инсталираното програмно осигуряване.

5.2 Инсталиране на пакет onesignal-node

Ще използваме пакет onesignal-node, за да получим достъп до OneSignal REST API. Направете папката на проекта (sendnotification) текуща папка и изпълнете следната команда, за да инициализирате проекта:

```
npm init
```

Трябва да въведете име и версия на проекта, описание на проекта, входна точка (push.js), ключови думи, кратко описание и име на Git хранилище (незадължително). Тази информация ще бъде въведена във файла **package.json**.

За да инсталирате пакета onesignal-node изпълнете следната команда:

```
npm install onesignal-node --save
```

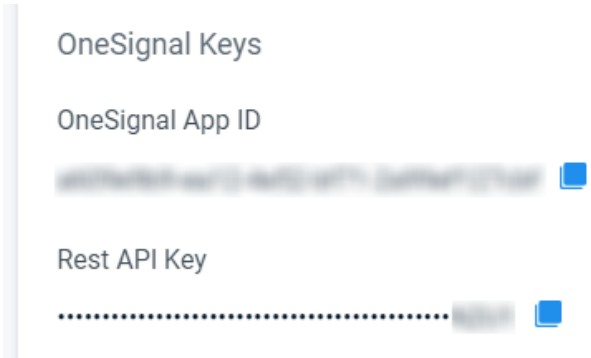
По този начин необходимия пакет ще бъде интегриран към проекта.

5.3 Сървърен код

Създайте файл **push.js** и го запишете в папката на проекта. Съдържанието на този файл е показано на Фиг. 12-17. Чрез метод `require` се задава, че програмния код ще използва модул onesignal-node. Програмното изпращане на нотификации изисква задаване на идентификатора на OneSignal проекта (appId) и идентификатора за REST API (apiId). Тези идентификатори може да получите от

https://app.onesignal.com/

Изберете вашия проект, а след това Settings -> “Keys & IDs”:



Копирайте тези данни и ги прехвърлете като стойност за константи `appId` и `apiId`.

```
const OneSignal = require('onesignal-node');
const appId = 'Идентификатор за приложението';
const apiId = 'Идентификатор за REST API';

const client = new OneSignal.Client(appId, apiId);
const notification = {
 headings: {
   'en': 'Напуснете сградата',
 },
 contents: {
   'en': 'Следвайте евакуационния план',
 },
 data:  {
   'floor': 'етаж 3',
   'type': 'Пожар'
 },
 big_picture: 'https://domain/picture-name.png',
 included_segments: ['Subscribed Users'],
};

client.createNotification(notification)
 .then(response => {
    console.log('Съобщението бе изпратено успешно.');
 })
 .catch(error => {
    console.log('Код на грешката: ' + error.statusCode);
    console.log(error.body);
 });
```

Фиг. 12-17 Програмен код за изпращане на push известия – файл push.js

За всички действия, които изискват идентификатор на приложението и ключ за REST API (създаване на известие, добавяне на устройство, създаване на сегмент), трябва да използвате OneSignal.Client, за да се създаде обект `client`. Чрез този обект ще можем да извикаме метод `createNotification` с който да създадем и изпратим push известие. Този метод изисква един аргумент – JSON обект `notification`. Обект `notification` съдържа полетата чрез които се описват мета данните на съобщението:

- **headings** – заглавие на съобщението (title).
- **contents** – съдържание на съобщението (body).
- **data** – JSON обект, описва потребителските данни във формат ключ: стойност.
- **big_picture** – URL към изображението с евакуационния файл в png или jpeg формат. Задължително е изображенията да са хоствани в хранилище, което е достъпно чрез HTTPS (не HTTP).
- **included_segments** – масив, в който се описват всички получатели на съобщението. При конкретният пример е указана стойност “Subscribed Users” и това означава, че съобщението ще бъде изпратено до всички абонати.

5.4 Изпълнение на сървърния код

За да стартирате сървърния код от конзолно ниво, изпълнете следната команда:

```
node push.js
```

Ако няма грешки, след стартиране на програмния код трябва да получите push известие.

Литература

1. Росен Иванов, JavaScript: Въведение в програмирането, Издателство "Фабер", 2018.
2. Росен Иванов, Нерелационни бази данни – ръководство за лабораторни упражнения, ISBN 978-954-683-642-7, 2021.
3. Axel Rauschmayer, JavaScript for impatient programmers, ISBN 978-1-09-121009-7, 2019.
4. Chris Anderson, Jan Lehnardt, Noah Slater, CouchDB: The Definitive Guide, O'Reilly Media, Inc., ISBN: 978-0-596-15589-6, 2010.
5. Christopher Bienko, Marina Greenstein, Stephen E Holt, Richard T Phillips, IBM Cloudant: Database as a Service Fundamentals, IBM RedPaper, 2015.
6. Christopher D. Bienko, Marina Greenstein, Stephen E Holt, Richard T Phillips, IBM Cloudant Database as a Service Advanced Topics, IBM RedPaper, 2015.
7. Dormann, A. Ionic 5: Create awesome apps for iOS, Android, Desktop and Web, D&D Verlag Bonn, 2020.
8. John M. Wargo, Apache Cordova 4 programming, Addison-Wesley, ISBN 978-0-13-404819-2, 2015.
9. Kerri Shotts, PhoneGap 2.x Mobile Application Development HOTSHOT, Packt Publishing Ltd., ISBN 978-1-84951-940-3, 2013.
10. Panhale, M., Beginning Hybrid Mobile Application Development, Apress, 2016.
11. Rap Payne, Beginning App Development with Flutter: Create Cross-Platform Mobile Apps, Apress, 2019.
12. Richard Clark, Oli Studholme, Christopher Murphy and Divya Manian, Beginning HTML5 and CSS 3, Apress, ISBN 978-1-4302-2874-5, 2012.
13. Rohit Ghatol, Yogesh Patel, Beginning PhoneGap: Mobile Web Framework for JavaScript and HTML5, Apress, ISBN 978-1-4302-3903-1, 2012.
14. Thomas Myer, Beginning PhoneGap, John Wiley & Sons, ISBN: 978-1-118-15665-0, 2012.

Програмиране за мобилни устройства

Автор

Росен Стефанов Иванов

Рецензент

проф. дмн Стоян Недков Капралов

Националност българска

Първо издание

Предпечатна подготовка

Росен Стефанов Иванов

http://kst.tugab.bg/mobile