СОФИЙСКИ УНИВЕРСИТЕТ „СВ. КЛИМЕНТ ОХРИДСКИ“

**ФАКУЛТЕТ ПО МАТЕМАТИКА И ИНФОРМАТИКА**

**Катедра „Информационни Технологии“**

**ДИПЛОМНА РАБОТА**

За получаване на образователно-квалификационна степен „магистър“

На Панайот Цонев Цанков, фак. № 23272

Студент от магистърска програма „Разпределени Системи и Мобилни Технологии“

Тема:

„Проектиране, разработка на много-платформена мобилна информационна система за прилагане на гъвкави методологии в проекти с отворен код“

Научен ръководител:

проф. д-р Васил Георгиев, катедра Компютърна Информатика

Съдържание

[Увод 3](#_Toc463917721)

[ГЛАВА ПЪРВА 4](#_Toc463917722)

[Функционален обзор - гъвкави методологии и инструменти 4](#_Toc463917723)

[1. Манифест на гъвкавата методология 4](#_Toc463917724)

[2. Скръм 6](#_Toc463917725)

[3. Канбан 6](#_Toc463917726)

[4. Съпоставка между Скръм и Канбан 8](#_Toc463917727)

[5. Разработване на проекти с отворен код 14](#_Toc463917728)

[6. HuBoard 16](#_Toc463917729)

[7. Team pulse 17](#_Toc463917730)

[8. Aha! 19](#_Toc463917731)

[ГЛАВА ВТОРА 20](#_Toc463917732)

[Технологичен обзор - технологии за разработка на много-платформени мобилни приложения 20](#_Toc463917733)

[9. Apache Cordova 20](#_Toc463917734)

[10. Xamarin 22](#_Toc463917735)

[11. NativeScript 25](#_Toc463917736)

[12. ReactNative 28](#_Toc463917737)

[13. Други 30](#_Toc463917738)

[ГЛАВА ТРЕТА 31](#_Toc463917739)

[Функционално проектиране на мобилното приложение 31](#_Toc463917740)

[14. Случаи на употреба при гъвкави методологии 32](#_Toc463917741)

[15. Случаи на употреба на мобилното приложение 33](#_Toc463917742)

[16. Дейности извършвани чрез приложението 34](#_Toc463917743)

[17. Потребителски интерфейс 36](#_Toc463917744)

[ГЛАВА ЧЕТВЪРТА 42](#_Toc463917745)

[Технологично проектиране на мобилното приложение 42](#_Toc463917746)

[18. Компоненти на системата 43](#_Toc463917747)

[19. Angular 2 Github service 44](#_Toc463917748)

[20. OAuth 46](#_Toc463917749)

[21. Проектиране и разработка на модул за достъп OAuth автентикация 46](#_Toc463917750)

[Иницииране 48](#_Toc463917751)

[Обработка при успех 48](#_Toc463917752)

[22. Потребителски интерфейс 52](#_Toc463917753)

[23. Начално табло 53](#_Toc463917754)

[24. Новини 55](#_Toc463917755)

[25. Хранилища 58](#_Toc463917756)

[26. Въпрои и Събития 62](#_Toc463917757)

[27. Въпроси (задачи) 69](#_Toc463917758)

[Заключение 73](#_Toc463917759)

[Източници 75](#_Toc463917760)

# Увод

Мобилните платформи на Android и iOS разделят сериозен пазар за мобилни приложения и отварят ниша за платформи за разработка със споделен код на приложения насочени и към двете еко системи.

В условията на остра конкуренция, компаниите разработващи такива платформи използват всички възможни средства, за да засегнат максимална част от разработчиците на мобилни приложения. Към този микс се включват и разработчици, които се пренасочват от десктоп и уеб сферите към мобилната сфера. В резултат наблюдаваме все по-голям процент от платформите да отварят кода си в опит да привлекат внимание.

Следва да разгледаме какви са условията за разработка с отворен код. Инструментите за работа и прилагане на гъвкави методологии в GitHub. Възможностите за разработване на много-платформено приложение. И ще разгледаме как е възможно да се проектира и разработи приложение, което да улесни комуникацията в GitHub.

# ГЛАВА ПЪРВА

# Функционален обзор - гъвкави методологии и инструменти

## Манифест на гъвкавата методология

През февруари 2001 г. 17 разработчици[[1]](#footnote-1) на софтуер се срещат в курорта Snowbird, Юта, за да обсъдят гъвкавите методи за разработка. Публикуват Манифест на гъвкавите методологии, който дефинира подходите, сега известни като гъвкави методи за разработка на софтуер. Част от авторите на манифеста създават The Agile Alliance, неправителствена организация, която насърчава разработката на софтуер в съответствие с принципите на манифеста.

Манифестът на гъвкавите методологии гласи следното[1]:

Откриваме по-добри методи за разработване на софтуер, като ги практикуване и помагаме на другите да го правят. В процеса на работа стигнахме до следните важни изводи:

* Хората и комуникацията стоят над процесите и инструментите
* Работещият софтуер е над подробната документация
* Сътрудничеството с клиента е над преговорите по време на сключване на договора
* Адресирането на промените стои над следването на плана

Това означава, че въпреки че десните елементи имат своята стойност, ние ценим точките отляво повече.

Значенията на елементите от лявата част на горните твърдения са описани тук[2]:

* Хора и комуникация - според гъвкавата методология самоорганизацията и мотивацията са важни, като например работата в съвместно помещение и [програмиране по двойки](https://bg.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%B8%D1%80%D0%B0%D0%BD%D0%B5_%D0%BF%D0%BE_%D0%B4%D0%B2%D0%BE%D0%B9%D0%BA%D0%B8).
* Работещ софтуер - работещият софтуер е по-полезен по време на срещи с клиента, отколкото представянето на документация по проекта.
* Сътрудничество с клиента – софтуерните спецификации не могат да бъдат изцяло изготвени в началото на проекта, затова непрекъснатото участие на всички заинтересовани страни е ключово.
* Адресиране на промените – гъвкавата методология за разработка се фокусира над бърза реакция към промените и непрекъснато развитие

The Agile Manifesto се основава на дванадесет принципа:

* Удовлетворение на клиентите чрез бърза доставка на полезен софтуер
* Промяна в спецификациите е възможна, дори и в късните фази на проекта
* Често предоставяне на работещ софтуер (в периоди от седмици, а не месеци)
* Работещият софтуер е основната мярка за напредък
* Устойчиво развитие, което успява да поддържа постоянно темпо
* Тясно, ежедневно сътрудничество между бизнес служители и разработчици
* Разговорите лице в лице са най-добрата форма на комуникация (съвместна локация)
* Проектите се изграждат около мотивирани хора, на които се има доверие
* Непрекъснато внимание към техническо съвършенство и добър дизайн
* Простота - изкуството максимално количество работа да бъде пропусната, е от съществено значение
* Самоорганизиращи се екипи
* Редовна адаптация към променящи се обстоятелства

През 2005 г. група начело с Алистър Кокбърн и Джим Хайсмит написва допълнение към принципите за [управление на проекти](https://bg.wikipedia.org/wiki/%D0%A3%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BD%D0%B0_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8), наречено „Декларацията на взаимозависимост”, със съвети за управлението на софтуерни проекти в съответствие с гъвкави методи за разработка[3].

През 2009 г. движение, ръководено от Robert C. Martin разширява принципите за [разработка на софтуер](https://bg.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D0%BD%D0%B0_%D1%81%D0%BE%D1%84%D1%82%D1%83%D0%B5%D1%80) според гъвкавата методология (Software Craftsmanship Manifesto) в съответствие с професионалната етика и майсторство.

Като резултат произтичат множество практики, като две от най-известните са Скръм[[2]](#footnote-2) и Канбан[[3]](#footnote-3).

## Скръм

Скръм е платформа за разработване и поддръжка на комплексни продукти[4]. Ken Schwaber и Jeff Sutherland разработват Скръм. Дефиниция на Скръм:

Платформа чрез която хората могат да адресират комплексни адаптивни проблеми, като продуктивно и креативно доставят продукт с високо качество. Скръм е: лек, лесен за разбиране, труден за овладяване.

Скръм на кратко[5]:

* Разделете вашата организация на малки, мултифункционални, самоорганизиращи се екипи.
* Разделете вашата работа и я организирайте в списък от малки задачи, водещи до конкретни резултати. Сортирайте списъка по приоритети и оценете приблизително усилието за всяка една задача.
* Разделете времето на кратки интервали с фиксирана дължина (обикновено 1-4 седмици), с възможност за демонстрация в края на всяка итерация на работеща функционалност, потенциално готова за предаване на клиента.
* Оптимизирайте плана си за продуктовата версия и приоритизирайте отново заедно с клиента на базата на идеите, събрани при анализа на резултата след всяка итерация.
* Оптимизирайте процеса, провеждайки ретроспекция след всяка итерация.

И така вместо голяма група, която прекарва дълго време изграждайки нещо голямо, вие имате малък екип, който за кратко време изгражда нещо малко. Но интегрира постоянно, за да се види цяла картинка.

## Канбан

В основата на производствената система на Toyota е абсолютното елиминиране на отпадък. Двата стълба поддържащи системата са[6]:

* Производство точно-на-време
* Интелигентна автоматизация[[4]](#footnote-4)

Канбан на кратко[5]:

* Визуализирайте работния поток
  + Разделете работата на парчета, напишете всяко парче на карта и поставете на стената.
  + Използвайте колони с имена, за да илюстрирате фазата, в която се намира всяка една задача в работния поток.
* Ограничете броят недовършени задачи[[5]](#footnote-5)– определете лимит за това колко започнати задачи може да има във всяка една графа/колона на работния поток.
* Измервайте време за изпълнение на целия процес, оптимизирайте процеса, така че времето за изпълнение да е възможно най-малко и предвидимо.



Фигура 1 Канбан табло[5]

## Съпоставка между Скръм и Канбан

Скръм се опитва да разбие стандартния процес на кратки спринтова, с типична продължителност от една до четири седмици, произвеждайки продуктови подобрения готови за клиенти всеки спринт и настройката спрямо обратна връзка от клиенти в края на всеки спринт. Планирането трябва да се обновява за всеки спринт. Потокът от готови за клиенти подобрения с незабавна обратна връзка е по-чест, но все пак не е постоянен.

Канбан е построен за постоянна и непрекъсната доставка до клиенти. Потокът и качеството се контролират внимателно за да бъдат открити проблеми незабавно. Лимитира броя на задачите в прогрес и по този начин не допуска събиране на дълг и позволява реагирането на промени на базара ежедневно[7].

**Скръм е доста по-задължаващ от Канбан**

Можем да сравним инструментите чрез анализирането на това какви правила имат. Задължаващ означава „повече правила за спазване“, а адаптивен означава „по-малко правила за спазване“:



Фигура 2 Методологии съпоставен спрямо брой правила [5]

RUP[[6]](#footnote-6) е силно ограничаващ – има над 30 роли, над 20 дейности, 70 артефакта; доста голямо количество информация.

XP е с доста повече правила в сравнение със Скръм. Включва се част от Скръм плюс набор от строго специфицирани практики при разработването на софтуер като програмиране, базирано на тестове и програмиране по двойки[[7]](#footnote-7).

**Скръм и Канбан табла**

В Скръм продуктовият списък за спринта съдържа задачите, които трябва да се изпълнят по време на текущия спринт. Често това се визуализира чрез карти на стената и се нарича Скръм дъска или дъска за задачи.



Фигура 3 Скръм и Канбан дъска [5]

Разликата между двете примерни дъски на *фиг 3* – малкото число 2 по средата на колоната на Канбан дъската, означава „Не може да има повече от 2 задачи в тази колона в даден момент”.

В Скръм няма правило, което да пречи на екипа да сложи по едно и също време всички задачи в колоната „В прогрес“.

**И двата са емпирични**

Скръм и Канбан са емпирични, в смисъла че от вас се очаква да експериментирате с процеса и да го адаптирате към вашите условия. В действителност вие трябва да експериментирате. Нито Скръм, нито Канбан дават всички отговори - те просто ви дават един базов набор от ограничения, с който да управлявате подобренията на процесите си.

**Списъкът със задачи в Скръм трябва да се побере в спринта**

И Скръм, и Канбан са базирани на инкрементална разработка, т.е. разбиване на малки парчета.

В Скръм работата, която ще се извършва по време на спринта, се планира по време на планирането на спринта. Планирането на спринта отговаря на следните въпроси:

* Какво може да бъде доставено в инкремента, получен в резултат от предстоящия Спринт?
* Как ще бъде свършена необходимата за доставянето на инкремента работа?

Един Скръм екип ще обещае да свърши точно толкова задачи, колкото мисли, че може да завърши в рамките на един спринт[4].

Канбан екипите се опитват да минимизират времето за изпълнение и нивото на потока и това индиректно се превръща в стимул задачите да се разбиват на сравнително малки парчета. Но няма изрично правило, според което задачите трябва да са достатъчно малки, за да се впишат в определено време. На една и съща дъска можем да имаме една задача, която отнема месец, и друга, която отнема 1 ден[5].

**Скръм изисква оценка и темпо**

В Скръм се очаква екипите да оценяват в относителни размери всяка задача, която обещават да изпълнят. Чрез събирането на размерите на всички изпълнени задачи в края на спринта, ние получаваме темпото. То е мярка за капацитета му – колко работа може да свърши за един спринт.

Добре е да се знае средното темпо, защото тогава могат да се правят реалистични прогнози за това кои задачи могат да се изпълнят в предстоящия спринт.

В Канбан оценяването не е задължително. Ако се наложи да се дават обещания, трябва да се прецени как да се подсигури предвидимост.

**При Скръм е задължителен приоритизираният продуктов списък**

В Скръм имаме винаги приоритизиран и сортиран продуктов списък, а промените по приоритетите могат да станат чак в следващия, а не в текущия спринт. В Канбан вие можете да изберете какъвто и да е начин за приоритизация.

**При Скръм са задължителни ежедневните срещи**

Ежедневният Скръм е 15-минутно ограничено във времето събитие, в рамките на което Екипът от Разработчици синхронизира дейността си и създава план за следващите 24 часа. Това се постига чрез инспектиране на работата, свършена от последния Ежедневен Scrum насам, и прогнозиране на работата, която ще се приключи преди следващия[4].

**При Скръм е задължителна графиката на оставащата работа**

Графиката на оставащата работа за спринта ежедневно показва колко работа остава до края на текущата итерация.



Фигура 4 Примерна графика на оставаща работа [8]

Мерната единица на вертикалната ос е същата като мерната единица, която се използва за задачите от спринта. Обикновено часове или дни (ако екипът разбива елемент от продуктовия списък за спринта на задачи) или сюжетни-точки (ако екипът не го прави).

**Обобщение на сравнението на Скръм с Канбан**

Прилики [5]:

* И двата са Лийн и Аджайл.
* И двата използват система на изтегляне за организация на работата.
* И двата ограничават броят на започнати и недовършени задачи.
* И двата използват прозрачност, за да има подобрения в процеса.
* И двата се фокусират върху доставянето на работещ софтуер рано и често.
* И двата се основават на самоорганизиращи се екипи.
* И при двата има разбиване на работата на парчета.
* И при двата планът за версията на продукта се оптимизира постоянно на базата на емпирични данни (темпо/време за изпълнение).

|  |  |
| --- | --- |
| Скръм | Канбан |
| Ограничените по време итерации са задължителни. | Ограничените по време итерации са опционални. Може да има отделни тактове за планиране, изкарване на версия и подобрение на процеса. Може да е провокиран от събитията, вместо от ограничения във времето. |
| Екипът обещава да свърши определена работа в дадена итерация. | Обещаването на определена работа за определено време не е задължително. |
| Използва се темпо като метрика за планиране и за подобрения в процеса. | Използва се време за изпълнение като метрика за планиране и за подобрения в процеса. |
| Задължително е да има мултифунционални екипи. | Мултифункционалните екипи са опционални. Позволени са специализирани екипи. |
| Работата трябва да се разбие на парчета, така че да бъдат завършени в един спринт. | Няма препоръка за определен размер на задачите. |
| Задължително е използването на графика на оставащата работа. | Няма диаграма или графика, която да е необходимо да се ползва. |
| Ограничаване на броя недовършени задачи косвено (за спринт) | Ограничаване на броя на недовършени задачи директно (за фаза в потока на работа) |
| Оценяването е задължително. | Оценяването не е задължително. |
| Не може да се добавят задачи по време на итерация. | Може да се добавят задачи винаги, когато има свободен капацитет. |
| Продуктовият списък за спринта е собственост на един екип. | Дъската на Канбан може да се споделя от различни екипи или хора. |
| Има 3 задължителни роли (Продукт Оунър, Скръм Мастър и Екип) | Няма никакви задължителни роли. |
| Скръм дъската се изчиства между спринтовете. | Канбан дъската е постоянна. |
| Задължителен е приоритизирания продуктов списък. | Приоритизацията е опционална. |

Таблица 1 Прилики и разлики на Скръм и Канбан [5]

## Разработване на проекти с отворен код

**Катедралата и базарът**

Линукс е подмолен. Кой би си помислил само преди пет години (1991), че от хакерските занимания в свободното време на няколко хиляди разработчици, пръснати из планетата, и свързани единствено от тънките нишки на Интернет, като на магия ще се получи операционна система от световен клас?

Стилът на разработка на Линус Торвалдс — „пускай рано и често; възлагай на други желаещи всичко, което можеш; бъди открит до степен на безразборност“, ми дойде изненадващо. Нямаше го тихото, благоговейно катедрално разработване — Линукс обществото по-скоро ми приличаше огромен шумен базар с различни планове и подходи (доста точно уподобени на от местата с Линукс архиви, които приемат софтуер от всеки), от който само по чудо би могла да се роди съгласувана и стабилна операционна система.[9]

В традиционния модел на развитие (който Ерик Реймънд нарича Катедралата), задачи и роли са ясно дефинирани и произтичат от целите на проекта зададени предварително. Разработването се случва централизирано, разделяйки участниците на три групи:

* Архитекти на системата
* Управители на проекта
* Изпълнители

[Фред Брукс](https://bg.wikipedia.org/w/index.php?title=%D0%A4%D1%80%D0%B5%D0%B4_%D0%91%D1%80%D1%83%D0%BA%D1%81&action=edit&redlink=1) застъпва този модел и отива по-далеч, казвайки, че за да се запази целостта на архитектурната система, проектирането на системата трябва да бъде направено от колкото е възможно по-малко архитекти[10].

От друга страна в моделът на Базар, ролите не са ясни и могат да варират през времето за изпълнение на проекта. Важните качества на Базарът са постоянната обмяна на знание, поток от идеи от които всички могат да се възползват, всичко което кара проекта да еволюира и да се развива с бързо темпо.

В този модел, ролите не са ясно дефинирани. Gregorio Robles показва, че софтуер, разработен по модела базар трябва да има следните черти[11]:

**Потребителите трябва да бъдат третирани като съразработчици**

За да бъде отворен работния процес до максимум, така че заинтересованите разработчици да бъдат интегрирани. Повечето съразработчици откриват повече възможности за продукта – да еволюира по-бързо, да се добавя функционалност в по бързи темпове.

В закона на Линус се твърди:

"Предоставяйки достатъчно очи всички грешки стават видими."[9]

Това означава, че ако много потребители имат достъп до изходния код, те в крайна сметка ще намерят всички грешки и да предложат как да се оправят. Трябва да се вземе предвид, че някои потребители са с напреднали програмни умения, а освен това, компютърът на всеки потребител осигурява допълнителна среда за тестване. Тази нова среда за тестване предлага способността да се намери и да определи нов "бъг".

**Ранните версии**

Първата версия на софтуера трябва да бъде пусната, макар и с лимитирана функционалност, възможно най-рано. Още в момента в който има нещо достатъчно за презентиране и функциониращо. По този начин се увеличава вероятността да се намерят съразработчици заинтересовани да участват.

