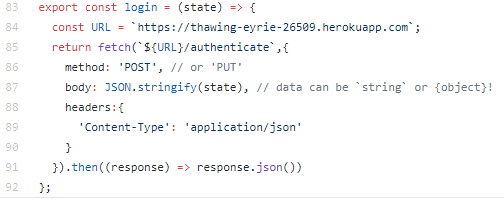
1) Кратки фрагменти код (практически реализации)

**Автентикация на потребители:**

Заявка, отговорна за изпращане на потребителско име и парола за автентикация до сървъра:



В state обекта има полета:

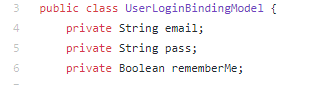
* user
* pass

Те биват изпратени до сървъра ни, който е хостнат в Heroku.

Сървърът приема заявка на /authenticate чрез следния метод от контролера UserJWTController:



Controller-ът получава UserLoginBindingModel, което е клас със следните полета:

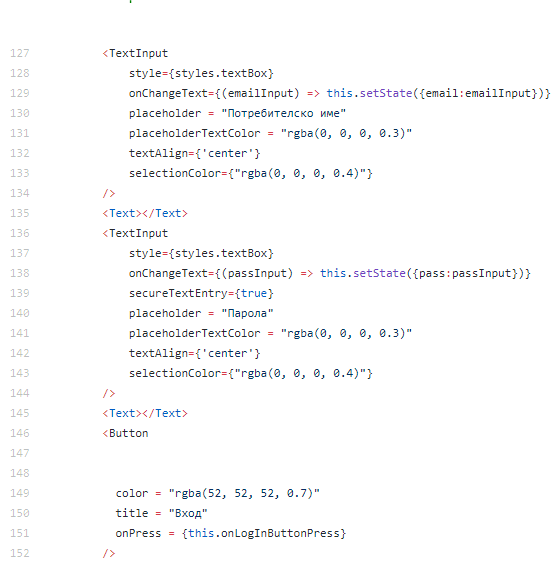


Те съвпадат с тези, които се изпращат от клиентската страна, и по-този начин Spring ги map-ва от тялото на POST заяката към класа UserLoginBindingModel.

След това се проверява дали тези email и pass принадлежат на даден потребител и ако това е така, то сървърът връща token:



В React Native това е компонентът, отговорен за изпращането на данните на потребителя:



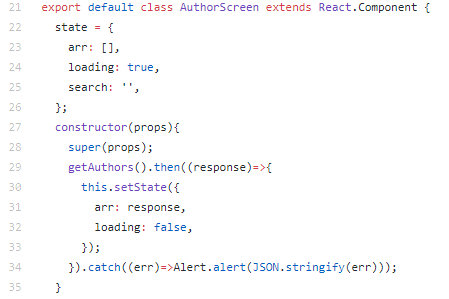
Всеки път, когато потребителят въведе нещо в полето за потребителско име или парола, се извикват call back функции, които обновяват state-ът, който ще се прикачи към заявката до сървъра при натискането на бутона-Вход, който от своя страна ще извика следната функция:



onLogInButtonPress() е декларирана като асинхронна arrow функция(ламбда израз), защото ще използва sendRequest(), което от своя страна връща promise и е необходимо да го await-нем, докато приключи.

Това става асинхронно и event loop-ът на JS не се задръства.

**Показване на български автори и тяхното филтриране по име:**



AuthorScreen е отговорен за изпращане на заявка до сървъра, която ще му върне масив от обекти, съдържащи:

* Имена
* ID
* URL към снимка на автора

state обектът запазва този масив и обновява състоянието си от режим на зареждане в режим на готовност.

Това е функцията, която прави GET заявка до сървъра:



От глобалния обект взимаме token, който е получен при автентикацията. Прикачваме го към header частта на заявката и я изпращаме. Конфигурирали сме всички ресурси на сървъра да изискват автентикация.

От страна на сървъра, тази заявка минава през Spring филтър:



Той проверява дали token-ът е валиден.

