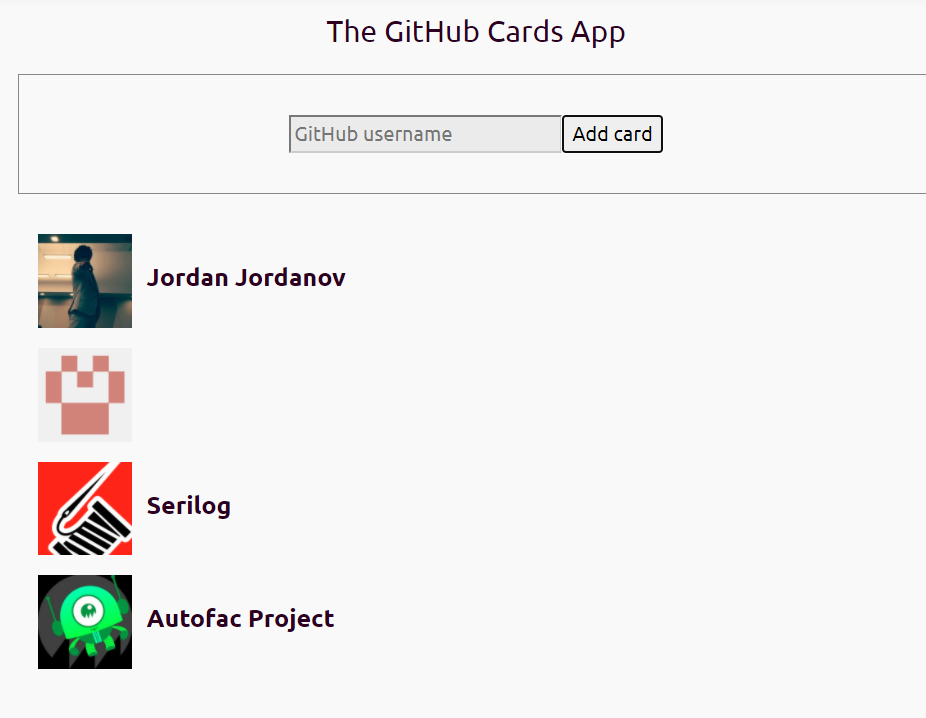
# The GitHub Cards App



***Какво ще изграждаме?***

Виждали сме прости компоненти, работили сме с множество компоненти и сме виждали как и кога да използваме състоянието и свойствата на даден компонент. Все още обаче не сме работили с реални данни. Ще направим точно това в този модул на курса и ще използваме публичния сървиз на GitHub. Ще изградим прост компонент на карта с профили на GitHub, който показва информация за списъци с вашите профили. Ще има формуляр, в който потребителят може да въведе име от GitHub и да използва „Добавяне на карта“ за да добави нов към списъка с показаните профили. Ще се учим как да приемаме входни данни от потребителя и как да ги използваме, за да осъществите повиквания към GitHub API. Целта на това приложение е да ви осигури комфортна работа с обекти от данни. Другото нещо за това приложение е, че ще използваме класови компоненти. А след това по-голямото приложение ще бъде изцяло написано с функционални компоненти и „куки“.