**Честата интеграция**

Развиването с малки стъпки позволяват проекта да расте инкрементално, в същото време позволява и паралелно развитие, отстраняване на проблеми и адаптиране към обратна връзка от съразработчици и потребители. Промените по кода следва да бъдат интегрирани (обединени в обща база код) възможно най-често, така че да бъде избегната необходимостта от оправянето на голям брой грешки в края на жизнения цикъл на проекта. Някои проекти с отворен код имат нощна разработка, където интегрирането се извършва автоматично на база на дневната.

**Няколко версии**

Трябва да има най-малко две версии на софтуера. Трябва да има версия на разработчика с повече функции и стабилна версия с по-малко възможности. Версията на разработчика е за потребители, които искат незабавното използване на най-новите функции, и са готови да приемат риска от използването на код, който все още не е старателно тестван. Потребителите могат след това да действат като съразработчици – да докладват проблеми и да осигуряват корекции за тях.

**Висока модулност**

Силното разбиването на модули позволява парализирането на работа и преизпълването на код. Все пак разпространението, и отстраняването на грешки се паралелизират повече.

**Структура за динамично взимане на решения**

Въпреки че като цяло решенията се вземат от разработчиците, които пишат код, обикновено съществува организационна структура, която взема стратегическите решения. Организационната форма зависи от типа на проекта, от техническите му характеристики – размер, брой потребители, брой разработчици, както зависи и от исторически причини, баланс на интереси и др.[11]

## HuBoard

HuBoard[12] е разработен върху публичното API на GitHub. Задачите във HuBoard са задачите в GitHub, така не се налага да се преодоляват проблеми със синхронизация. Адаптивната дъска на HuBoard дава на разработчиците незабавна обратна връзка на задачите в целия си напредък.

Дъската поддържа гъвкаво филтриране, с цел ограничаване на видимите задачи за по лесен избор.

Асоциирането на разработчик със задача става чрез плъзване на аватар на разработчика върху задачата. Приоритетът на задачите се запазва чрез метаданни в описанието и в GitHub. Подредбата и състоянието може да се променя чрез придърпване на отделните карти по дъската.

Интеграцията с GitHub се извършва, като по конвенция се заделят етикети в GitHub и преместването на карта променя асоциираните с нея етикети.

Като разширения HubBoard може да се интегрира с Slack, HipChat, Gitter.

## Team pulse

TeamPulse е гъвкав инструмент за управление на проекти, който помага на екипи да планират и разработват софтуерни продукти. Това е система, която е разработена да се инсталира на сървъри на употребяващата я компания, и е с потребителски портал за ангажиране на клиенти и заинтересовани страни.

TeamPulse може да бъде използван от всяка организация, която трябва да управлява множество софтуерни проекти. Може да отговори на нуждите на малките предприятия, както и на организации на корпоративно ниво.[13]

**Създаване на Работа**

За да започнете работа с TeamPulse първо трябва да създадете проект или да въведете съществуващ от Team Foundation Server.

Обектите описващи работа са разделени на няколко нива. Високо ниво – бизнес изисквания, съдържат множество потребителски сценарии. Обикновено собственикът на продукта описва целите на проекта чрез тях. Ниско ниво – потребителски сценарии, в TeamPulse се използват описание на изисквания и функционалност, която ще е достъпна за крайните потребители. Задачи – софтуерните разработчици, следвайки бизнес изискванията, потребителските сценарии, вътрешни потребности или проблеми появили се в продукта, определят задачите произтичащи от тях.[13]

**Планиране**

Собственникът на продукта може да редактира приоритетно подреден списък с бизнес изисквания, да създава итерации и цикли за работа и да асоциира работа с работен цикъл и планиран пуск на продукта.

При започване на нов цикъл, разработчиците разбиват бизнес изискванията на конкретни задачи, които да бъдат приблизително оценени и приоритизирани.

За оценка има гъвкава система която позволява всяка задача да бъде оценена с оптимистична, вероятна и песимистична стойност, от които се изчислява автоматично стандартно отклонение - Техниката за преглед и оценка на програми, или PERT[[8]](#footnote-8).

PERT е техника за планиране и изграждане на график на проект. Много оптимизационни проблеми включват някакъв аспект от планирането по време на дейности, които могат да се изпълняват последователно, паралелно, или зависят по време от използването на критичен ресурс. PERT диаграмите за разбирането и формулирането на такива проблеми.[14]

В TeamPulse се препоръчва [Бета разпределение](https://bg.wikipedia.org/w/index.php?title=%D0%91%D0%B5%D1%82%D0%B0_%D1%80%D0%B0%D0%B7%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5&action=edit&redlink=1), където по минимум (5% процентов [квантил](https://bg.wikipedia.org/wiki/%D0%9A%D0%B2%D0%B0%D0%BD%D1%82%D0%B8%D0%BB)), средно (50% квантил) и максимум (95% квантил) се определя очакваната продължителност или цена по опростена формула за очаквана стойност:

Според [Закона за големите числа](https://bg.wikipedia.org/wiki/%D0%97%D0%B0%D0%BA%D0%BE%D0%BD_%D0%B7%D0%B0_%D0%B3%D0%BE%D0%BB%D0%B5%D0%BC%D0%B8%D1%82%D0%B5_%D1%87%D0%B8%D1%81%D0%BB%D0%B0) за много големи проекти, акумулираното разпределение за целия проект клони към [нормално](https://bg.wikipedia.org/wiki/%D0%9D%D0%BE%D1%80%D0%BC%D0%B0%D0%BB%D0%BD%D0%BE_%D1%80%D0%B0%D0%B7%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5).

**Следене на напредъка на разработката**

Рискът за цялостна оценка е по-малък при по-голям брой задачи. Оценките не се задават в дни, а в точки. По време на работа софтуерните разработчици записват по кои задачи, колко време са работили, без да бъдат притискани. По този начин се събира статистика, която позволява в бъдеще по оценките на задачите, да бъде изчислено време за работа. Също така в реално време да се следи прогресът по заданието.[13]

**TFS синхронизиране**

Продуктът е тясно обвързан с TFS. Има последвала интеграция с GitHub. При работа с продукта ако последват промени директно в TFS системата и TeamPulse се наблюдават разлики за малък период от време.

## Aha!

Управлението на продукта е важна организационна роля в софтуерна компания. За управителя на продукт често се смята изпълнителният директор и е отговорен за стратегията, планът за развитие и бизнес изискванията към продукта.[15]

Aha! е ориентиран по-скоро към по-високо ниво на управлението на продукт, и не толкова към самите екипи от разработчици.

Инструментът поставя бизнес целите на първо място, управителите могат да задават визията на продукта и да дефинират, как очакват печалба от съответствието на ключови стратегически момент и качествата на продукта и издаването на нови програмни версии.

Собствениците на продукт могат лесно да създават, поддържат обновени и споделят план за развитието на продукта. Да позволят на клиенти, служители и други заинтересовани страни да представят идеи и с това реално да въздействат върху развитието на продукта. Всички виждат на едно място как се визуализира приоритета на целите и могат да насърчават тези, които имат значение.[15]

Aha! има интеграция с GitHub, но работи с идеи на много по високо ниво. Съответно много от реалните задачи в GitHub не са директно свързани с идея в Aha!

# ГЛАВА ВТОРА

# Технологичен обзор - технологии за разработка на много-платформени мобилни приложения

## Apache Cordova

Apache Cordova е платформа с отворен код за разработване на мобилни приложения. Позволява на разработчиците да използват уеб стандартни технологии – HTML5, CSS3 и JavaScript за много-платформени разработки. Приложенията се изпълнява в обвивка за всяка платформа, и разчитат на съобразени със стандарт API за достъп до всяко от свойствата на хардуерното устройство като сензори, данни, статус на мрежата и др.

Може да използвате Apache Cordova ако сте:

* Разработчик на мобилно приложение и искате да го разширите, за да е достъпно от повече от една платформа. Без да се налага да го пренаписвате с езика и инструментите за всяка от тях.
* Уеб разработчик и искате да дистрибутирате уеб приложението си в порталите на съответните мобилни платформи.
* Разработчик на мобилни приложения и сте заинтересуван да смесите присъщите за мобилната платформа компоненти с WebView[[9]](#footnote-9), който може да достъпва функциите на мобилното устройство, или искате да разработите модул между WebView и мобилните компоненти.

Няколко компонента изграждат приложение с Cordova. Фигура 5 показва изглед от високо ниво на архитектурата на Cordova приложението.



Фигура 5 Архитектура на Cordova [16]

#### **WebView**

с добавена функционалност от Cordova, може да реализира цялостния интерфейс на приложението. На някой платформи, то може да е и компонент от интерфейс смесен със стандартните за платформата компоненти.

#### **Web App**

На това място се съдържа кодът на приложението. Приложението само по себе си е имплементирано като уеб страница. По подразбиране се отваря страницата index.html, която реферира CSS, JavaScript, картинки, медия и други ресурси нужни за работата му. Приложението се изпълнява във WebView облечено в мобилно приложение, което дистрибутирате на порталите на съответните мобилни платформи.

Контейнерът има много важен файл – config.xml. Той съдържа информация за приложението и специфични параметри, например дали приложението отговаря на смени в ориентацията на мобилното устройство.

#### **Модули**

Модулите са съществена част от екосистемата на Cordova. Те предоставят интерфейс за комуникация между Cordova и присъщите за платформата компоненти, както и достъп до стандартни за мобилното устройство API. Това позволява и извикване на JavaScript от съответния за платформата език.

Apache Cordova проектът поддържа набор от модули наречени Core Plugins. Те поддържат достъп до по-често използваните API като тези, за работа с батерия, камера, списъци с контакти и т. н.

В допълнение към основните модули, има множество написани от трети страни. Могат да бъдат намерени чрез търсачката за модули на Cordova или в npm (node package manager).

## Xamarin

Xamarin е софтуерна компания, собственост на Microsoft, основана през 2011 от инженерите създали Mono. Mono за Android и MonoTouch са много-платформена имплементация на Common Language Infrastructure (CLI) и Common Language Specifications (често наричани Microsoft .NET).

Със споделена C# кодова база, разработчиците могат да използват инструментите на Xamarin за да пишат приложения за Android, iOS и Windows използвайки стандартния за платформата потребителски интерфейс[17].

Xamarin позволява два подхода при споделянето на код между различните мобилни платформи:

* Споделен проект – Тип проект който позволява да бъдат организирани ресурсите и кода на приложенито, като посредством символи за условно компилиране се предоставя платформено-специфична функционалност - Фигура 6.



Фигура 6 Архитектура на споделен проект в Xamarin. [17]

* Portable Class Libraries (PCL) – Библиотека с общ код, като посредством интерфейси се предоставя платформено-специфична функционалност - Фигура 7.



Фигура 7 Архитектура на Xamarin PCL. [17]

Платформата Xamarin се състои от няколко елемента които позволяват да се разработват приложения за iOS и Android [17]:

* Езикът C# - език с познат синтаксис и усъвършенствани качества като Generics, Linq и Parallel Task Library.
* Mono .NET framework – Позволява много-платформена имплементация на широко спектърния Microsoft .NET framework.
* Компилатор – В зависимост от платформата произвежда естествено за мобилната платформа приложение. Компилаторът също така поддържа и множество оптимизации, като автоматичното игнориране на неизползван код.
* IDE – Xamarin Studio IDE и модули за Visual Studio позволяват да създавате и инсталирате Xamarin проекти.

#### **Компилиране**

Въпреки, че Xamarin позволява да пишете C# и да споделяте код между множество платформи, имплементацията варира сериозно между всяка система.

C# кодът стига до мобилното приложение по различен начин за всяка платформна:

* **iOS** – C# се компилира AOT (ahead-of-time, преди да е време) до ARM ассембли. Включва се кодът за .NET framework, като неизползваните класове се игнорират по време на компилирането, с цел намален размер на приложението. Apple не позволяват генериране на изпълним код по време на изпълнение на приложението, затова някой свойства на езика C# не се поддържат.
* **Android** – C# се компилира до IL (Intermediate Language – междинен език) и се пакетира с MonoVM + JIT (Just in time – компилиране точно на време). Неизползваните класове се игнорират. Приложението се изпълнява рамо до рамо с Java/ART (Android runtime) и взаимодейства със вградените компоненти чрез JNI (Java Native Interface)
* **Windows Phone** – C# е основният език за разработване на Windows Phone и това се получава естествено в Xamarin.

**Достъп до SDK на платформите**

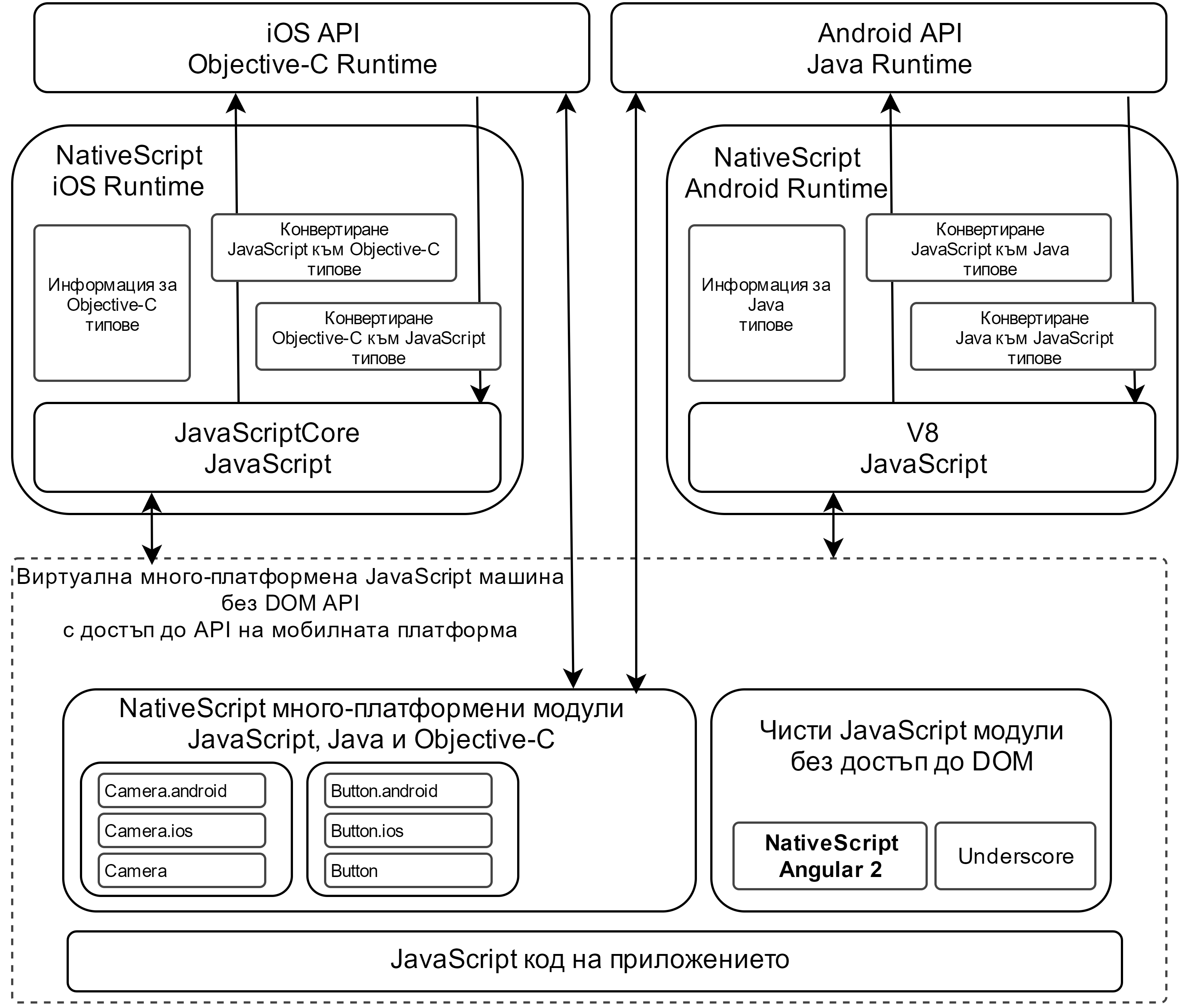
Xamarin позволява достъп до платформено специфичните API чрез C# синтаксис [17]:

* **iOS** – Xamarin.iOS предоставя достъп до CocoaTouch SDK на Apple като пространство от имена които може да достъпвате от C#. Например UIKit на Apple, който съдържа всички потребителски интерфейс компоненти може да бъде включен чрез достъп до MonoTouch.UIKit.
* **Android** – Xamarin.Android позволява до Android SDK като пространство от имена.

## NativeScript

Телерик отваря за публичен достъп NativeScript, платформа за създаване на много-платформени приложения за Android и iOS.

NativeScript не е нов JavaScript диалект, нито обложка около вградената в мобилните платформи функционалност. Вместо това включва JavaScript виртуална машина и механизъм за превеждане на JavaScript извиквания от и към методи на съответната платформа. Приложенията са написани на JavaScript (или TypeScript, CoffeeScript, и т. н.) и се интерпретират от V8 за Android и от WebKit JavaScriptCore за iOS[18].



Фигура 8 Архитектура на NativeScript

NativeScript позволява достъп до предоставените от платформата API, като автоматично конвертира JavaScript типовете към тези на платформата, когато се извикват методи на API и обратно типовете от платформата към JavaScript, когато методите връщат резултат.

За пример, създаване на обект файл в Android:

var file = new java.ios.File(path);

Изпълняват се следните стъпки:

* JavaScript кодът се интерпретира от V8
* Данни за типовете определят извикването към съответния метод на мобилната платформа. Тези данни за типове се компилират предварително от API на платформата, като съдържат необходимата информация за методите и типовете на аргументите.
* Модул за конвертиране на типовете конвертира JavaScript низ към java.lang.String обект.
* JNI извикване създава инстанция на java.io.File използвайки конструктора с един аргумент от тип java.lang.String.
* Създава се JavaScript обект, на чиито функции извикванията се делегират към Java обекта.

Процесът на конвертиране забавя с около 10% (според NativeScript) изпълнението на програмата, сравнено със подобна написана на съответния за платформата език (Java за Android, Objective-C или Swift за iOS).

В резултат приложенията написани на NativeScript могат да използват всички API на платформата от JavaScript код, без да е необходимо от ръчно написан код, на Java или Objective-C, който да извършва прехода. Компонентите на потребителският интерфейс са вградените в платформата компоненти, събитията в интерфейса се обработват чрез класове като View.OnClickListener за Android и UIControl.addTarget за iOS, декларирани в JavaScript.

За да се облекчи разработването на приложения и нужните познания върху различните платформи, NativeScript добавя абстракции върху API-тата на Android и iOS, като позволява голяма част от кода, както и от дефинициите на потребителския интерфейс да е споделена, абстракциите включват: Приложение, Камера, Цвят, Конзола, Дата, Файлова Система, HTTP, Изображения, Настройки, Местоположение, Потребителски Интерфейс и т.н.[18]

Някой от другите качества на NativeScript:

* Компилиране за iOS може да се извърши отдалечено без да е нужен Mac или iOS инструменти.
* Разработването не е обвързано с конкретно IDE.
* Поддържа JavaScript библиотеки, които не са обвързани с браузър или node.js API.
* CSS за контрол върху изгледа на компонентите на потребителския интерфейс.

**Angular 2**

NativeScript има абстракция върху модулите за потребителски интерфейс и HTTP, които позволяват използването на платформата Angular 2. Angular, построен от Google, е широко разпространен сред уеб разработчиците.