След това заявката стига до AuthorController:



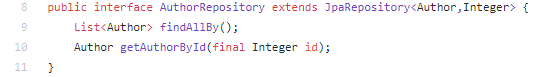
Той използва добрите практики от Spring и ни позволява да използваме dependency injection за следните услуги:

* AuthorService
* WorkService

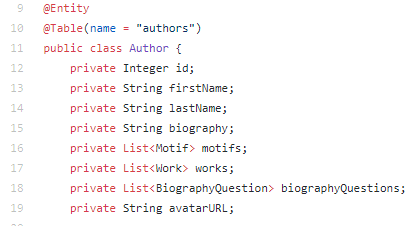
Контролерът на route /authors приема заявката и извиквка метода findAll() на AuthorService, който е имплементиран от UserServiceImpl по следния начин:



Неговата роля е и да извлече списък на всички автори от AuthorRepository. Има dependency, което е интерфейс, играещ ролята на връзка към базата данни:

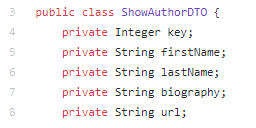


JPARepository ни предоставя готови методи за работа с каквато и да е релационна база данни. По този начин ние не пишем SQL и сме много по-ефикасни. Това Repository работи със следното entity за автор:



Освен данните, които потребителят изисква, авторите имат и списъци от своите мотиви, творби и биографични въпроси върху тях.

Тъй като това entity има прекалено много информация за нуждите на заявката, създаваме следното DTO(Data Transfer Object):



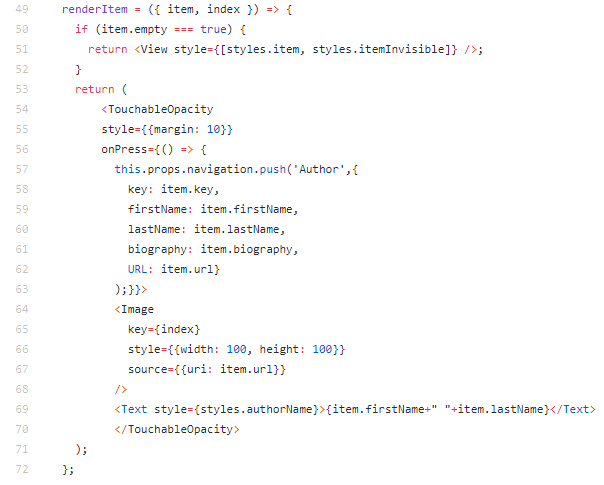
Използвайки този подход, получаваме още едно преимущество - избягване на circular dependency, когато връщаме данните под формата на JSON!

Това е причината Author Service и Controller да връщат:

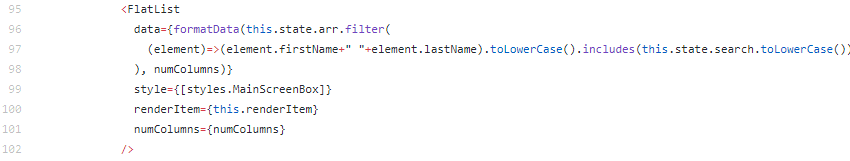


В клиентската страна авторите се визуализират като снимки, под които са прикачени техните имена.

Тази функция е отговорна за тази задача:

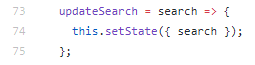


И се извиква в render функцията:



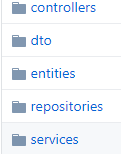
Входните данни се филтрират спрямо state обекта, от който се взима текста на търсачката.

Всеки път когато се въвежда текст в търсачката, се извиква:



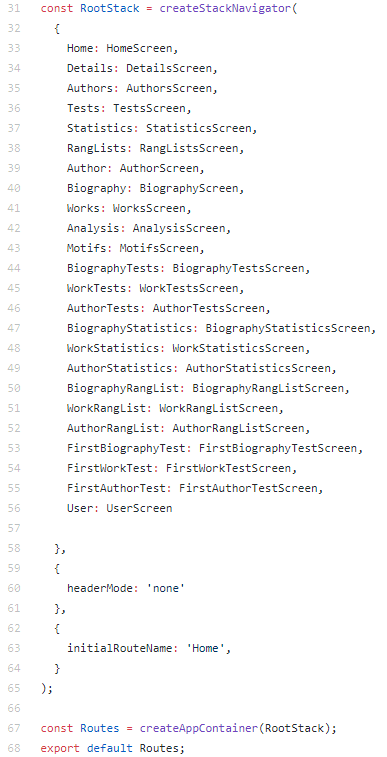
След което се извиква render() функцията и тя визуализира търсения автор спрямо въведеното от потребителя в търсачката име. Facebook са оптимизирали този процес и той по никакъв начин не влоша бързодействието на приложението.