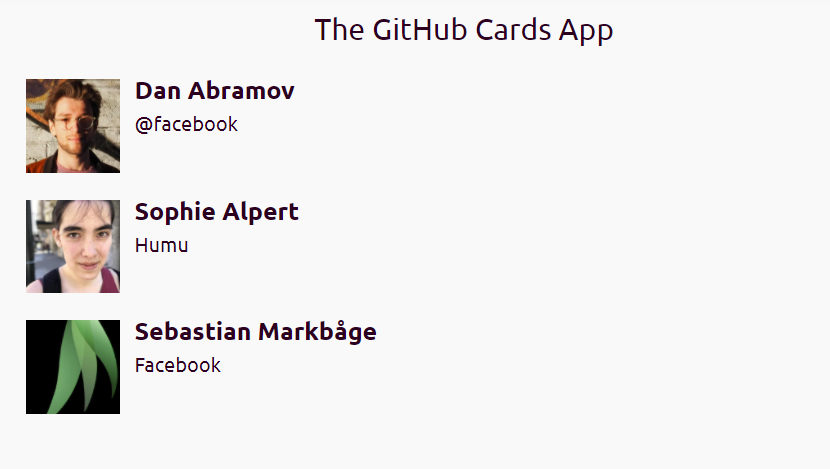
***React Class Components***

В <http://jsdrops.com/rgs2.1> подготвих малък компонент на функцията на приложението и го прикачих към DOM. Ще използваме този, за да съдържа всички останали компоненти в това приложение. Първото решение, което трябва да вземете в приложение на React е структурата на компонентите. Трябва да решите колко компонента да използвате и какво трябва да описва всеки. Това често е лесно, ако имате пълната картина на приложението, което създавате, но на практика нямате. Нашето приложение ще бъде списък с карти от GitHub. Първата ви улика, че имате нужда от компонент, който да представлява една карта, и друг компонент, който да представлява самия списък Нека започнем с преобразуване на този компонент в клас. Вместо функционален компонент, дефинирате клас със същото име и наследявате специален клас в React, React.Component. Наследяването на React.Component прави класа ви официален компонент на React. Ще научим две основни концепции в този пример: конструкто и ключовата дума this в класове. Всеки React компонент трябва да има функция за рендиране. Това е единствената необходима функция. Тя връща виртуалното DOM описание на вашия компонент. Създаваме истанции на класа и всяка получава своите свойства и състояние. Заглавието трябва да стане this.props.title. Създаваме класов компонент Card, който наследява React.Component, във функцията за рендиране връщаме div placeholder със клас github-профил.

Резултатът може да се провери на <https://jscomplete.com/playground/rgs2.2> .