## ReactNative

React Native е JavaScript платформа за разработване на приложения, които използват вградените в iOS и Android компоненти. Базиран е на React, JavaScript библиотеката на Facebook за изграждане на потребителски интерфейс, но е насочен към мобилни платформи.

React е JavaScript библиотека за разработване на потребителски интерфейси, обикновено в уеб. Построен с отворен код през 2013 от Facebook, React придобива голяма популярност. React е сравнително тясно специализиран, насочен е единствено към изграждането на потребителският интерфейс на приложението, обратно на големите платформи в стил MVC.[19]

React Native е просто React но за мобилни приложения. Има малки разлики, като например в елементите на потребителския интерфейс, които се използват - <View> вместо <div>, и <Image> вместо <img>. Опитът на разработчиците остава същия, като основни познания по Objective-C и Java са от ползва.

В React, компонентът описва собствения си външен вид, като след това React се справя със самото рендeриране. Чиста абстракция разделя двете функции. За да се рендерира компонент за уеб, React използва стандартни HTML тагове. Същия слой на абстракцията, известен като „мост“ позволява на React Native да използва за рендериране същинските за мобилната платформа API-та - Фигура 9.



Фигура 9 Сравнение на ReactNative и ReactJS [19]

Приложенията се пишат чрез стандартен JavaScript, CSS и HTML. Вместо да бъдат компилирани до естествения за платформата код, React Native изпълнява JavaScript в съответната за платформата JavaScript виртуална машина.

JavaScript кодът се зарежда и изпълнява, като се зарежда и интерпретира от съответната JavaScript машина. Първоначално изпълнението регистрира JavaScript модулите и изпълнението започва от файлът входната точка.

Ето как изглежда извикването на код, чрез „мост“ към функционалност от мобилната платформа:



Фигура 10 ReactNative – мост между Objective-C и JavaScript [20]

Извикванията започват от Java или Objective-C (Фигура 10), към JavaScript. Извикванията се нареждат в опашка. Когато JavaScript нишката се освободи, започва обработка на следващото в опашката извикване. Когато бъде обработено, връща резултати обратно към Java и Objective-C. [20]

## Други

* RoboVM – Решение подобно на Xamarin, базирано на Java, с прекратено развитие
* Kony – JavaScript базирано решение, няма безплатна версия
* Appcelerator – Решение подобно на React Native и NativeScript
* Marmalade – C++ базирано решение, предимно за игри
* Microsot – VisualStudio инструменти за разработка с Cordova и C++, след покупката на Xamarin – C#
* Haxe – много-платформен език за програмиране

# ГЛАВА ТРЕТА

# Функционално проектиране

Решението за разработването на мобилното приложение се води от инструментите в глава първа, и спрямо целите на екипът на NativeScript, за който приложението ще бъде разработено. Случаите му на употреба ще целят да допълнят услугите от вече използваните инструменти:

**GitHub**

Използва се поради:

* Всички данни се запазват в GitHub, не изисква допълнителна поддръжка на сървърна част. Мобилното приложени може да се интегрира с HuBoard, ако спазва същите конвенции и чете форматът на метаданни на HuBoard от GitHub.
* Интуитивен интерфейс за настолен компютър
* Поддържа различни 2 нива на достъп до публични или частни хранилища

Липсват:

* Версия за мобилни устройства на дъската със задачи
* Не поддържа приблизителни оценки на задачите

**TeamPulse**

TeamPulse се използва от екипа в предишни проекти при работа с TFS по проекти с частен код. Използването му е преустановено след преминаване към GitHub.

Използва се поради:

* Добра система за приблизителна оценка на задачите
* Добре развита система за поддържане на приоритетно подреден списък с изисквания
* Табла за прогрес на задачи и оставаща работа, визуализация

Липсват:

* Няма версия за мобилни устройства
* Изисква синхронизация между две отделни бази данни
* Разработен върху TFS и в последствие интегриран с GitHub

TeamPulse е разработен от същата фирма, към която принадлежи екипът, но използването му е преустановено с аргументите, че изисква допълнителна поддръжка за интеграция с GitHub. Мобилното приложение аналогично няма да бъде използвано, ако не разчита директно на GitHub, за сървърното си осигуряване.

**Aha!**

Използва се поради:

* Мощна система за визуализация
* Споделяне на план за развитие на продукта

Липсват:

* Високото ниво на планиране в Aha! не позволяват всички по-краткосрочни задачи да бъдат описани

## Случаи на употреба при гъвкави методологии

Мобилното приложение ще се използва по желание от разработчиците в екип, работещ с гъвкави методологии и по проект с отворен код в GitHub. Методологията много се доближава до Скръм, но се пропускат стриктните срещи за планиране и ретроспекция. В екипа няма и ограничение за брой започнати задачи, което го различава и от Канбан.

В екипа са ясно обособени следните роли: Собственик на продукт, разработчици, качествен контрол. Като допълнително потребителите на проекта могат да играят роля на съразработчици и качествен контрол. От това произтичащите взаимоотношения са описани във Фигура 11.

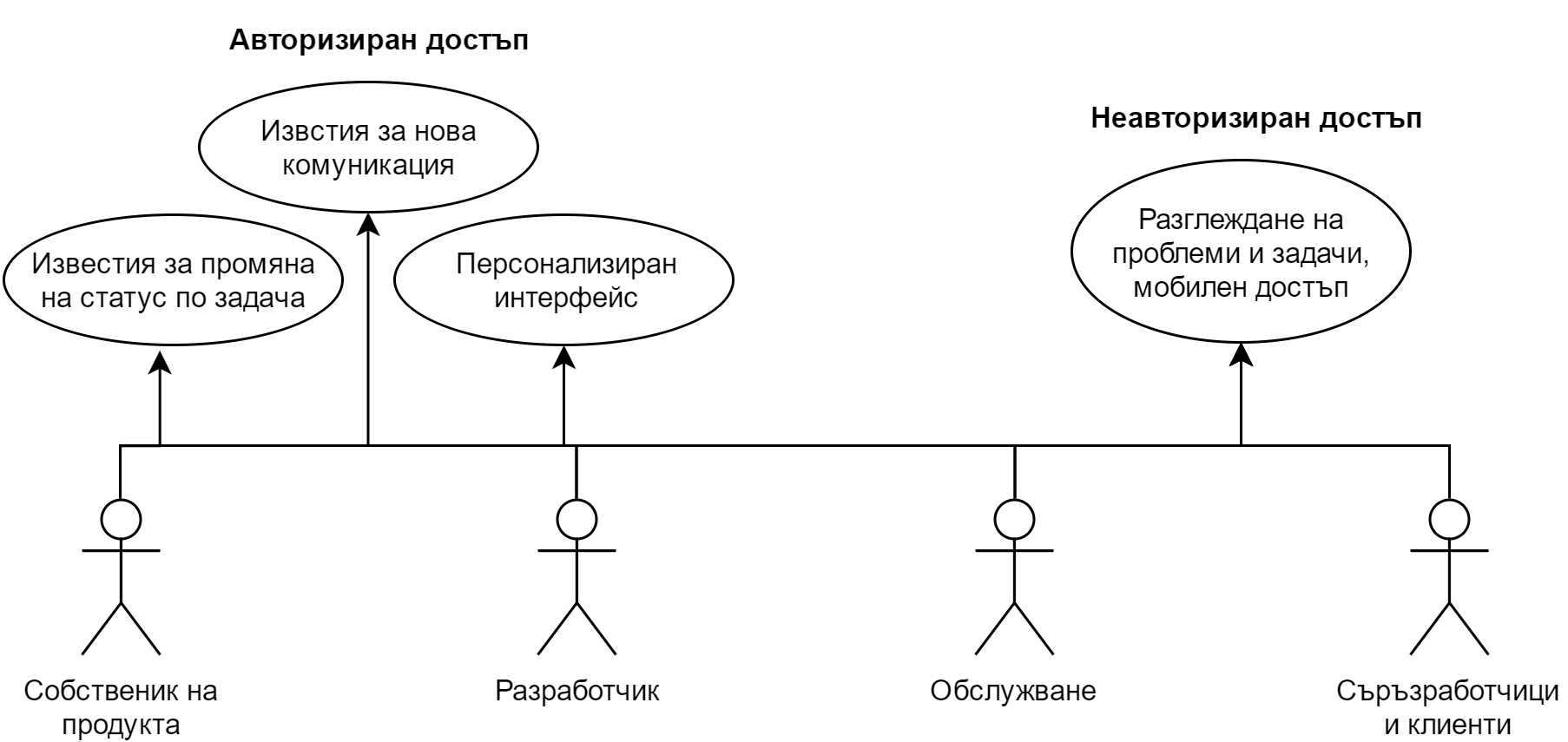


Фигура 11 Случаи на употреба на гъвкави методологии в екипа

Мобилното приложение може да допринесе полза предимно в случаите на изчакване на съобщения, промени на статус и чрез персонализиран потребителски интерфейс.

## Случаи на употреба на мобилното приложение

При работа с отворен код информацията за задачи, прогрес, статус, код ревюта и др. са публично достояние. За това приложението ще позволява достъп и използване от нерегистрирани потребители. То не може да замени цялата гама от прилагани инструменти в Скръм и Канбан, случаите на употребата му са описани във Фигура 12.



Фигура 12 Случаи на употреба на мобилното приложение.

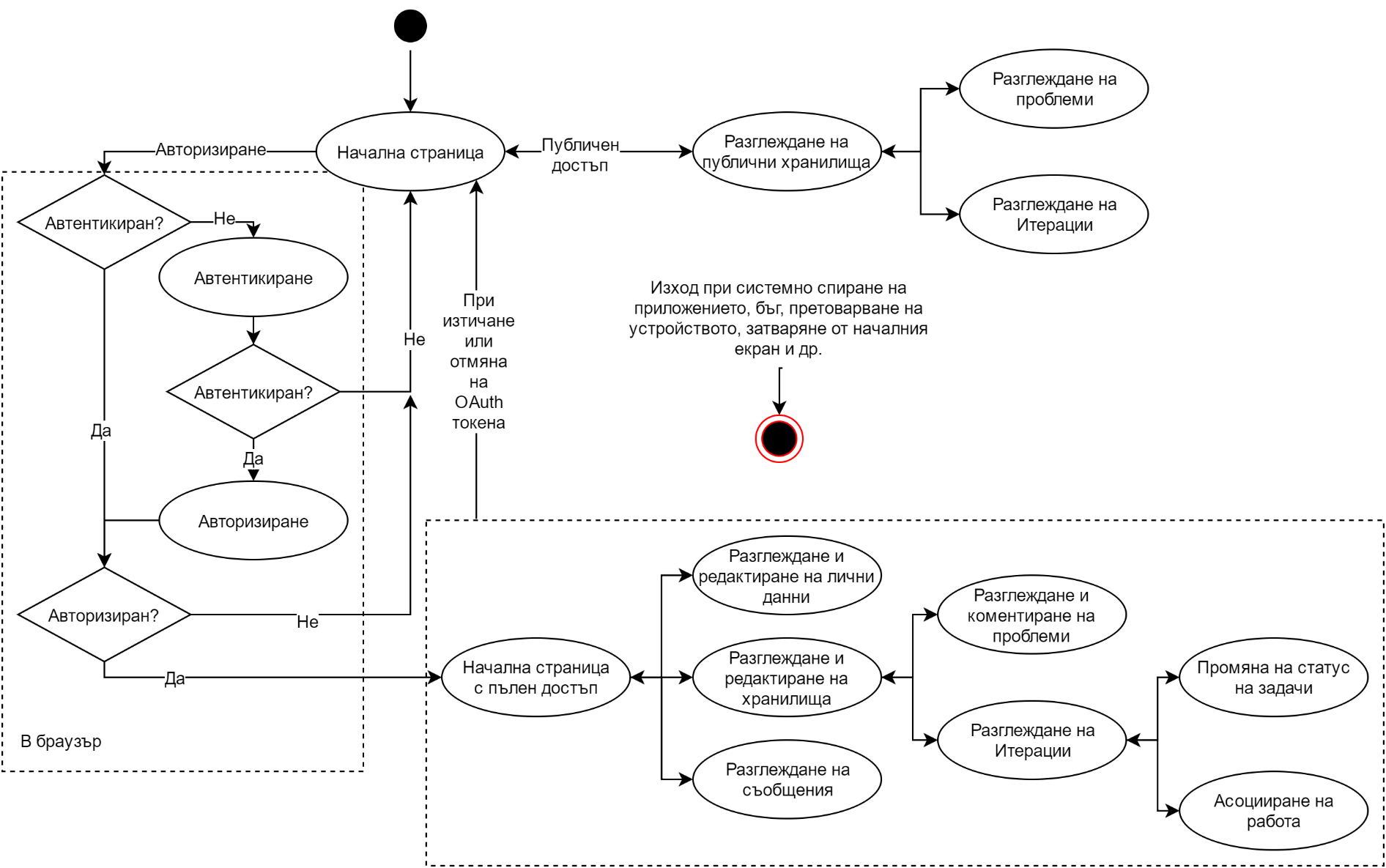
## Дейности извършвани чрез приложението

Дейностите, които потребителите на приложението са описани на Фигура 13. Разделят се на три по-големи групи.

* Дейности позволени върху данни с публичен достъп
* Автентикация
* Дейности позволени върху данни нуждаещи се от допълнителни права (автентикация)

Автентикацията се извършва във браузър. По този начин има голяма вероятност потребителя вече да е посещавал мобилната страница на GitHub и да няма нужда повторно да въвежда име и парола.

Веднъж авторизирано, GitHub съхранява информация за него и при повторно поискване за автентикиране пренасочва потребителя обратно към приложението, без да е нужно потвърждение от потребителя.



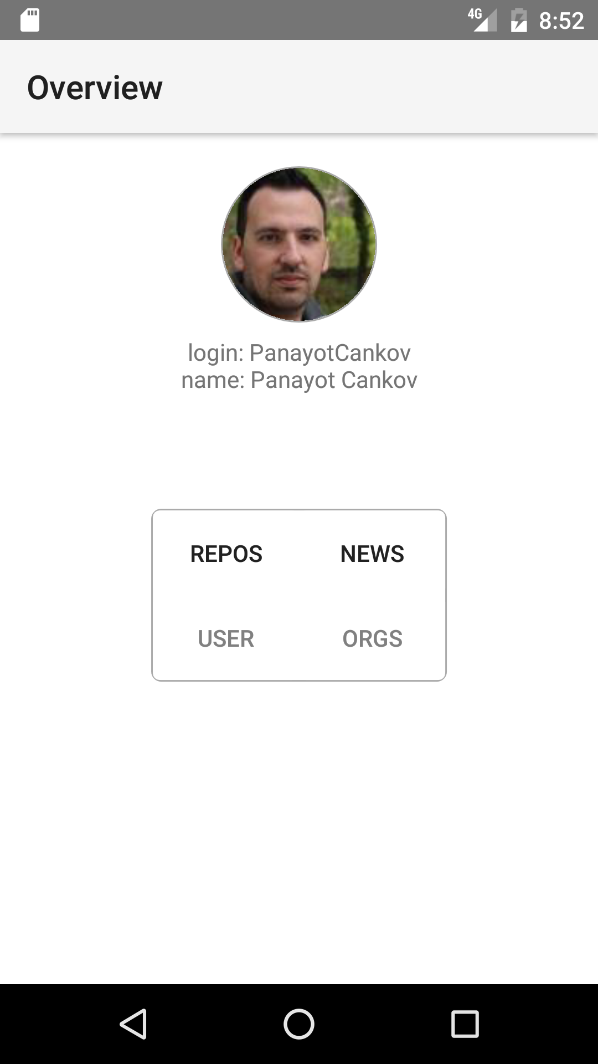
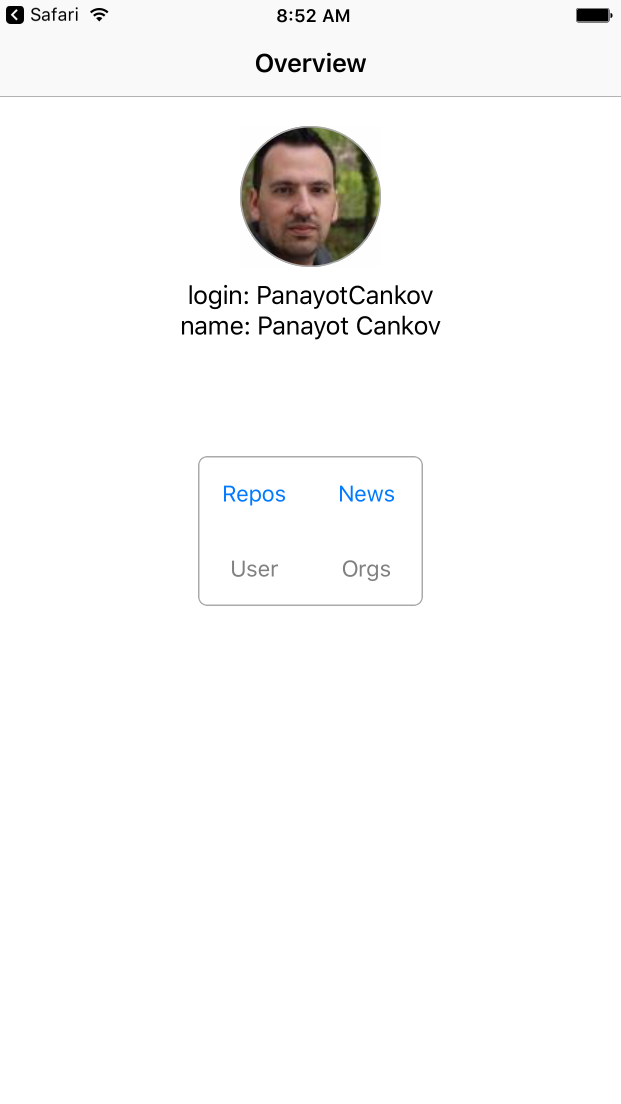
Фигура 13 Диаграма на дейностите на мобилното приложение.

## Потребителски интерфейс

Приложението трябва да е достъпно за iOS и Android потребители поради хетерогенното разпределение на личните мобилни устройства на хората в екипа. Използваните технологии описани по-късно в глава четвърта предоставят тази възможност.

**Начална страница**

Началната страница позволява инициирането на авторизиране при натискане на полето за потребителски аватар в горния край на екрана. Визуално след авторизиране се променя само с това, че се зарежда аватара на потребителя, който замества бутона за иницииране на автентикирането. Изглед от потребителския интерфейс на началната страница на Фигура 14.