В сървъра всеки модул като Автори, Потребители, Тестове, Резултати има следните компоненти:

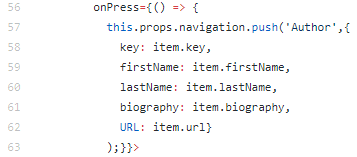


Навигацията в приложението става посредством routes.js

В него е записано как се достъпва всеки екран от клиентската страна. Това позволява лесно надграждане на сегашната логика, защото ако имаме нужда от нов екран, то просто добавяме в routes.js и вече спокойно може да го използваме в приложението,



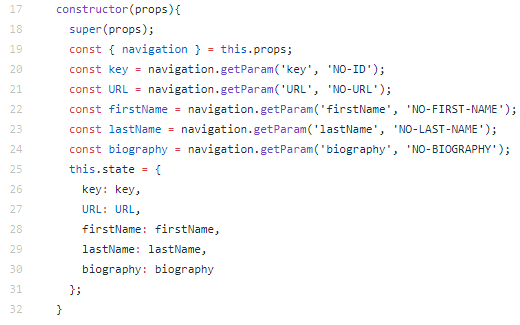
Навигирането между екраните става чрез this.props.navigation.push(‘screenName’,objectData);



В този пример screenName е ‘Author’, което в routes.js отговаря на екрана за автори(AuthorScreen.js)

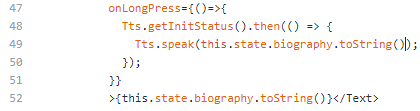
objectData е обект, който го изпращаме на екрана, към който ще навигираме.

Ето как се получават данните в дестинационния екран:



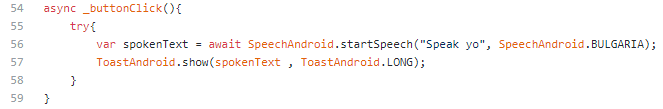
След като ги извлечем данните, ги запазваме в state, за да ги използваме из компонента.

Това е пример как използваме функционалността Text to Speech:



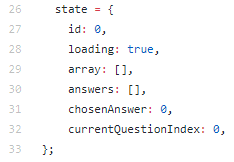
В горния фрагмент, като се задържи екрана за по-дълго време се активира call back функцията, която ще статира Text to speech функционалността и ще изговори подадения текст.

Това е функцията, с която позволяваме на потребителя да изкаже, какво би искал да търси в приложението:



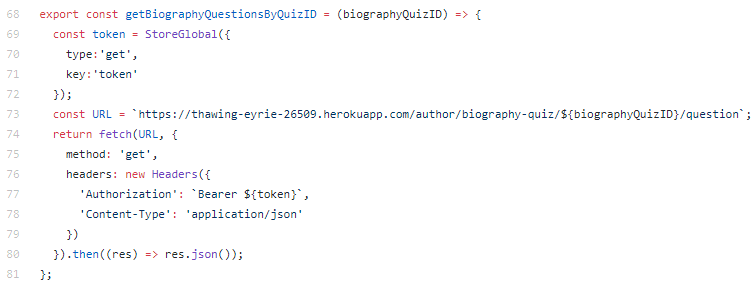
**Правене на тестове:**

State на компонента:

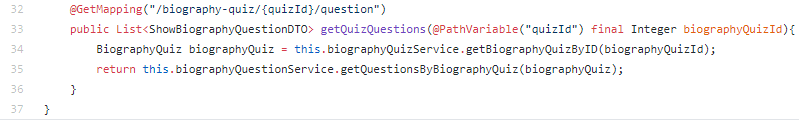


в array ще вкараме въпросите дошли от заявката към сървъра, а пък answers ще запишем отговорите от теста.

Ето я заявката към сървъра:

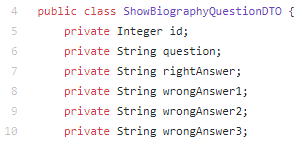


По id на даден quiz правим заявка, която ще извлече всички негови въпроси.  
Заявката се приема от BiographyQuizController на метода:

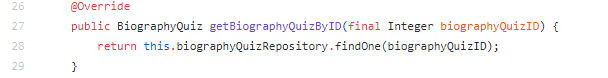


Неговата роля е да извика услуга, която ще намери посочения biographyQuiz по ID и след като го вземе, да извлече всички въпроси, които се отнасят към него и да ги върне като лист от Data Transfer Object.