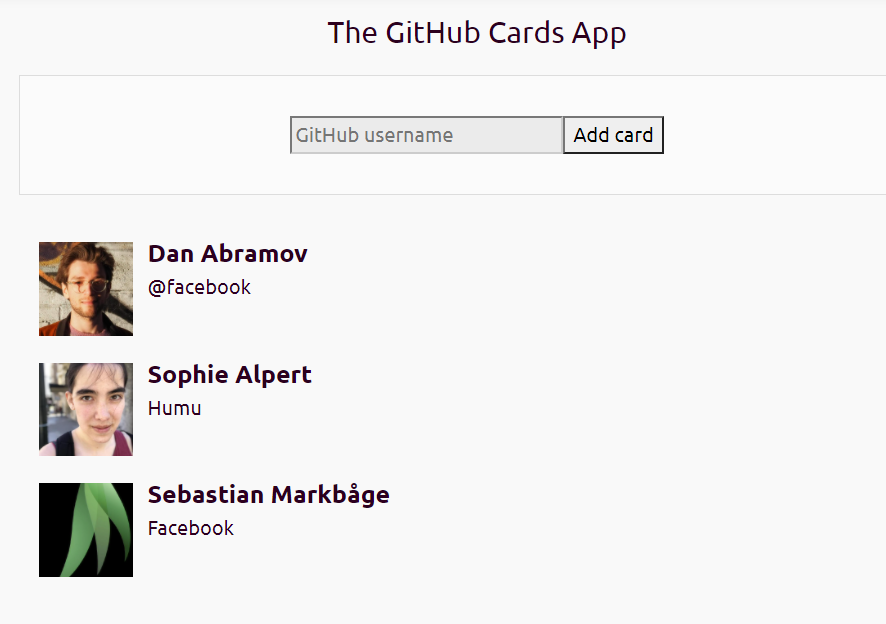
***Работа с данни***

Под връзката <https://jscomplete.com/playground/rgs2.3> добавих временен масив testData, с който да работим. Тези данни бяха копирани от API на GitHub, така че след като приключим тестването на нашето приложение с тези фалшиви тестови данни, ще работи с реалните данни на API. URL адресът, който използвах за копиране на данните от GitHub, е api.github.com/users след това поставяте произволно потребителско име от GitHub и ще видите данните като JSON обект. Тъй като искаме да изобразим множество карти, се нуждаем от друг компонент, който да държи различните карти. Ще наречем този нов компонент CardList. Възможно е да управляваме състоянието на приложението в този компонент. Може да го създадем като функционален компонент. Функционалният компонент CardList ще получи стандартния аргумент за свойствата връща div, който ще съдържа нашия списък с карти. В този div ще изобразим компонент Card и ще трябва да променим компонента на приложението да изобрази CardList вместо Card. Нека поставим някои реални данни в компонента Card. За това ще използваме масива testData. Вътре в функцията за рендиране на компонента Card, ще създадем локална променлива. За изображението ще използваме profile.avatar\_url а за името се нуждаем от profile.name. А за името на компанията се нуждаем от profile.company. Компонента Card ще получава данните чрез свойствата. Един от начините да направите това е да вземете обекта, който държи свойствата и да го разнесете вътре в елемента на компонента Card.  
РЕЗУЛТАТ: <https://jscomplete.com/playground/rgs2.4> .



**Инициализиране и прочитане обекта на състояние (State Object)**

За да вземем евентуално въвеждане от потребителя, можем да използваме прост HTML формуляр с вход и бутон. Нека създадем нов React компонент: Form, който наследява React.Component. Той се нуждае от функция за рендиране и трябва да върне някакъв DOM. Използваме входен елемент за да му дадем placeholder + бутон за добавяне на нова карта. За да се покаже този компонент в браузъра, трябва да го включим някъде в това, което изобразяваме. Подходящо мястно на компонента Form е в компонента на приложението на най-високо ниво. Компонентът App ще обработва връзката между компонента CardList и компонента Form. Масивът от данни, движещ кода, който имаме до момента, все още е глобална променлива. Компонентите трябва да избягват четенето на глобални променливи. Вместо да го четем директно от компонента CardList, нека го направим част от компонента App и да накараме компонента CardList да го получи като свойтсво. Сега в компонента CardList, вместо testData директно, трябва да използваме props.profiles. Компонентът Form ще трябва да добави запис към този масив. За да позволим както CardList, така и компонентът Form да имат достъп до масива на профилите, трябва да го поставим в състоянието на самия компонент на приложението. За да инициализираме обект на състояние за компонента App, трябва да добавим конструктор. Този специален метод получава и свойствата на инстанцията. Трябва да извика JavaScript супер метода, който е необходим за зачитане на връзката между класа на приложението и класа, от който се простира React.Component. React ще се оплаче, ако не го направите.  
Очакван резултат: <https://jscomplete.com/playground/rgs2.5>



**Вземане на данни от потребителя**

За да вземем данни от потребителя, трябва да дефинираме манипулатор на събития в React UI. На бутона „Добавяне на карта“ можем да дефинираме събитие onClick, но предпочитам да го обработим със събитие onSubmit за самия елемент на HTML формуляр. Използвайки onSubmit, можете да използвате функциите за подаване на естествени формуляри. Например можете да направите това въвеждане задължително и събитието onSubmit ще отчете това в браузъра. Нека дефинираме нова функция handleSubmit**.** Трябва да предотвратим поведението по подразбиране за подаване на формуляр, като използваме event.preventDefault. Това е важно, когато работите с формуляри, защото ако изпратите формуляра, страницата ви ще се обнови. Нуждаем се от събитие onChange, за да може DOM да каже на React, че нещо се е променило в този вход и трябва да го отразите и в потребителския интерфейс. Ще използваме вградена функция за събитието onChange. Тази функция получава обекта на събитието като свой аргумент и трябва да промени стойността на елемента на състоянието userName в това, което е въведено в текстовото поле. За целта в компонента използваме this.setState и му предаваме обект, който има новото състояние. В този случай трябва да го предадем на username като вземете стойността, която потребителят е въвел директно от DOM, като използваме event.target.value. Това е синтаксисът, който трябва да използваме, за да променим състоянието на класов компонент. Резултатът може да видите тук: <https://jscomplete.com/playground/rgs2.6>