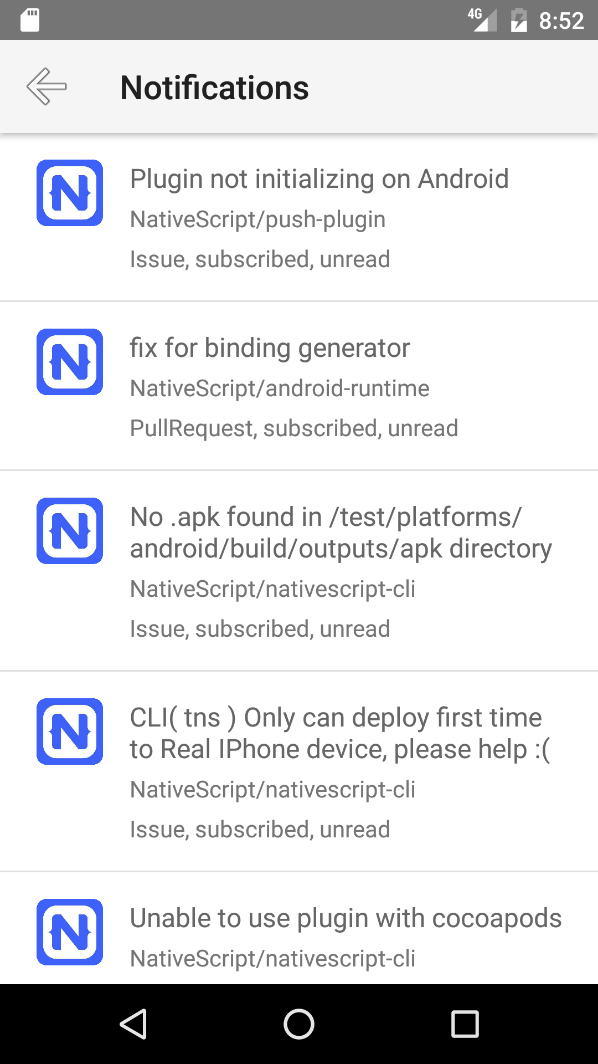
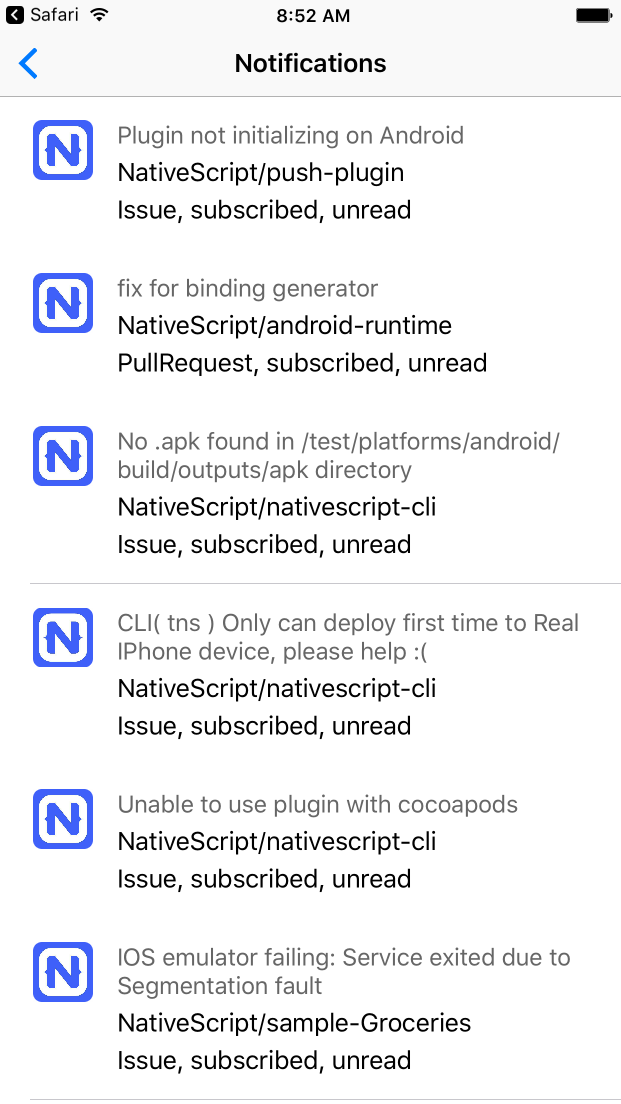
Фигура 14 Потребителски интерфейс на началната страница

**Информация за потребителя**

Личните данни на потребителя, като аватар, име, ел. поща и др. от GitHub се появяват при натискане на аватара на потребителя, след авторизиране. Страницата прилича на началната страница, но бутоните за навигация се заменят с текстви полета със съответната информация.

**Съобщения**

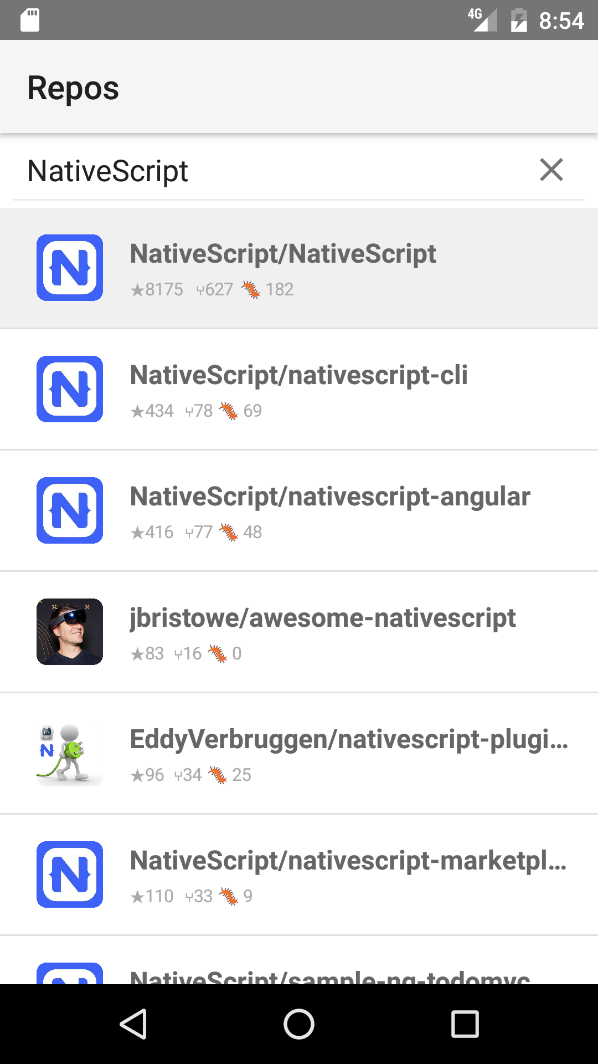
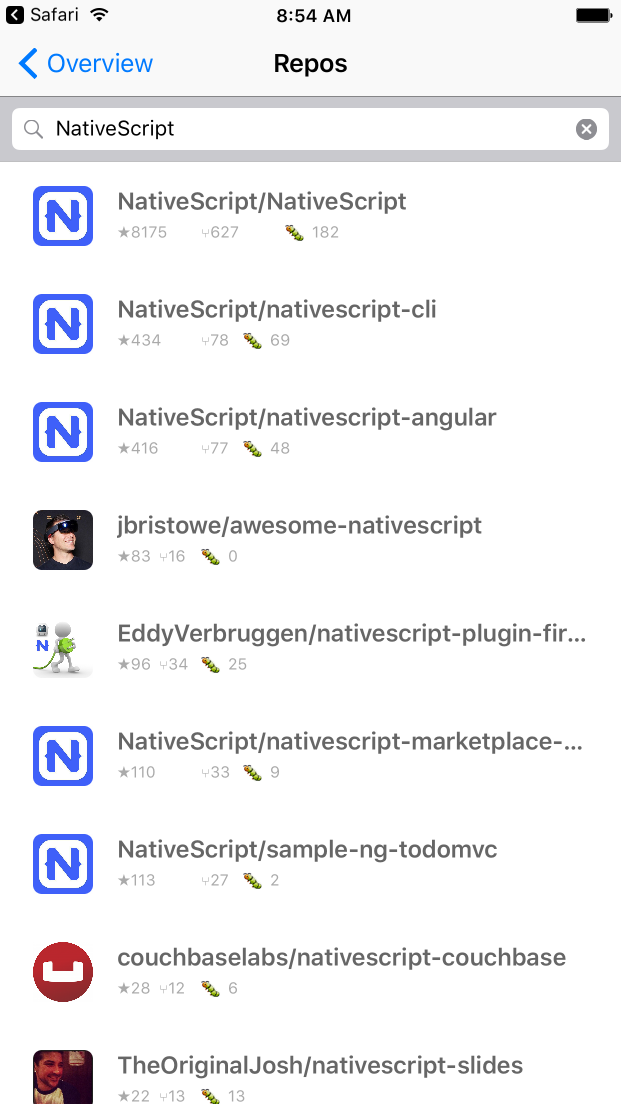
GitHub поддържа API за работа със съобщения. След авторизация, от началната страница потребителя може да премине към страница със съобщения. Проблеми и теми в които потребителя участва активно, се следят за промени и се отразяват в съобщенията на потребителя - Фигура 15. При избор на съобщение може да бъде прочетена комуникацията по нея.



Фигура 15 Потребителски интерфейс за GitHub съобщения.

**Хранилища**

От началната страница потребителя може да премине към страница с хранилища. В зависимост от това дали е автентикирал приложението за достъп до личните му данни, ще се покажат хранилищата в които активно участва. Ако не е, остава функционалността да търси и показва информация за публични хранилища - Фигура 16.

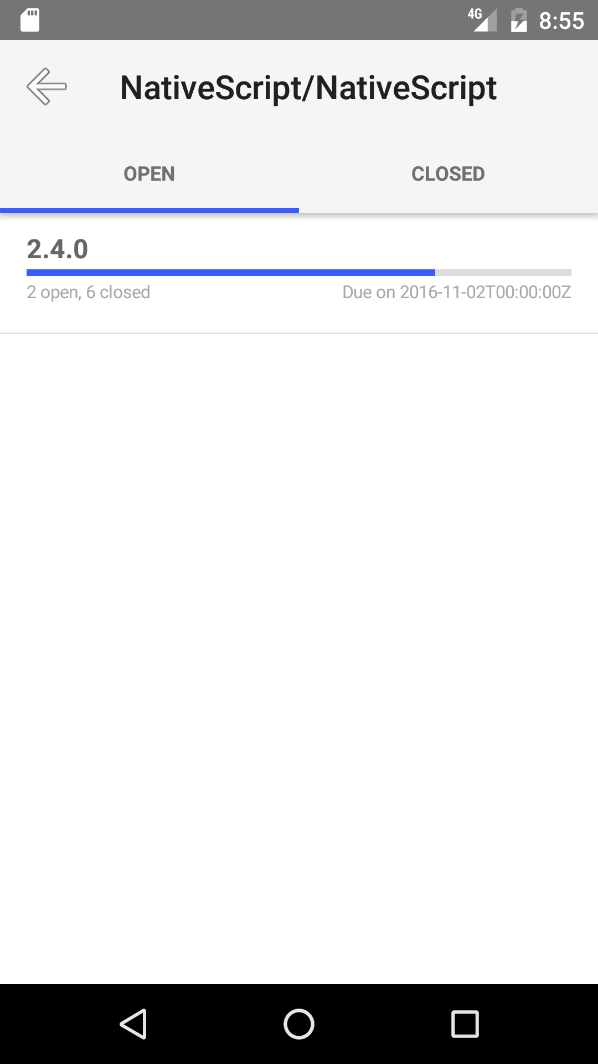


Фигура 16 Потребителски интерфейс показващ хранилища

При избор на хранилище приложението навигира към екран със спринтовете му.

**Спринтове**

За маркиране на Скръм спринтове се използват GitHub Milestones, те позволяват да бъдат асоциирани със срокове и задачи. Потребителският интерфейс разделя спринтовете в две категории – отворени и затворени. При продължителна работа се натрупват голям брой неактивни затворени спринтове. По подразбиране се представят отворените, като броя на оставащи и изпълнени задачи се изобразява визуално с компонент за прогрес - Фигура 17.

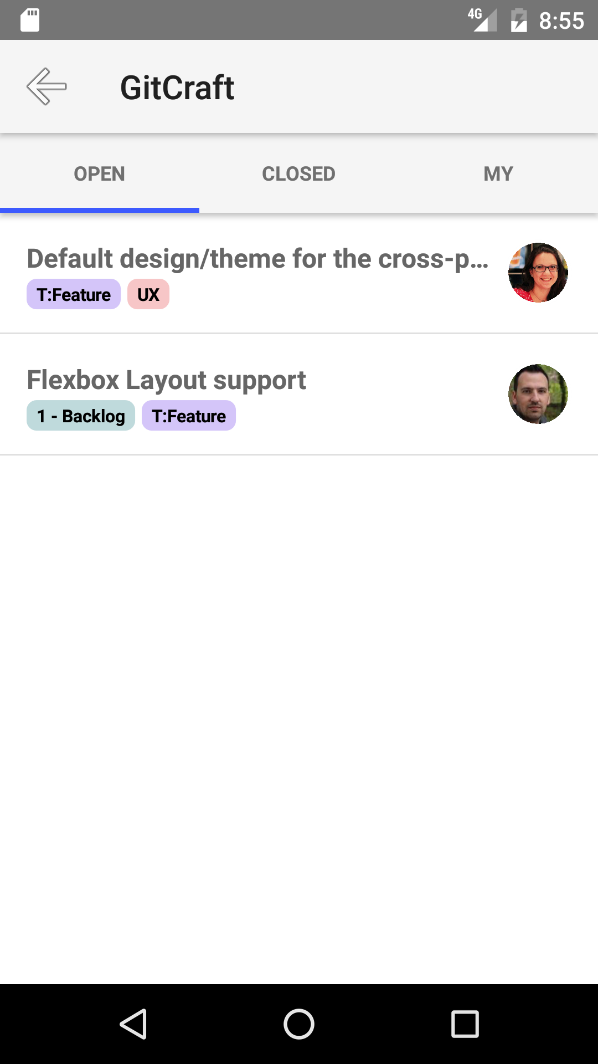
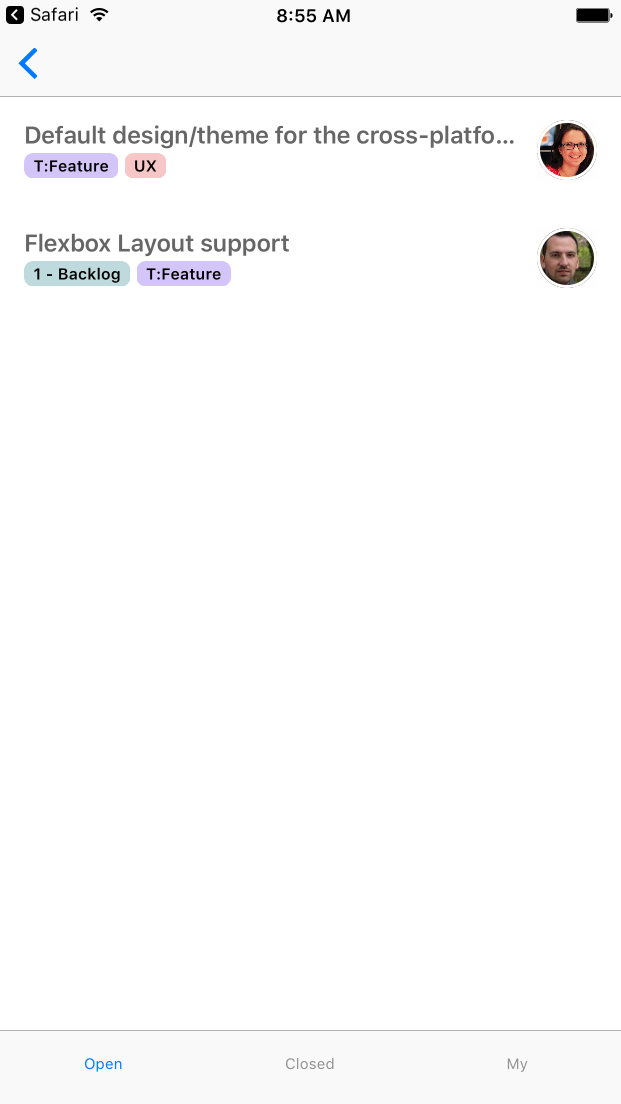


Фигура 17 Потребителски интерфейс за спринтове на хранилище

При избор на спринт, се показва табло със задачите от съответният спринт.

**Задачи по спринтове**

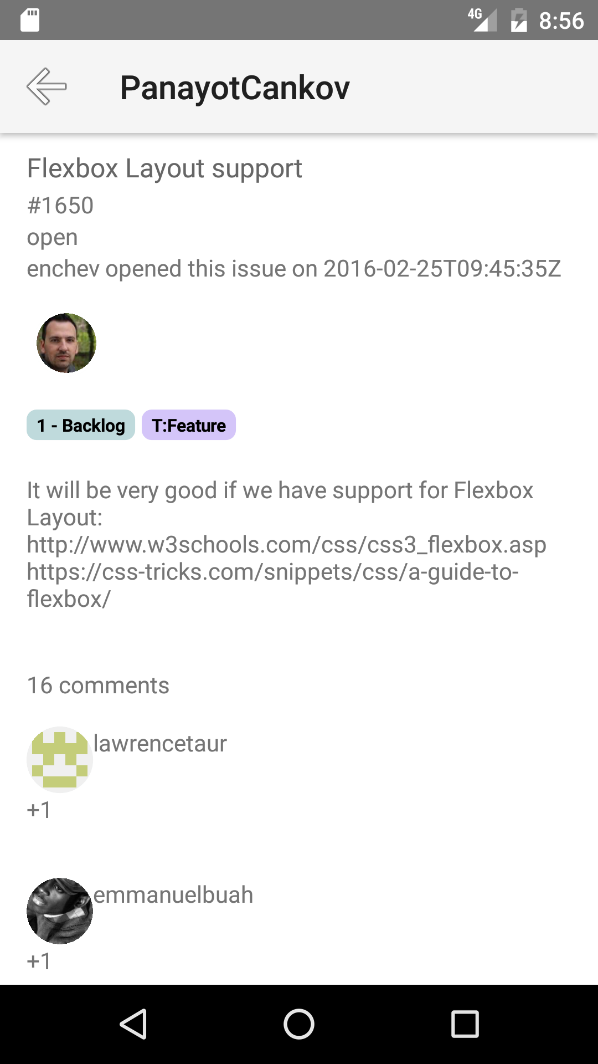
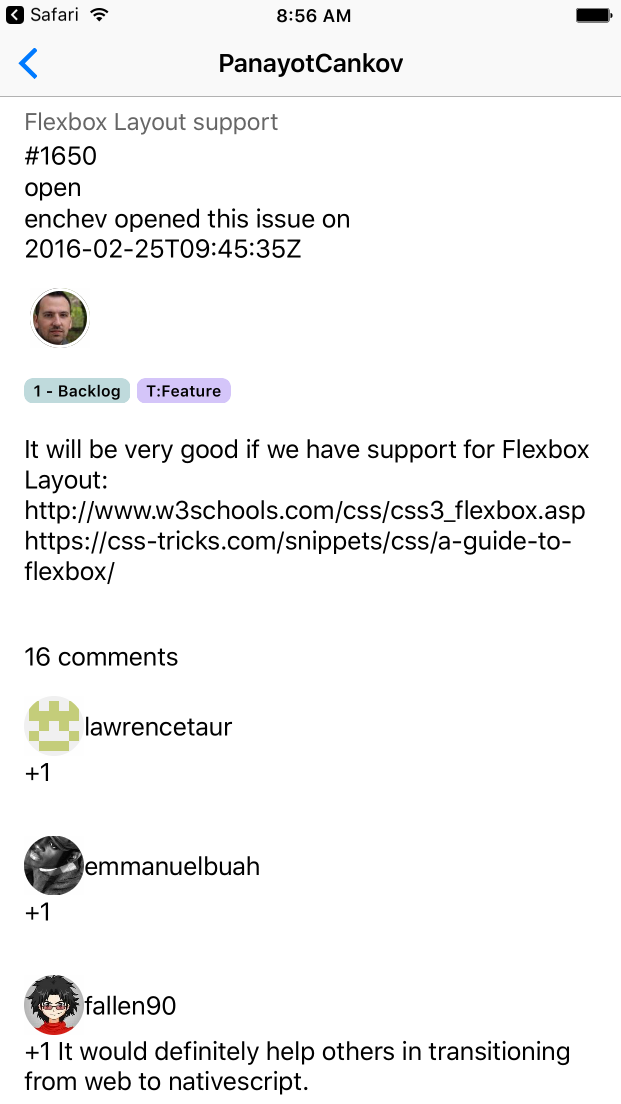
Потребителският интерфейс изобразяващ задачите за спринт ги групират за да може изгледа да доближава Скръм и Канбан дъските. Предоставя се информация за етикети асоциирани със задачите, разработчици асоциирани със задачите, както и заглавие - Фигура 18. При натискане на задача се навигира към страница с детайли, където може да се проследи и евентуална комуникация с клиенти и съразработчици.