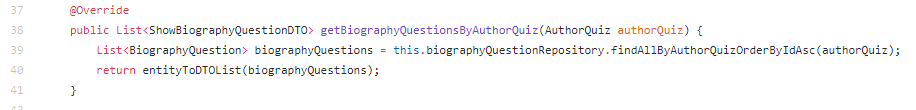
Ето какво представлява ShowBiographyQuestionDTO:



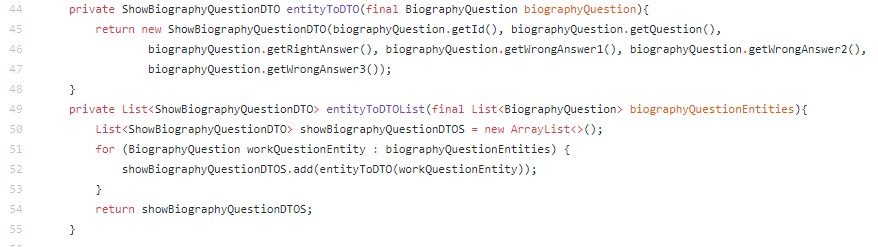
Това е услугата, която намира дадения bigraphyQuiz по ID:



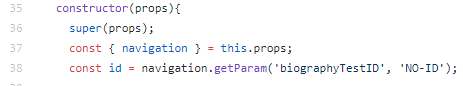
A това е услугата, която по даден bigraphyQuiz намира всички въпроси към него:



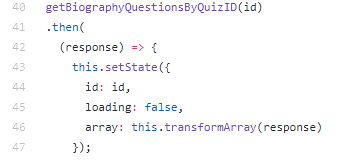
Конвертирането на BigraphyQuistion, което е entity, към ShowBiographyQuestionDTO става чрез следните два метода:



В компонента от клиентската страна получаваме id-то на избрания тест



Ще го подадем на заявката:

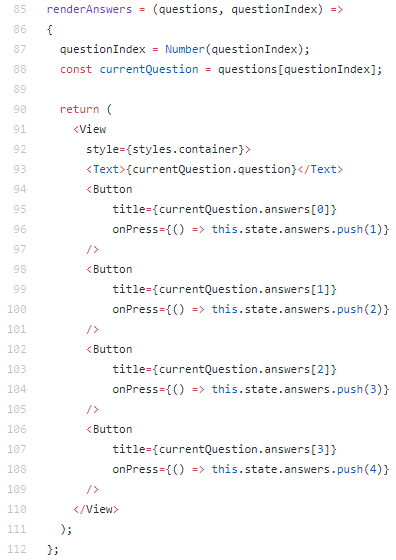


и обновяваме състоянието на компента.

Сега тези въпроси може да ги визуализираме върху екрана, като взехме решение да ги показваме един по един:

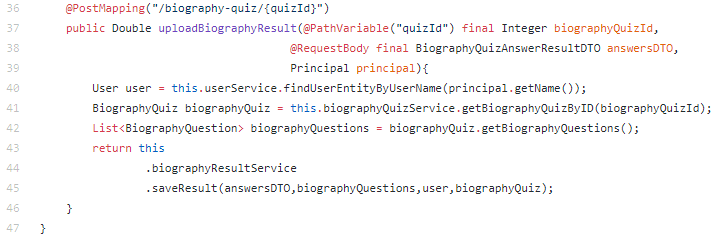


Ето и функцията, която рендира въпрос, по зададен индекс:



Когато потребителя, избере даден отговор, той се добавя като индекс в масив.

С POST заявка изпращаме отговорените въпроси на сървъра, който ги приема в BiographyResultController:



Ролята на uploadBiographyResult() метода е да извика услуга, която да намери теста, и да му вземе всичките въпроси.

В услугата biographyResultService, се подават отговорените въпроси, въпросите от базата данни, потребителя, който ги е отговорил и теста. Услугата прави проверка, колко верни отговора е има потребителя, пресмята неговия резултат, запазва го в базата данни и връща процента на клиентската страна, която ще го визуализира на потребителя.