Фигура 18 Потребителски интерфейс показващ задачи по спринт.

**Задачи**

Към страницата със задачи може да довете задача от спринт, или съобщение. Потребителският интерфейс показва информация за това, кой е създал задачата, асоциирани етикети, детайлно описание, както и коментари от потребители и съразработчици - Фигура 19.



Фигура 19 Потребителски интерфейс GitHub задачи

# ГЛАВА ЧЕТВЪРТА

# Технологично проектиране на мобилното приложение

В тази глава ще разгледаме проектирането и разработката на мобилно приложение, което да подпомага работата в GitHub, прилагането на гъвкави методологии и комуникацията по проекти с отворен код. Избор на инструменти за разработка

За разработката на приложението ще използваме NativeScript, Angular 2 и TypeScript. Изборът е обоснован предимно от възможността екипът на NativeScript да тества собствения си продукт и да получи обратна връзка.

Технологията ще задоволи нуждата от интеграция с GitHub, която се случва през HTTPS и протича изцяло чрез JSON формат, както и изграждането на богат потребителски интерфейс демонстриран в глава трета. Подобно приложение със сходен успех би могло да се разработи и с останалите платформи разгледани в глава втора.

## Компоненти на системата

Мобилното приложение представлява потребителски интерфейс към услугите предоставени от GitHub. Аналогично на приложението, инструментите за работа със гъвкави методологии в GitHub се интегрират с него и съответно са достъпни.

Системите използвани активно от екипът включват:

* **Aha!** – Планираното на високо ниво, поддържа интеграция с GitHub и предоставя възможност задачите в Aha! да бъдат асоциирани с GitHub задачи. Съответно при изпъленение на задача в GitHub това се отразява в Aha!
* **HuBoard** – Както споменахме в глава първа, използва се като електронна Скръм или Канбан дъска. Промените по дъската се отразяват директно
* **CI Travis** – Облачна платформа за продължителна интеграция, позволява изпълнение на тестове срещу симулатори и емулатор.
* **CI Jenkins** – Сървър обслужващ продължителна интеграция, вътрешно в тима. Позволява тестове да бъдат изпълнявани срещу физически мобилни устройства.
* **Електронни таблa –** Екипът на NativeScript разполага с няколко теливизионни табла, които показват на живо информация от продължителната интеграция.
* **GitHub CLI** – Не може да не споменем и използването на командния интерфейс на GitHub, тъй като чрез него е възможно да се променя статуса за задачи.
* **SourceTree** – Приложение с визуален потребителски интерфейс за работа с Git.

Всички тези системи се интегрират чрез публичните интерфейси на GitHub. Мобилното приложение ще има максимален достъп до данни от тях ако и то използва публичното API на GitHub - Фигура 20.



Фигура 20 GitHub интеграция на сървъри и услуги в екипа на NativeScript

## Реализация

GitHub API то е имплементирано чрез Angular 2 service, който предоставя възможност за изпращане на заявки към GitHub през Http service на Angular 2.

Макар и динамичен език за програмиране JavaScript подкрепен с TypeScript притежава мощна система за типизация и статична проверка на типовете по време на компилация.

GitHub заявките минават през “request” метод. Описан е с няколко TypeScript overload-а, по един за всеки адрес от GitHub API-то.

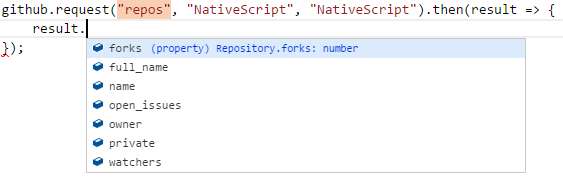
Пример:

request(repos: "repos", owner: string, repo: string): Promise<Repository>;

При използването му:

github.request("repos", "NativeScript", "nativescript-cli").then(result => {});

TypeScript компилатора ще успее определи типа на “result” като “Repository” и съответно ще позволи само достъпът до полета съществуващи в „Repository” интерфейсът. По време на работа в интегрираната среда (изпозлваме VSCode, но WebStorm, Atom, Sublime и т. н. също имат интеграция за TypeScript) подава автоматични подсказки въз основа на типовете:



Макар и да няма толкова мощни механизми за рефакториранe като инструментите за рaбота със силно типизирани езици като C#, Objective-C и Java, статичната проверка на типовете позволяват на приложение написано на JavaScript да скалира лесно и да е удобно за поддръжка.

Като пример, ако GitHub API претърпи промяна, достатъчно е тази промяна да се отрази в съответния интерфейс и местата, където се използва неправилно, ще бъдат открити от статичния анализ на типовете.

Да разгледаме основната част от класът GitHubService:

import { Injectable, NgZone } from '@angular/core';

@Injectable()

export class GitHubService {

private static access\_token: string;

// …

constructor(private http: Http, public zone: NgZone) {

console.log("New GitHub instance!");

}

public request(... args: (string | {})[]): any {

let querryUri = "https://api.github.com/";

querryUri += args.filter(s => typeof s === "string").join("/");

let last = args[args.length - 1];

let params = typeof last === "object" ? last : undefined;

if (params || GitHubService.access\_token) {

querryUri += "?";

let separate = false;

for(let key in params) {

// TODO: Url escape

querryUri += (separate ? "&" : "") + key + "=" + params[key];

separate = true;

}

if (GitHubService.access\_token) {

querryUri += (separate ? "&" : "") + "access\_token" + "=" + GitHubService.access\_token;

separate = true;

}

}

console.log("Querry: " + querryUri);

return this.http.get(querryUri, {

// Enables experimental reactions.

headers: new Headers({ Accept: "application/vnd.github.squirrel-girl-preview" })

})

.toPromise()

.then(response => Promise.resolve(response.json()));

}

// …

}

Класът е маркиран е с Injectable декоратор, което добавя информация за типовете на конструктора му и позволява на Angular2 DI (Dependency Injection – инжектиране на зависимости) да създаде нова инстанция на този клас услуга когато е нужно. Зависимостите на Http и NgZone ще бъдат удовлетворени автоматично от DI, което улеснява създаването на обектния граф при инициализирането на приложението.

В JavaScript, както и в TypeScript, overload на метод е възможно ако подадените типове се разгледат с интроспекция. Методът request ще играе основна роля в нашето приложение, като той ще се използва за достъп до всички API на GitHub. Имплементиран е като приема произволен брой аргументи, първите се конкатенират в път за заявката. Ако последният обект не е стринг, се използва за параметри на заявката. По-късно ще разгледаме един по един типизираните варианти на този метод.

Не на последно място се добавят и HTTP хедъри, които да позволят някой по-нови (нестабилни) API-та, както и access\_token параметър. За разлика от параметрите на API тата, които се подават като аргумент на метода, access\_token е избран да е общ за всички заявки и се подава винаги. По-късно ще разгледаме и как се генерира и използва.

## OAuth

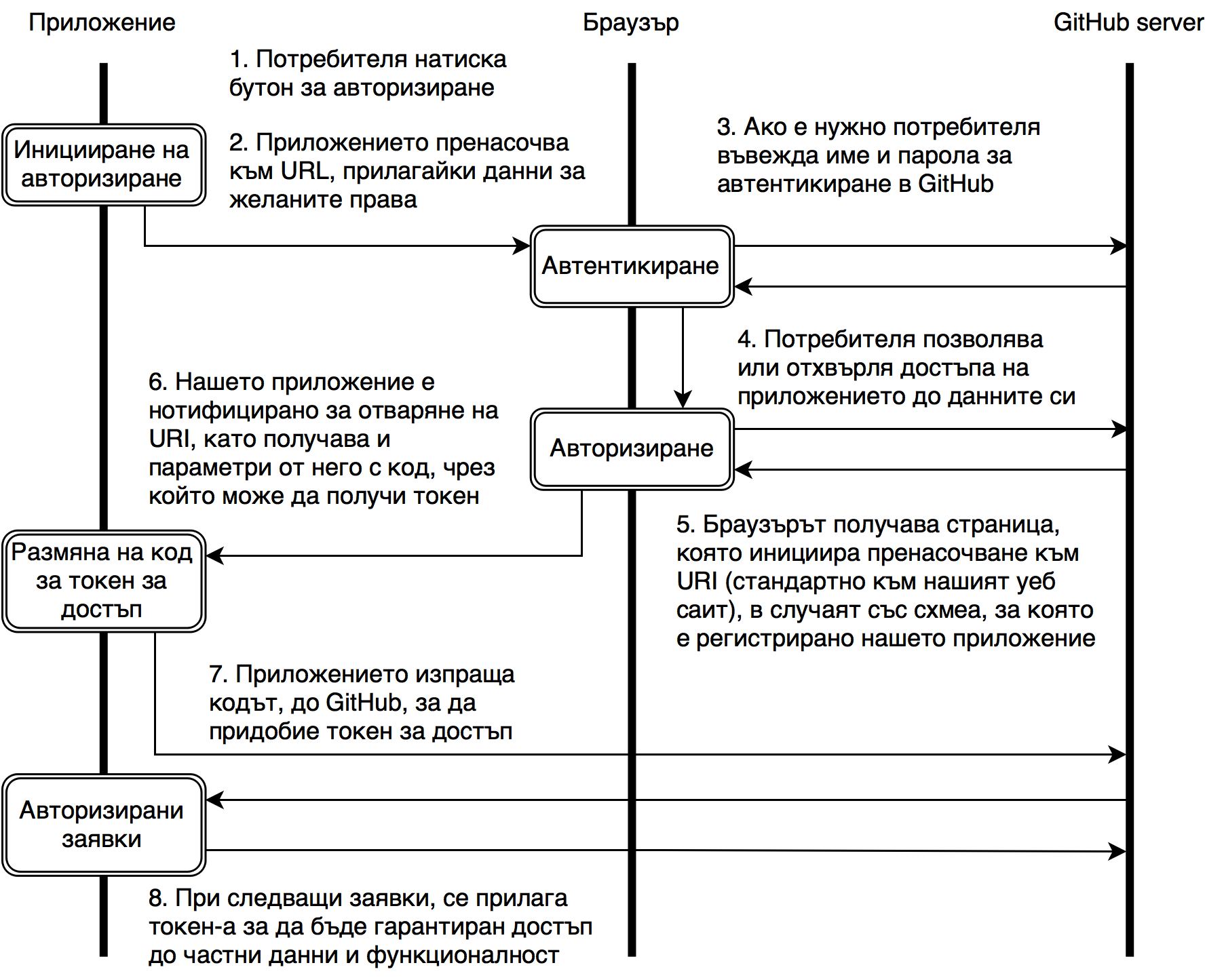
Приложението ни е съобразено със стандартите за добър потребителски опит и позволява да бъде използвано без потребителя да го авторизира с достъп до данните и правата му в GitHub. Съответно при стартиране на приложението е възможно разглеждането на публични хранилища в GitHub, като анонимен потребител, но не е възможно да се правят промени, коментари или да се разглеждат частни хранилища.

Авторизирането на приложението става с OAuth 2 и гарантиране на код за авторизиране (authorization code grant) поддържан от GitHub.

## Проектиране и разработка на модул за достъп OAuth автентикация

Когато приложението покаже страница във WebView има достъп до съдържанието, което се показва и би могло да прихване потребителското име и парола, докато потребителя се автентикира. По този начин може да придобие достъп до всички услуги достъпни за потребителя, дори заявката за авторизация да е включвала само част от услугите. Например, нашето приложение може да поиска достъп за четене на хранилища, но с потребителското име и парола използвайки основна автентикация (basic authentication) да изтрие всички организации в които потребителя е администратор.

Android и iOS позволяват на приложенията да бъдат регистрирани като подходящи за отваряне на URL със специална схема, да поска страницата на GitHub за OAuth да бъде отворена във стандартния за платформата браузър и след автентикиране и авторизиране, браузърът да върне код за достъп в нашето приложение.



Следвайки този механизъм не е нужно потребителите да вярват напълно на приложението, за да го използват.

### Иницииране

Кодът за OAuth е част от нашият GitHubService. При искане за авторизация от потребителският интерфейс се извиква метода requestOAuth():

import \* as utils from "utils/utils";

public requestOAuth() {.

utils.openUrl(GitHubService.OAuthRequestURL);

}

Пакетът utils е част от модулите на NativeScript, който предоставя методи обобщаващи някой често използвани функционалности на двете мобилни платформи Android и iOS, от него се извиква openUrl. И двете операционни системи предоставят възможност за отваряне на URL като в зависимост от схемата се стартират различни приложения. В този случай адресът <https://github.com/login/oauth/authorize> ще бъде отворен в браузър.

Следващите стъпки се извършват от потребителя в браузър, като се автентикира и авторизира приложението ни. При потенциален отказ или връщане в приложението ни по друг начин (например хардуерният бутон в Android назад) приложението ни остава напълно работещо, като се показват само публични данни, възможно е и повторно иницииране на процеса. Какво става ако все пак потребителя даде достъп на приложението ни необходимите права? Браузърът ще пренасочи към URL: gitcraft://oauth.

### Обработка при успех

За да регистрираме приложението за схемата gitcraft:// е нужно да обновим AndroidManifest.xml за Android и Info.plist за iOS. Тази функционалност в двете платформи Android и iOS се различава и се извършва декларативно а не чрез императивен код. Това я прави неудобна за уеднаквяване чрез общ модул.

Добавяме в AndroidManifest.xml на Android приложението:

<activity

android:name="com.tns.NativeScriptActivity"

android:label="@string/title\_activity\_kimera"

android:configChanges="keyboardHidden|orientation|screenSize">

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />

</intent-filter>

<intent-filter>

<data android:scheme="gitcraft" />

<action android:name="android.intent.action.VIEW" />

<category android:name="android.intent.category.LAUNCHER" />

<category android:name="android.intent.category.DEFAULT" />

<category android:name="android.intent.category.BROWSABLE" />

</intent-filter>

</activity>

По този начин главното Android активити ще се използва и за стартиране на приложението от началният екран и за получаване на Intent с данни при улавяне на URL със gitcraft схема.

При iOS това става както споменахме в Info.plist:

<key>CFBundleURLTypes</key>

<array>

<dict>

<key>CFBundleURLName</key>

<string>org.nativescript.GitCraft</string>

<key>CFBundleURLSchemes</key>

<array>

<string>gitcraft</string>

</array>

</dict>

</array>

Следваща стъпка е да обработим съобщенията. В main.ts, стартовият скрипт на приложението:

import \* as application from "application";

import { isAndroid, isIOS } from "platform";

if (isIOS) {

class AppDelegate extends NSObject implements UIApplicationDelegate {

static ObjCProtocols = [UIApplicationDelegate];

applicationHandleOpenURL(app, url): boolean {

return GitHubService.applicationHandleOpenURL(app, url);

}

}

application.ios.delegate = AppDelegate;

} else if (isAndroid) {

GitHubService.registerForURLIntent();

}

По този начин създаваме динамично клас AppDelegate за наследник на NSObject и имплементираме Objective-C протокол UIApplicationDelegate, директно в JavaScript. Статичното поле ObjCProtocols прави впечатление. При компилиране от TypeScript към JavaScrip се прави статичен анализ на типовете, но след това информацията за типовете, включително и това, че класът AppDelegate имплементира UIApplicationDelegate се губи. NativeScript за iOS използва няколко резервирани статични полета.

Така създаденият клас се подава на NativeScript в модула application. За съжаление това е особеност на iOS и приложението не позволява добавянето на наблюдатели след стартиране. За Android това е възможно и регистрирането става в GitHubService.

Методът registerForURLIntent приложението се абонира за съобщение, че е било събудено и ще провери дали това не е в отговор на нашата схема:

public static registerForURLIntent() {

let handler = args => {

console.log("activityResumed!");

let intent = args.activity.getIntent();

let data = intent.getData();

if (data) {

let scheme = data.getScheme();

let host = data.getHost();

if (scheme === "gitcraft" && host === "oauth-cb") {

let code = data.getQueryParameter("code");

let state = data.getQueryParameter("state");

GitHubService.exchangeForAccessToken({ code, state });

}

}

};

application.android.on("activityResumed", handler);

}

При успех, се извлича код (code) и се извиква методът exchangeForAccessToken. Това се извършва със API на Android. NativeScript няма абстракция върху URL и се налага извличането да се повтори с подобен код за iOS в applicationHandleOpenURL:

public static applicationHandleOpenURL(application, url) {

let urlComponents = NSURLComponents.componentsWithURLResolvingAgainstBaseURL(url, false);

let items = urlComponents.queryItems;

let code: string = null;

let state: string = null;

console.log("Items: " + items);

items.enumerateObjectsUsingBlock(item => {

console.log(" query params: " + item);

if (item.name == "code" && !code) {

code = item.value;

} else if (item.name == "state" && !state) {

state = item.value;

}

});

GitHubService.exchangeForAccessToken({ code, state });

return true;

}

Обработката на кодът получен от GitHub се обработва с API абстракция върху двете платформи и е общ:

private exchangeForAccessToken(params: { code: string, state: string }) {

console.log("exchangeForAccessToken: " + params.code + " " + params.state);

let url = "https://github.com/login/oauth/access\_token";

url += "?client\_id=ddad3314e37c5efbf57f";

url += "&client\_secret=304bc86ebc19fa0de7dcdbf8bb9afdbbad45d639";

url += "&code=" + params.code;

this.http.post(url, "", {

headers: new Headers({ Accept: "application/json" })

}).toPromise().then(result => {

console.log("Result from OAuth: " + result.status);

let resultJson: {

access\_token: string,

token\_type:"bearer",

scope: string,

} = result.json();

console.log("result " + JSON.stringify(resultJson));

this.zone.run(() => {

GitHubService.access\_token = resultJson.access\_token;

this.authorizedChange.emit({});

console.log("in NgZone " + this.zone);

this.requestUser();

// Save the token...

this.persistToken(GitHubService.access\_token);

});

}).catch(error => {

console.log("OAuth error: " + error);

});

}

Полученият код се разменя с http заявка за токен за достъп. Той се запазва и по-късно се прилага към всяка заявка. Това удостоверява пред GitHub че заявките са упълномощени от потребителя и приложението притежава допълнителни права.

## Потребителски интерфейс

Angular 2 организира потребителският интерфейс в компоненти, а компонентите и услугите в модули. В main.ts след регистрацията за URI схемата ни, следва регистрирането на компонентите и пътища за навигация налични в приложението ни.

@NgModule({

declarations: [

BackgroundColorPipe,

ColorPipe,

AppComponent,

HomeComponent,

UserComponent,

RepositoriesComponent,

RepositoryComponent,

MilestoneComponent,

NotificationsComponent,

IssueComponent

],

bootstrap: [AppComponent],

imports: [

NativeScriptModule,

NativeScriptHttpModule,

NativeScriptRouterModule,

NativeScriptRouterModule.forRoot([

{ path: "", component: HomeComponent},

{ path: "user", component: UserComponent},

{ path: "repos", component: RepositoriesComponent },

{ path: "repos/:owner/:name", component: RepositoryComponent },

{ path: "repos/:owner/:name/issues/:issue", component: IssueComponent },

{ path: "milestone/:owner/:name/:milestone", component: MilestoneComponent },

{ path: "notifications", component: NotificationsComponent }

])

]

})

class AppComponentModule { }

Входящата точка е AppComponent, в който се случва основната навигация и се показват всеки един компонент от описаните в NativeScript маршрутизатора.

Компонентите се състоят от код, шаблон (template) и CSS за стилизиране. За добра практика се счита кодът на всеки компонент да служи за връзка между потребителския интерфейс и услугите, които съдържат бизнес логиката.

## Начално табло

Началната страница е екранът, който се показва при стартиране на приложението.

import {Component, ChangeDetectionStrategy} from "@angular/core";

import {Router} from "@angular/router";

import {GitHubService} from "./github.service";

@Component({

selector: "Home",

templateUrl: "home.component.html",

styleUrls: ["home.component.css"]

})

export class HomeComponent {

constructor(public github: GitHubService, public router: Router) {

console.log("new HomeComponent!");

}

}

Компонентът изисква GitHubService при инстанцирането си. Съответно шаблонът за потребителският нитерфейс:

<ActionBar title="Overview">

</ActionBar>

<GridLayout rows="auto, auto, \*">

<GridLayout class="authorization" rows="auto, auto" (tap)="github.authorized ? router.navigate(['user']) : github.requestOAuth()">

<Image row="0" class="avatar" [src]="github.authenticatedUser ? github.authenticatedUser.avatar\_url : null"></Image>

<Label row="1" class="authorize-hint" textWrap="true" textAlignment="center"

[text]="github.authorized ? (github.authenticatedUser ? 'login: ' + github.authenticatedUser.login + '&#xA;name: ' + (github.authenticatedUser.name || '-') + '&#xA; &#xA; ' : ' &#xA; &#xA; ') : 'Public access only.&#xA;Tap to authorize GitCraft&#xA;to access private repositories'"></Label>

</GridLayout>

<GridLayout row="1" class="routes" rows="\*, \*" columns="\*, \*">

<Button row="0" col="0" text="Repos" (tap)="router.navigate(['repos'])"></Button>

<Button row="0" col="1" text="News" (tap)="router.navigate(['notifications'])"></Button>

<Button row="1" col="0" text="User" color="gray"></Button>

<Button row="1" col="1" text="Orgs" color="gray"></Button>

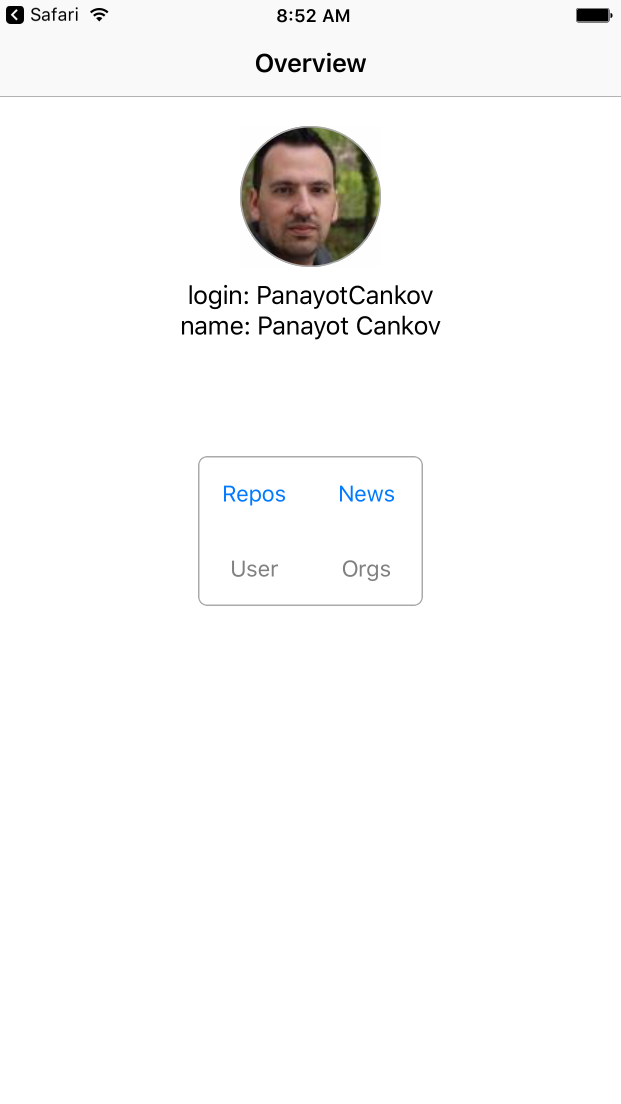
</GridLayout>

</GridLayout>

Компонентите консумират директно GitHubService. При интеракция, вкючително и резултат от HTTP заявка, Angular 2 ще извърши претърсване за промени. В горната част на интерфейсът има картинка с аватара на потребителя ако е авторизирал приложението, или бутон за авторизиране ако не е.

В долната има няколко бутона за навигация към съответните части на приложението.

Ето и как изглежда това на мобилните устройства:



## Новини

Умните телефони са преди всичко средство за комуникация и нашето приложение трябва да се възползва максимално от това. GitHub има API за съобщения и новини. Когато клиенти отварят предложения за подобрения, описват бъгове или добавят коментари, GitHub организира тези събития и ако потребителя е заинтересован да ги следи, могат да бъдат добавени към новините му.

API се консумира чрез заявка към:

* GET /notifications – всички новини за потребител
* GET /repos/:owner/:repo/notifications – всички новини за хранилище

Съответно новините могат и да бъдат маркирани като прочетени чрез:

* PUT /notifications – маркира всички
* PUT /repos/:owner/:repo/notifications – маркира всички в хранилище
* PATCH /notifications/threads/:id – маркира конкретна новина

Абонаментът за новини може да се управлява през:

* GET /notifications/threads/:id/subscription
* PUT /notifications/threads/:id/subscription

При прекомерна потребителска активност по някой наболял проблем, част от разработчиците може да пожелаят да пропуснат комуникацията, в този случай ще имат възможност да игнорират новини от съответната тема.

Новините имплементираме с NotificationsComponent:

const apigithubPrefix = "https://api.github.com/";

@Component({

selector: "Notifications",

templateUrl: "notifications.component.html",

styleUrls: ["notifications.component.css"]

})

export class NotificationsComponent {

public notifications: Notification[];

public loading: boolean = true;

constructor(private github: GitHubService, private router: Router, private route: ActivatedRoute, private location: Location) {

console.log("Create NotificationsComponent");

}

public ngOnInit() {

this.github.request("notifications").then(result => {

this.notifications = result;

this.loading = false;

});

}

public onNotificationTap(args: ItemEventData) {

let notification = this.notifications[args.index];

console.log("Tapped on " + notification.subject.title);

let url = notification.subject.url;

if (url.substr(0, apigithubPrefix.length) === apigithubPrefix) {

let urlTail = url.substr(apigithubPrefix.length).split("/");

console.log("Navigate: [" + urlTail.join(", ") + "]");

this.router.navigate(urlTail);

}

}

}

При навигация към този компонент Angular 2 извиква ngOnInit(). Компонентът стартира зареждане на новините от GitHubService, където значението на request с „notifications” стрингов аргумент е натоварено както следва:

request(notifications: "notifications"): Promise<Notification[]>;

Това описание при компилация от TypeScript до JavaScript се губи, но позволява статично да се проверяват типовете. Интерфейсът Notification е описан в модула с GitHubService и изброява полетата в JSON, който се връща като резултат от GitHub:

export interface Notification {

id: string;

repository: {

id: string;

name: string;

full\_name: string;

description: string;

private: boolean;

};

subject: {

title: string;

url: string;

type: "Issue"

};

unread: boolean;

updated\_at: string;

last\_read\_at: string;

reason: "subscribed" | "manual" | "author" | "mention" | "team\_mention" | "state\_change" | "assign";

}

Първоначално страничката за новини показва индикатор за активност (малко въртящо се кръгче), което след получаване на масивът с новини, изчезва и се появява списък с новините.

Шаблонът с потребителски интерфейс:

<ActionBar title="Notifications">

<NavigationButton android.systemIcon="ic\_menu\_back" (tap)="location.back()"></NavigationButton>

</ActionBar>

<GridLayout>

<ListView [items]="notifications" (itemTap)="onNotificationTap($event)">

<template let-notification="item">

<GridLayout class="notification" rows="auto, auto, auto" columns="auto, \*">

<Image class="icon" rowSpan="3" [src]="notification.repository ? notification.repository.owner.avatar\_url : null"></Image>

<Label row="0" col="1" [ngClass]="notification.unread ? 'title' : 'title unread'" [text]="notification.subject.title" textWrap="true"></Label>

<Label row="1" col="1" class="repo" [visibility]="notification.repository ? 'visible' : 'collapsed'" [text]="notification.repository ? notification.repository.full\_name : ''"></Label>

<Label row="2" col="1" class="status" [text]="notification.subject.type + ', ' + notification.reason + (notification.unread ? ', unread' : '')"></Label>

</GridLayout>

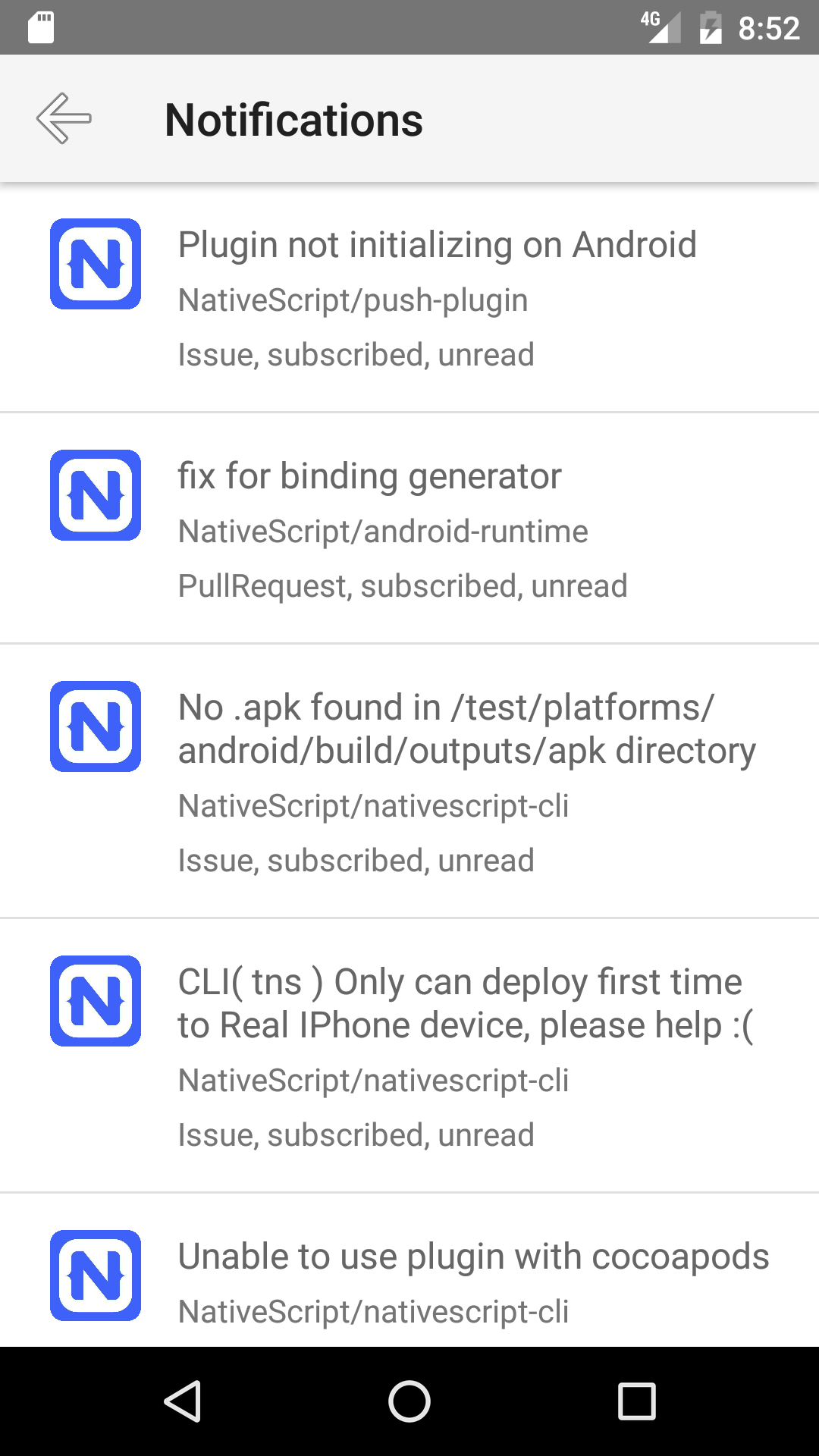
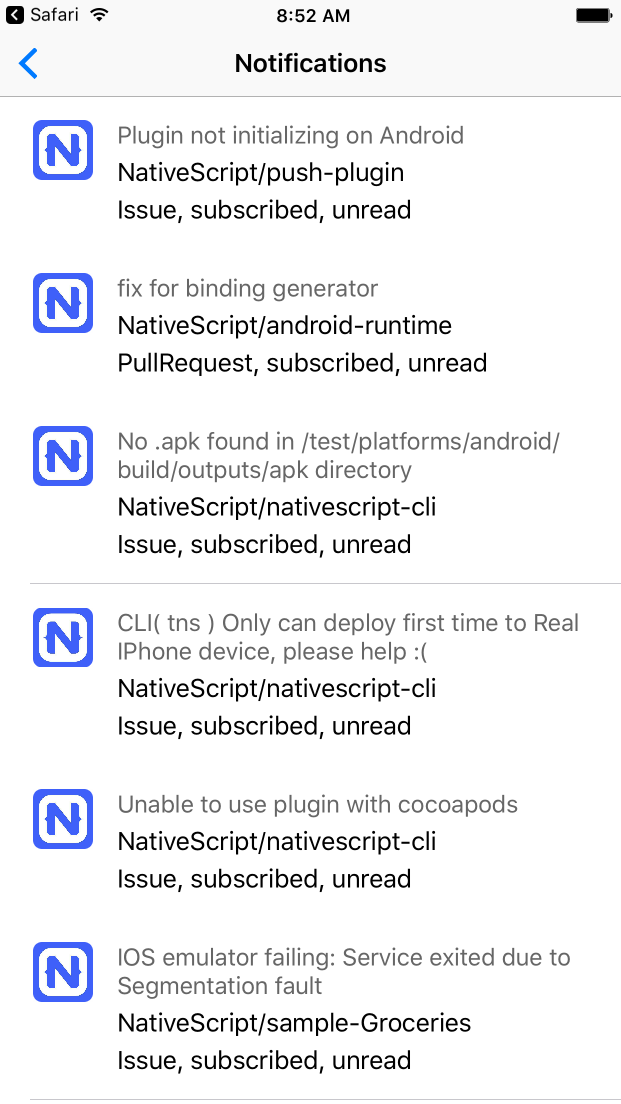
</template>

</ListView>

<ActivityIndicator [busy]="loading"></ActivityIndicator>

</GridLayout>

Описва заглавната лента със бутон за връщане назад. А списъкът с новини показва всяка от тях с аватара на хранилището към което принадлежи, заглавието на новината, името на хранилищитето и типа – проблем (issue) или код за ревю (pull request):



Списъкът е интерактивен, при натискане на елемент, приложението навигира към страничка, на която може да се проследи целият разговор.

## Хранилища

Хранилищата в GitHub съдържат кодът на съответен проект заедно с историята за развитието му. При навигиране от началната страница към страницата с хранилища се показва списък за потребителя, ако е автентикирал приложението. Дори да не е, е възможно търсене на публични хранилища за преглед.

API се консумира чрез заявка към:

* GET /user/repos – връща хранилища на потребителя
* GET /search/repositories/&q=<search> - търси хранилища по дума

В GitHubService request(“user”, “repos”) е описан като:

request(user: "user", repos: "repos", UserReposQuery): Promise<Repository[]>;

export interface Repository {

name: string;

full\_name: string;

forks: number;

open\_issues: number;

watchers: number;

private: boolean;

owner: Owner;

}

Имплементирани са в компонента RepositoriesComponent:

@Component({

selector: "Repositories",

templateUrl: "repositories.component.html",

styleUrls: ["repositories.component.css"]

})

export class RepositoriesComponent implements OnInit, OnDestroy {

private \_searchText: string;

public repositories: Repository[];

public loading: boolean = false;

private authorizeChangeSubscription: any;

constructor(private github: GitHubService, private router: Router, private zone: NgZone, private location: Location) {

console.log("Create RepositoriesComponent!");

}

ngOnInit() {

this.authorizeChangeSubscription = this.github.authorizedChange.subscribe(next => {

if (this.github.authorized && !this.repositories) {

this.listOwnRepos();

}

});

}

ngOnDestroy() {

this.authorizeChangeSubscription.unsubscribe();

}

При инициализиране на компонента, се абонира за authorizationChange на GitHubService. По този начин ако потребителя завърши процес по авторизиране на приложението докато приложението е на този компонент, промяната ще бъде отразена като се заредят хранилищата на потребителя.

Изпращането на заявка за собствени хранилища се случва в listOwnRepos:

private listOwnRepos() {

if (!this.github.authorized) {

return;

}

console.log("List own repos!");

this.loading = true;

this.github.request("user", "repos").then(

result => {

this.repositories = result;

this.loading = false;

},

error => {

console.log("Error: " + error);

this.loading = false;

}

);

}

А за търсене текстово поле е свързано с searchText поле.

Компонентът SearchBox и в двете платформи показва и X бутон, чрез който текстът за търсене се изчиства. Изчистват се и резултатите в onClear, като евентуално се заменят със собствените хранилища:

public onClear() {

console.log("onClear");

this.listOwnRepos();

}

При въвеждане на текст в полето за търсене и натискане на бутонът „Enter” на клавиатурата, се стартира заявката за търсене чрез извикване на метода onSearch:

public onSearch(text: string) {

console.log("onSearch: " + text);

this.loading = true;

this.github.request("search", "repositories", { q: text }).then(

result => {

console.log("Success: " + result.items.map(r => r.full\_name).join(", "));

this.repositories = result.items;

this.loading = false;

},

error => console.log("Error: " + error)

);

}

Описанието на потребителският интерфейс наподобява по структура този за новини. Списъкът със хранилища отново включва шаблон с аватар за хранилището, име и статус в който са описани брой на хората които следят хранилището, брой на отворени проблеми и разклонения (forks).

Преди списъка е добавено поле за търсене, а върху списъка индикатор за активност, който да се появява докато се извършва заявка.

<ActionBar title="Repos">

</ActionBar>

<GridLayout rows="auto, \*">

<SearchBar #searchbar row="0" hint="Search" [text]="searchPhrase" (clear)="onClear()" (submit)="onSearch(searchbar.text)"></SearchBar>

<ListView row="1" [items]="repositories" rowHeight="72" (itemTap)="onRepositoryTap($event)">

<template let-repository="item">

<GridLayout class="repo" rows="auto, auto" columns="auto, \*">

<Image class="icon" rowSpan="2" [src]='repository.owner.avatar\_url'></Image>

<Label class="title" row="0" col="1" [text]='repository.full\_name'></Label>

<Label class="private" row="0" col="1" text="private" [visibility]="repository.private ? 'visible' : 'collapsed'"></Label>

<Label class="status" row="1" col="1" [text]='"★" + repository.watchers + " &#09;⑂" + repository.forks + " &#09;🐛 " + repository.open\_issues'></Label>

</GridLayout>

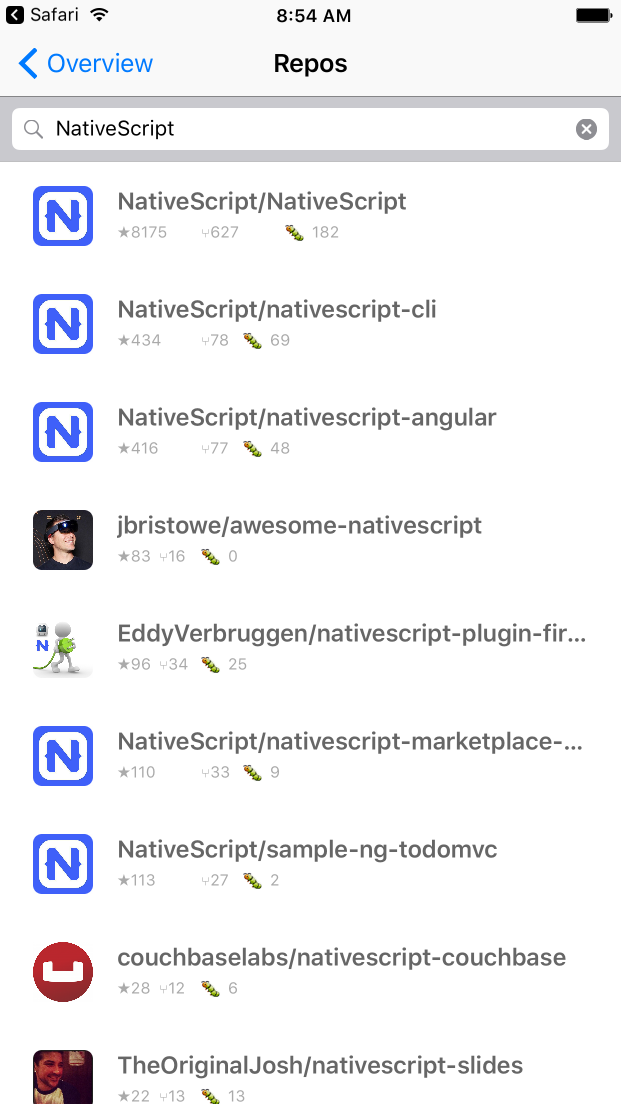
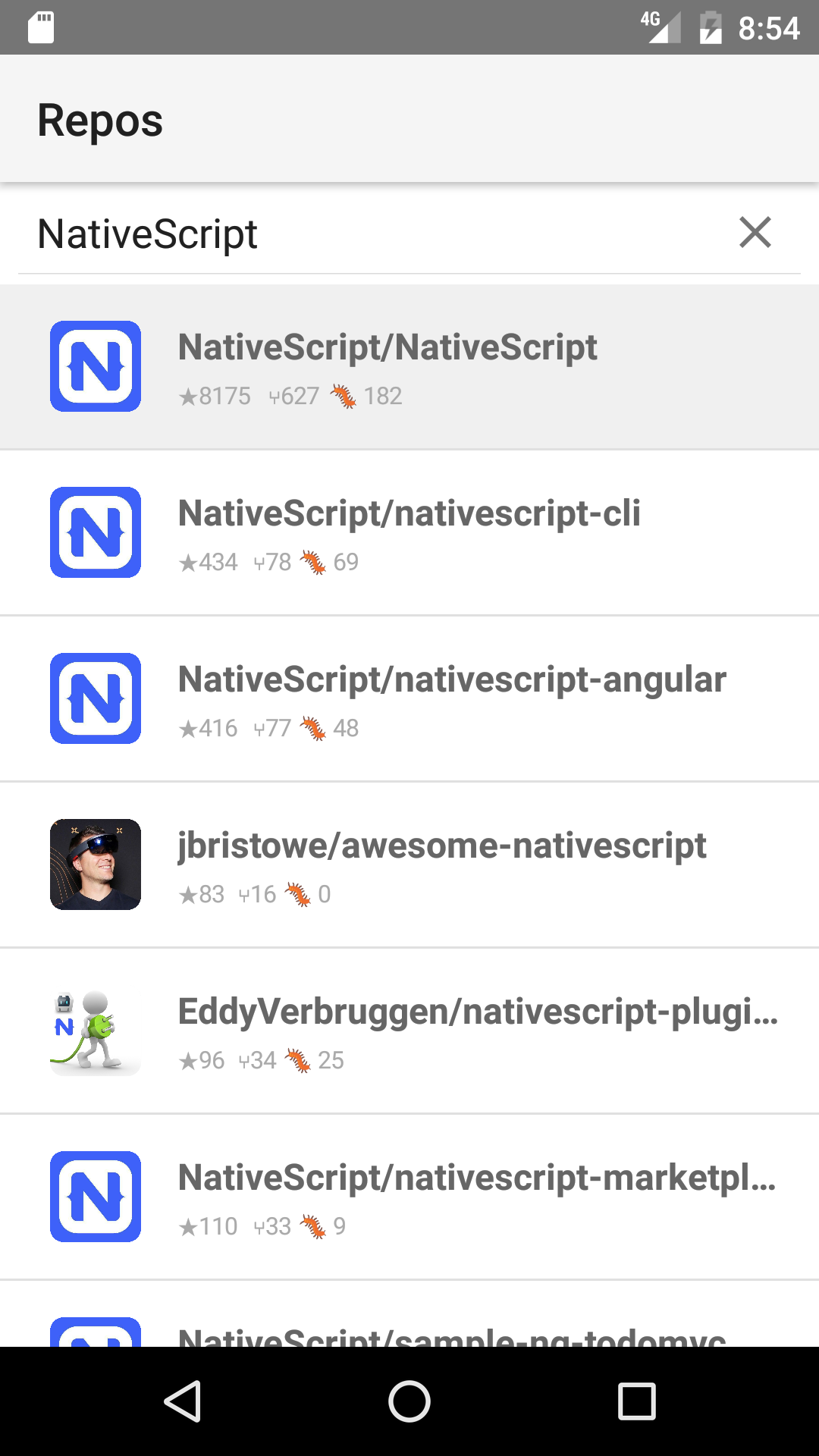
</template>

</ListView>

<ActivityIndicator [busy]="loading"></ActivityIndicator>

</GridLayout>

Придружен е от CSS който задава отстояния и прави картинките с аватар с леко заоблени ъгли. Това различава аватарите за хранилища, от аватарите на автори, за които използваме кръгли картинки. Ето и как изглежда компонента в Android и съответно iOS:



## Въпрои и Събития

(issues, milsetones)

Хранилищата в GitHub по подразбиране са с включена поддръжка за въпроси (issue). Заявки за желани качества на продукта, оплакване от бъгове и прочие се регистрират под формата на въпроси от клиенти. Също така разработчиците на съответното хранилище могат да описват задачи, които да бъдат изпълнени от общността, или да са просто насочващи за развитието на продукта. Собственикът на продукта може да описва въпроси от по високо ниво, които съответно потребителите да си разделят и развият в по-малки под-въпроси, да ги асоциират един с друг и да започнат работа по тях.

* GET /issues – връща всички въпроси асоциирани с потребителя
* GET /orgs/:org/issues – връща всички въпроси в организация асоциирани с потребителя
* GET /repos/:owner/:repo/issues – връща всички въпроси в хранилище, възможно е да бъдат избрани само въпроси за важно събитие (milestone)
* GET /repos/:owner/:repo/issues/:number – връща информация за конкретен въпрос

За създаване и модифициране се използват

* POST /repos/:owner/:repo/issues – създаване на въпрос
* PATCH /repos/:owner/:repo/issues/:number – промяна на въпрос

Последният може да бъде използван за отваряна и затваряне на въпрос, както и за смяна на етикети. Етикетите често се използват за определяне на състоянието на въпроса, което е аналогично на преместване на етикетче по Скръм или Канбан дъските.

Важните събития (milestones) могат да бъдат използвани за да бъдат описване Скръм спринтове, и да бъдат асоциирани въпроси с тях. Визуализирането на едно такова важно събитие (milestone) се доближава и може да замести физическа Скръм дъска.

При избор на хранилище приложението навигира към RepositoryComponent показващ отворени и затворени събития.

@Component({

selector: "Repository",

templateUrl: "repository.component.html",

styleUrls: ["repository.component.css"]

})

export class RepositoryComponent {

ngOnInit() {

this.route.params

.map(params => ({ owner: decodeURIComponent(params['owner']), name: decodeURIComponent(params['name']) }))

.subscribe(params => {

console.log("Owner: " + params.owner + ", Name: " + params.name);

this.owner = params.owner;

this.name = params.name;

this.github.request("repos", params.owner, params.name, "milestones", { state: "all" }).then(result => {

this.milestones = result;

this.openMilestones = result.filter(milestone => milestone.state === "open");

this.closedMilestones = result.filter(milestone => milestone.state === "closed");

});

});

}

При инициализация се декодира пътят зададен при навигация от Angular 2 модулите, за да се извлече собственик и име на хранилище. Чрез тях се изпраща заявка чрез GitHubService request:

request(repos: "repos", owner: string, repo: string, milestones: "milestones", querry?: MilestonesQuery): Promise<Milestone[]>;

export interface Milestone {

title: string;

id: number;

number: number;

description: string;

creator: Owner;

open\_issues: number;

closed\_issues: number;

state: "open" | "closed";

created\_at: string;

updated\_at: string;

due\_on: string;

closed\_at: string;

}

При реализация на потребителският интерфейс се използва TabView. Компонентът изглежда различно за Android и iOS. В едната платформа бутоните за смяна на съдържанието се позиционират върху съдъжанието, а в другата – под него. Ето и шаблонът за потребителски интерфейс:

<ActionBar [title]="owner + '/' + name">

<NavigationButton android.systemIcon="ic\_menu\_back" (tap)="goBack()"></NavigationButton>

</ActionBar>

<TabView>

<ListView \*tabItem="{title: 'Open'}" [items]="openMilestones" rowHeight="72" (itemTap)="onOpenMilestoneTap($event)">

<template let-milestone="item">

<!-- … -->

</template>

</ListView>

<ListView \*tabItem="{title: 'Closed'}" [items]="closedMilestones" rowHeight="72" (itemTap)="onClosedMilestoneTap($event)">

<template let-milestone="item">

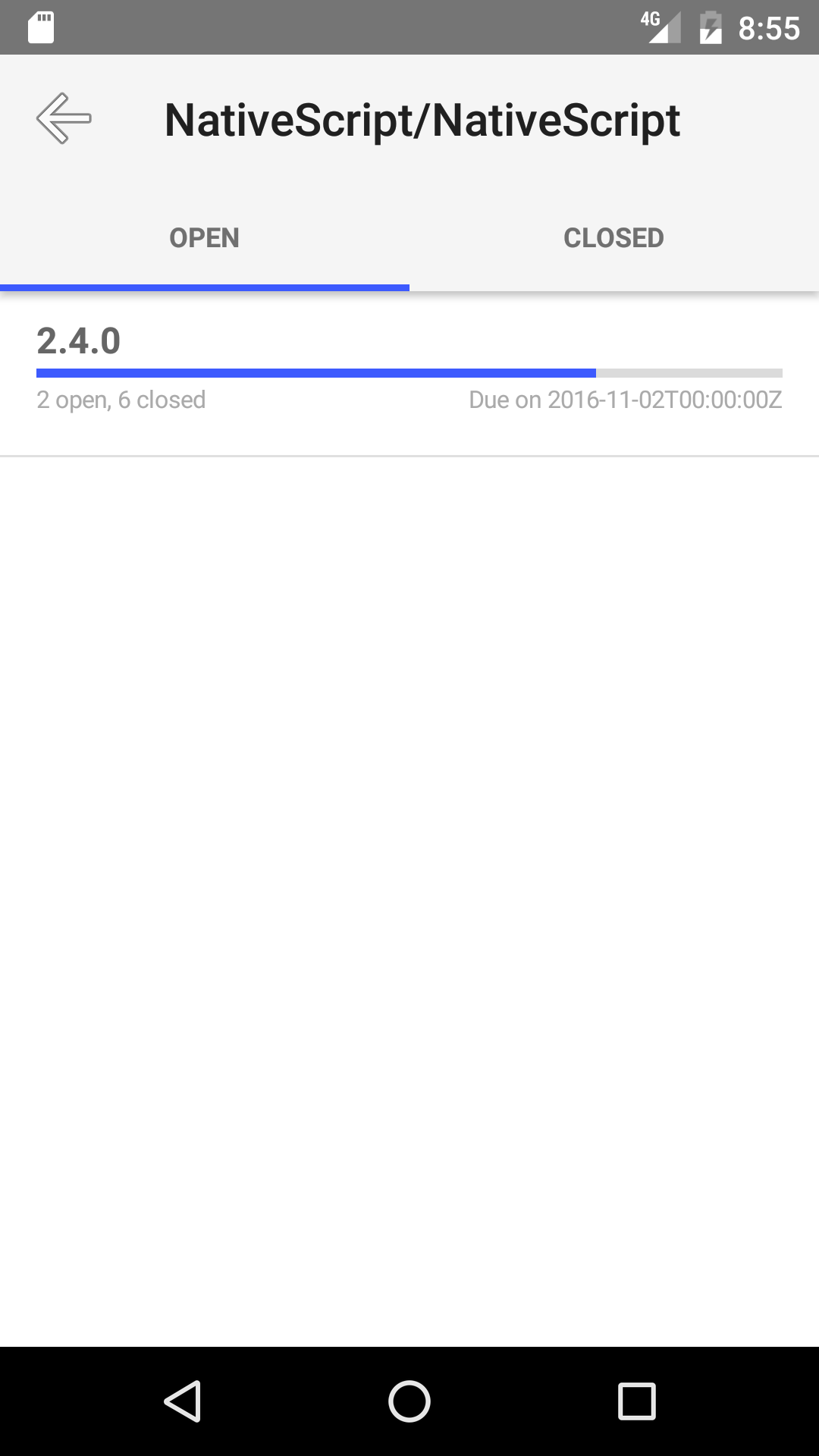
<!-- … -->

</template>

</ListView>

</TabView>

Потребителите на приложението ще получат изглед, с който са свикнали на съответната операционна система:



Съответно и компонентите се държат по различен начин. В Android при плъзгане хоризонтално върху съдържанието, се премества постепенно между двата панела с отворени и затворени събития. Докато в iOS при плъзване от най-левия край на екрана в дясно се извършва Back-Swipe жест по подобен на Android начин се показва предната страница. Това в iOS е обосновано, тъй като за разлика от Android няма хардуерен бутон за навигация назад.

След избор на събитие се показват всички проблеми асоциирани със събитието. Навигацията е към MilestoneComponent:

@Component({

selector: "Milestone",

templateUrl: "milestone.component.html",

styleUrls: ["milestone.component.css"]

})

export class MilestoneComponent {

ngOnInit() {

this.route.params

.map(params => ({ owner: decodeURIComponent(params['owner']), name: decodeURIComponent(params['name']), id: decodeURIComponent(params['milestone']) }))

.subscribe(params => {

console.log("Owner: " + params.owner + ", Name: " + params.name);

this.owner = params.owner;

this.name = params.name;

this.number = parseInt(params.id);

this.github.request("repos", params.owner, params.name, "issues", { milestone: params.id, state: "all" }).then(

(result: Issue[]) => {

this.issues = result;

this.openIssues = result.filter(issue => issue.state === "open");

this.closedIssues = result.filter(issue => issue.state === "closed");

if (this.github.authenticatedUser) {

this.myIssues = result.filter(issue => issue.assignees && issue.assignees.some(asignee => {

console.log(" - " + asignee.login);

return asignee.login === this.github.authenticatedUser.login;

}));

} else {

this.myIssues = null;

}

}

);

});

}

При инициализация се изпраща заявка чрез:

request(repos: "repos", owner: string, repo: string, issues: "issues", query?: IssuesQuery): Promise<Issue[]>;

export interface Issue {

id: number;

number: number;

title: string;

user: Owner;

labels: Label[];

state: "open" | "closed";

locked: boolean;

assignee: Owner;

assignees: Owner[];

milestone: Milestone;

comments: number;

created\_at: string;

updated\_at: string;

closed\_at: string;

body: string;

reactions?: {

url: string;

total\_count: number;

"+1": number;

"-1": number;

laugh: number;

hooray: number;

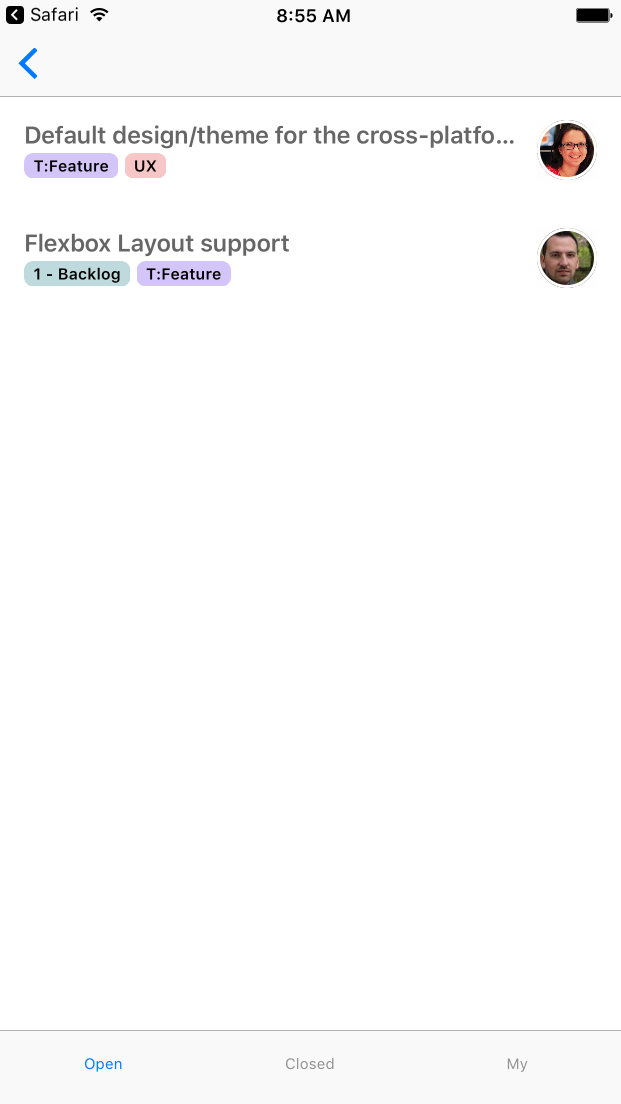
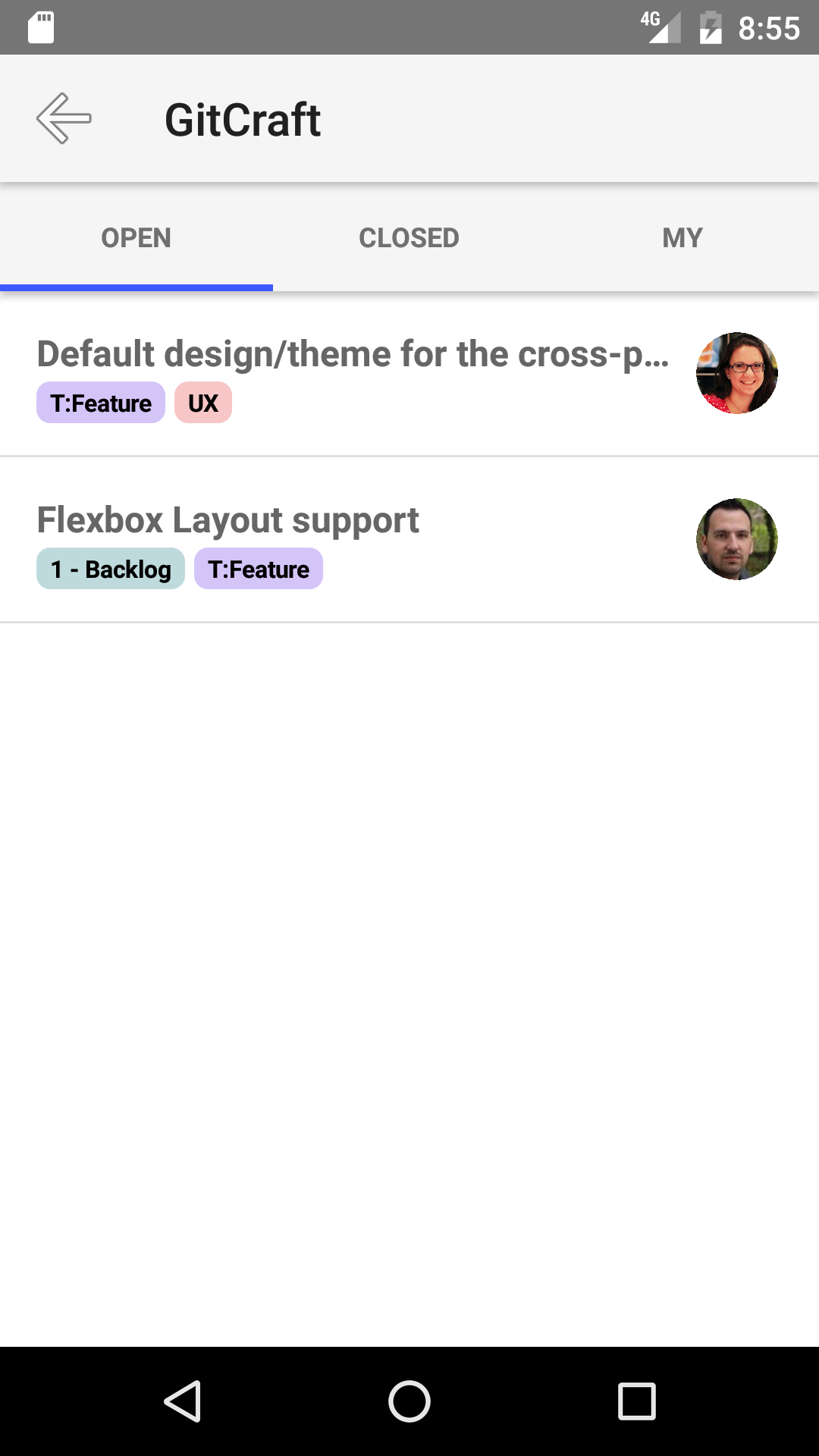
confused: number;

heart: number;

}

}

Въпросите (issues, задачи) се показват разделени според статуса си в TabView. Като освен отворени/затворени, ако потребителя е авторизирал приложението за достъп до данните си се добавя и категория „мои“:



## Въпроси (задачи)

(issues)

При избор на върпос (issue, задача) от таблото за събитие или от екрана с новини, се навигира към IssueComponent. В него могат да се видят детайли и коментари по съответната тема.

@Component({

selector: "Issue",

templateUrl: "issue.component.html",

styleUrls: ["issue.component.css"],

})

export class IssueComponent implements OnInit {

public ngOnInit() {

this.route.params

.map(params => ({ owner: decodeURIComponent(params['owner']), name: decodeURIComponent(params['name']), number: decodeURIComponent(params['issue']) }))

.subscribe(params => {

this.owner = params.owner;

this.name = params.name;

this.number = params.number;

this.github.request("repos", this.owner, this.name, "issues", this.number).then(result => {

this.issue = result;

this.github.request("repos", this.owner, this.name, "issues", this.number, "comments").then(result => {

this.comments = result;

});

});

});

}

При инициализиране на компонента се правят две последователни заявки. Първата е за заглавието, текста и атрибутите на темата. Втората е за коментарите по темата:

request(repos: "repos", owner: string, repo: string, issues: "issues", number: string): Promise<Issue>;

request(repos: "repos", owner: string, repo: string, issues: "issues", number: string, comments: "comments"): Promise<Comment[]>;

export interface Comment {

id: number;

user: {

login: string;

id: number;

avatar\_url: string;

};

created\_at: string;

updated\_at: string;

body: string;

}

Шаблонът за потребителски интерфейс е както следва:

<ActionBar [title]="github.authenticatedUser.login">

<NavigationButton android.systemIcon="ic\_menu\_back" (tap)="location.back()"></NavigationButton>

</ActionBar>

<ScrollView>

<GridLayout>

<StackLayout class="issue" [visibility]="issue ? 'visible' : 'collapsed'">

<Label class="title" [text]="issue && issue.title" textWrap="wrap"></Label>

<Label class="number" [text]="issue && ('#' + issue.number)"></Label>

<Label class="state" [text]="issue && issue.state"></Label>

<GridLayout columns="auto, \*">

<Label col="1" class="created" [text]="issue && (issue.user.login + ' opened this issue on ' + issue.created\_at)" textWrap="true"></Label>

</GridLayout>

<StackLayout class="asignees" rowSpan="3" col="1" orientation="horizontal">

<Image class="assignee" \*ngFor="let assignee of (issue && issue.assignees || []); let i = index" [class.first]="i === 0" [src]="assignee.avatar\_url" [style.zIndex]="-i"></Image>

</StackLayout>

<StackLayout class="labels" row="1" orientation="horizontal">

<Label class="label" \*ngFor="let label of (issue && issue.labels || [])" [text]="label.name" [color]="label.color | color" [backgroundColor]="label.color | backgroundColor"></Label>

</StackLayout>

<Label class="body" [text]="issue && (issue.body)" textWrap="wrap"></Label>

<Label [text]="issue && (issue.comments + ' comments')"></Label>

<GridLayout class="comment" \*ngFor="let comment of comments" rows="auto, auto" columns="auto, \*">

<Image class="avatar" [src]="comment.user.avatar\_url"></Image>

<Label col="1" [text]="comment.user.login"></Label>

<Label row="1" colSpan="2" [text]="comment.body" textWrap="true"></Label>

</GridLayout>

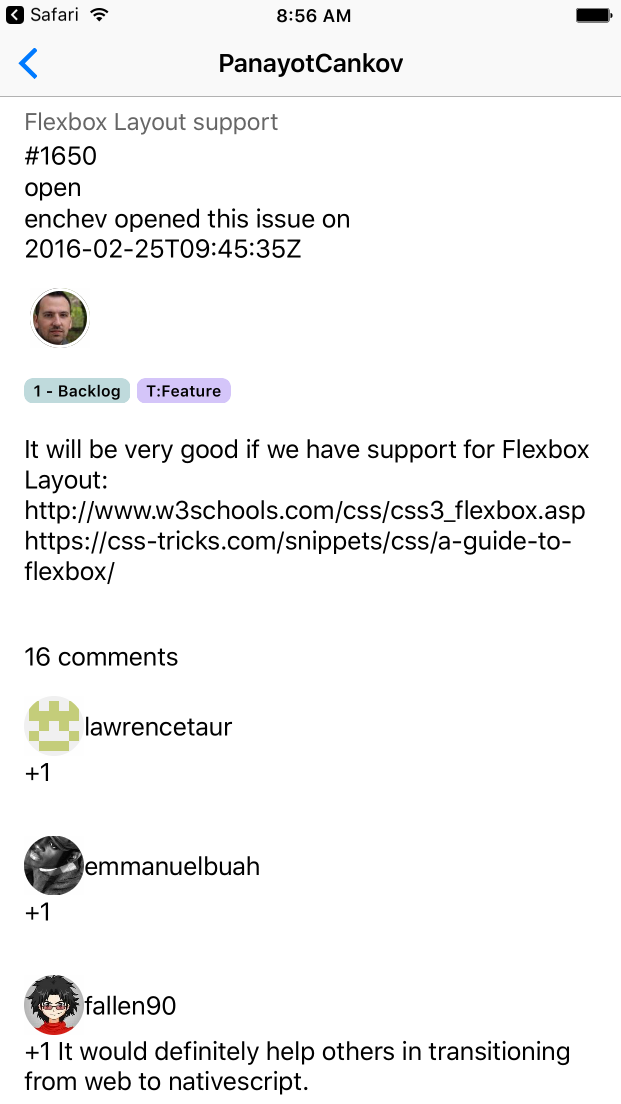
</StackLayout>

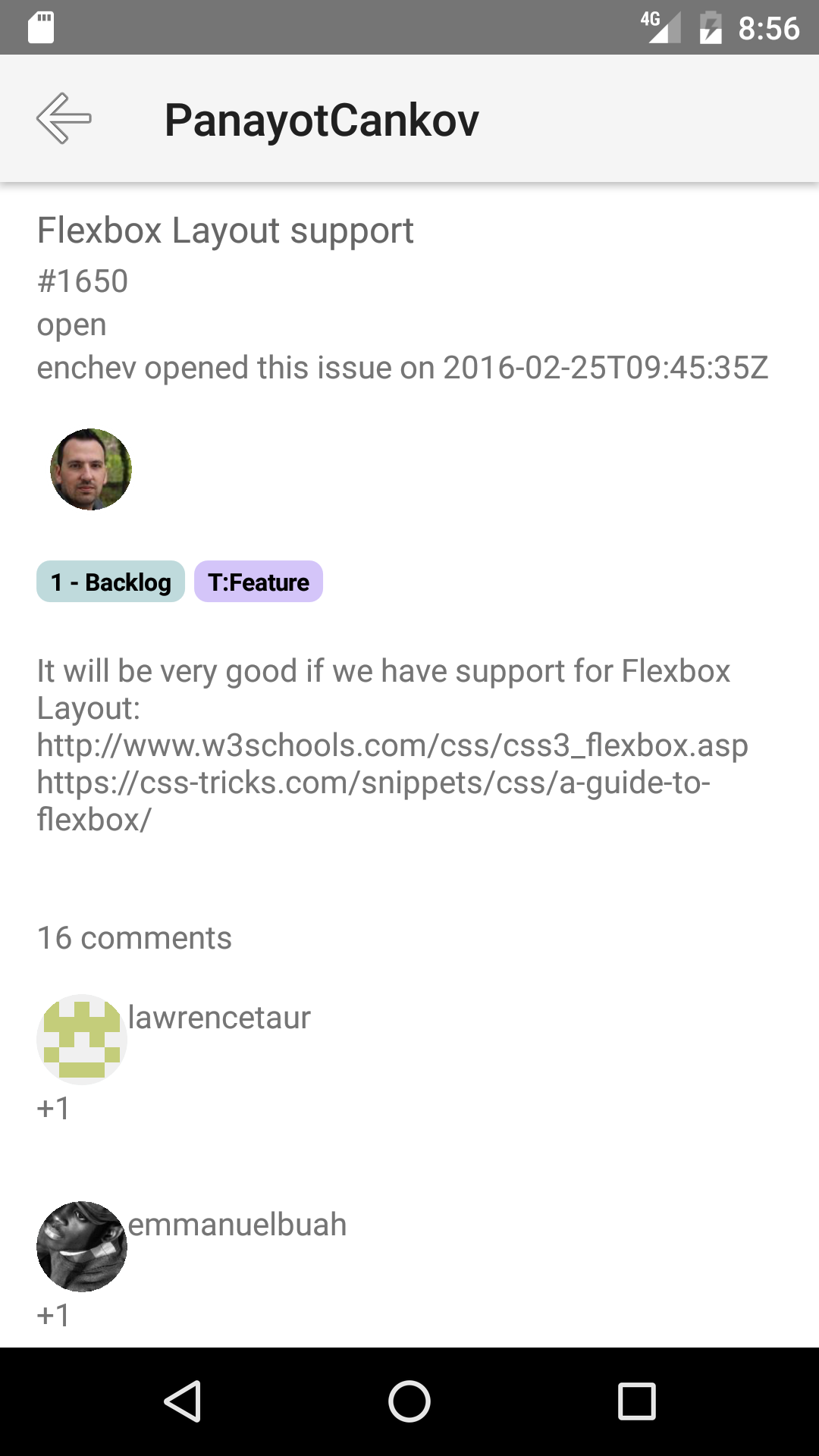
</GridLayout>

</ScrollView>

Най-отгоре се изписват заглавието на върпоса, последван от инфомрация за статус, дати на създаване и промяна. Следва списък със асоциираните разработчици, чрез аватарите им. Етикети свързани със въпроса.

Следват коментарите, един под друг се нареждат аватар на GitHub потребител, който е коментирал, име и текст на коментара.





# Заключение

Избраният технологичен стек позволи да изработим приложение с потребителски интерфейс, който максимално да се доближава до присъщия за операционната система.

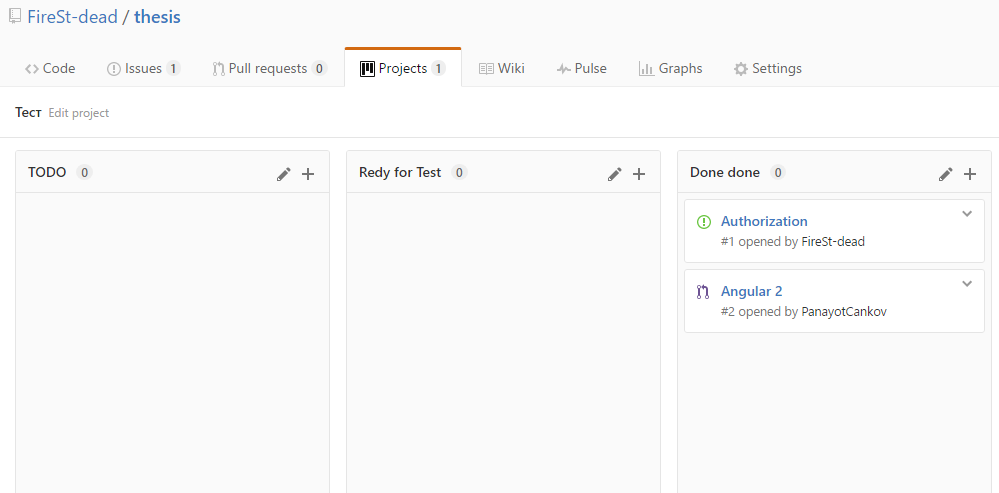
По време на разработката на приложението основните проблеми бяха поддържането на проекта в синхрон с последните версии на използваните библиотеки. При стартирането на проекта Angular 2 все още не бе официален и протекоха пускове на кандидати за официални версии RC4, RC6 (Relase Candidate, кандидате за пуск) както и официалната му версия заедно с нова официална версия на NativeScript. Изборът на TypeScript позволява бързото откриване на промени, които засягат потребителски код (в приложението), а документацията на Angular 2 и NativeScript бяха своевременно обновявани.

Приложението е работещо за две платформи но в момента поддържа само базова функционалност за целите му.

Тепърва може да се възползва от функционалността на мобилните платформи и да:

* Се регистрира като доставчик за резултати при търсене
* Да следи за новини на заден фон и да нотифицира за съобщения
* Да обновява иконата на приложението със брои непрочетени новини
* Да запазва токена за достъп пезопасно в Keychain за iOS и AccountManager за Android
* Да се разработи интерактивен копонент за смяна на статус на въпрос
* Да се разработи пълна Скръм/Канбан дъска за таблет
* Да се автоматизира преместване на въпрос от едно хранилище в друго

Идеята в HuBoard етикети да се използват за статус на въпрос и да отбелязват местоположението му на Скръм/Канбан дъска бе единствената възможност подобна реализация да не зависи от допълнителен сървър, но GitHub също се развива и пусна версия на сайта си с „Проекти“. Към всяко хранилище вече могат да се добавят проекти, като GitHub предоставя уеб потребителски интерфейс:



Това напълно ще инвалидира бизнеса на HuBoard, но пък в момента, в който предоставят публично API за работа с проекти, ще може да се интегрират с приложението.

# Източници

1. Agile Manifesto, 2001, <http://agilemanifesto.org/>, // Март 2013г.
2. Beck, Kent. Principles behind the Agile Manifesto. // Agile Alliance, 2001г. <http://agilemanifesto.org/principles.html>
3. Anderson, David. Declaration of Interdependence. // 2005г. <http://pmdoi.org/>
4. Ken Schwaber and Jeff Sutherland. The Scrum Guide // Юли 2013г. <http://www.scrumguides.org/docs/scrumguide/v1/Scrum-Guide-BG.pdf>
5. Книберг Х., М. Скарин. Канбан и Скръм - как да вземем най-доброто и от двете // 2010г.
6. Taiichi Ohno, Toyota Production System, Productivity Press // 1988г.
7. Erik Brechner, Agile Project Management with Kanban // 2015г. Microsoft Press
8. Henrik Kniberg, Scrum and XP from the Trenches // 2007г.
9. Eric S. Raymond – Cathedral and the Bazaar // 1999г., превод Никола Колев, Калоян Доганов 2000г.
10. P. Frederick, Jr. Brooks – The Mythical Man-Month // Август 2002г.
11. Robles, Gregorio – A Software Engineering Approach to Libre Software // 2004г.
12. HuBoard - <https://huboard.com/> // Октомври 2016г.
13. TeamPulse - <https://docs.telerik.com/teammpulse/> // Октомври 2016г.
14. John W. Chinneck Practical Optimization: A gentile introduction // June 2015г.
15. Aha! - <http://www.aha.io/> // Октомври 2016г.
16. Cordova - <https://cordova.apache.org/docs/en/6.x/guide/overview/> // Октомври 2016г.
17. Xamarin - <https://developer.xamarin.com/guides/cross-platform/application_fundamentals/> // Октомври 2016г.
18. NativeScript – <https://www.infoq.com/news/2015/03/nativescript/> // Октомври 2016г.
19. ReactNative - <https://www.infoq.com/articles/react-native-introduction> // Октомври 2016г.

1. [Tadeu Zagallo](https://plus.google.com/+tadeuzagallo/posts), Bridging in React Native - <http://tadeuzagallo.com/blog/react-native-bridge/> // Октомври 2016г.

1. [Kent Beck](https://bg.wikipedia.org/w/index.php?title=Kent_Beck&action=edit&redlink=1), Mike Beedle, Arie van Bennekum, [Alistair Cockburn](https://bg.wikipedia.org/w/index.php?title=Alistair_Cockburn&action=edit&redlink=1), [Ward Cunningham](https://bg.wikipedia.org/w/index.php?title=Ward_Cunningham&action=edit&redlink=1), [Martin Fowler](https://bg.wikipedia.org/w/index.php?title=Martin_Fowler&action=edit&redlink=1), James Grenning, [Jim Highsmith](https://bg.wikipedia.org/w/index.php?title=Jim_Highsmith&action=edit&redlink=1), [Andrew Hunt](https://bg.wikipedia.org/w/index.php?title=Andy_Hunt_(author)&action=edit&redlink=1), [Ron Jeffries](https://bg.wikipedia.org/w/index.php?title=Ron_Jeffries&action=edit&redlink=1), Jon Kern, [Brian Marick](https://bg.wikipedia.org/w/index.php?title=Brian_Marick&action=edit&redlink=1), [Robert C. Martin](https://bg.wikipedia.org/w/index.php?title=Robert_C._Martin&action=edit&redlink=1), [Stephen J. Mellor](https://bg.wikipedia.org/w/index.php?title=Stephen_J._Mellor&action=edit&redlink=1), [Ken Schwaber](https://bg.wikipedia.org/w/index.php?title=Ken_Schwaber&action=edit&redlink=1), [Jeff Sutherland](https://bg.wikipedia.org/w/index.php?title=Jeff_Sutherland&action=edit&redlink=1), and [Dave Thomas](https://bg.wikipedia.org/w/index.php?title=Dave_Thomas_(programmer)&action=edit&redlink=1) [↑](#footnote-ref-1)
2. Scrum [↑](#footnote-ref-2)
3. Kanban [↑](#footnote-ref-3)
4. Autonomation, 自働化, взаимствана дума от Sino-Japanese. [↑](#footnote-ref-4)
5. WIP - Work In Progress [↑](#footnote-ref-5)
6. Rational Unified Process – итеративен процес за разработка на софтуер. [↑](#footnote-ref-6)
7. Pair programming [↑](#footnote-ref-7)
8. PERT - Program Evaluation and Review Technique [↑](#footnote-ref-8)
9. WebView – обобщение на визуалните компоненти за дисплей на HTML в мобилните платформи: iOS - UIWebView, Android - android.webkit.WebView и др. [↑](#footnote-ref-9)