**AJAX**

Сега, след като можем да прочетем въведеното от потребителя въведено, ние сме готови да поискаме от GitHub API данни за един профил, използвайки този тип въвеждане. Можем да използваме родното извикване, налично в браузърите, за да осъществим повикване AJAX. Но тази детска площадка също има библиотеката на axios и axios е проста за използване тук. Така че във функцията handleSubmit всичко, което трябва да направим, е axios.get и след това да посочим крайната точка на API, където са данните, и това е входът на API, от който се нуждаем. Така че тук, ако трябва отново да извлечем информация за профила на Дан, просто използваме този синтаксис. Но тъй като искаме това да бъде динамична част, ще продължим и ще направим всичко това в низ от шаблон, ще отбележим обратни отметки тук и вътре в това ще можем да инжектираме стойността на потребителското име, което потребителят е въвел в полето , което е this.state.userName. Поставяме това тук. Добре, по този начин правите извикване на аксиос. Сега axios.get връща обещание. Така че, за да консумираме това обещание, можем да го изчакаме и това ще ни върне обект на отговор. За да накараме това чакащо повикване да работи, трябва да маркираме този handleSubmit като асинхронизиран. Сега можем да console.log обекта за отговор, нека продължим и тестваме това. И така, тук, въведете всяко валидно потребителско име за GitHub, щракнете върху Добавяне на карта и handleSubmit ще отиде и ще извлече данните от профила на конкретното потребителско име и ще отпечата отговора. Сега този отговор има атрибут data, който съдържа действителната информация за профила от GitHub. Така че това, от което се нуждаем тук, е response.data. Това е просто начинът, по който работи аксиосът. Той връща обект на отговор и след това атрибут на данни за този обект на отговор, който има JSON данни, анализирани и готови за нас. Сега всичко, което трябва да направим по-нататък, е да добавим този обект от данни, който получаваме от API, към масива с профили, който имаме в състоянието на приложението, и когато го направим, реакцията ще направи повторно изобразяване на компонента на приложението и ще покаже новия GitHub профил . Този компонент на формуляр обаче не може да осъществи директен достъп до елементите на състоянието на профилите тук, защото реагиращите компоненти имат еднопосочен поток от данни и компонентът не може да промени състоянието на своя родител. Но не забравяйте, че компонентът на App може да предава свойства на своя дъщерен компонент на Form и тези свойства могат да бъдат прости примитивни стойности или препратки към функции.

**Обобщение**

Изградихме няколко прости React компонента:

1) Card, за да визуализира информация за профил в GitHub,

2) CardList, за да преобразува масив от записи в масив от компоненти на Card

3) компонент форма за четене на данни от потребителя

4) компонент на приложение за управление на връзката между всички

# Създаване на локална среда за разработка

Създаването на среда за разработка не е сред най-забавните неща. Трябва да накарате много различни инструменти да работят заедно. Инструментите имат различни APIта и всеки инструмент ще трябва да бъде конфигуриран. Възможно е да срещнете проблеми като различни версии да не са съвместими и да не работят заедно. Цялата среда може внезапно да спре да работи след определено надграждане на инструментите и ще трябва да отделите известно време за отстраняване на грешки. Освен това средата за разработка се различава от продуктивната среда, което означава, че това, което работи за вас в разработката, може да не работи в производството. За щастие има някои инструменти на високо ниво, които можете да използвате, за да избегнете някои от кошмарите за работа със среди.

# Create React App

Проектът Create React App може да ви помогне да създадете локална среда за разработка на React с помощта на една команда. Не е нужно да научавате нови API или да поддържате нови конфигурации. За да имате локална среда за разработка, имате нужда от Node. Create React App е само npm пакет, а npm е Node Package Manager. Според документацията на Реакт ( <https://reactjs.org/docs/create-a-new-react-app.html> ) правилният начин за стартиране на ново приложение е:

**npx create-react-app stargame**

Инсталацията на пакетите ще отнеме около 2/3 минути.

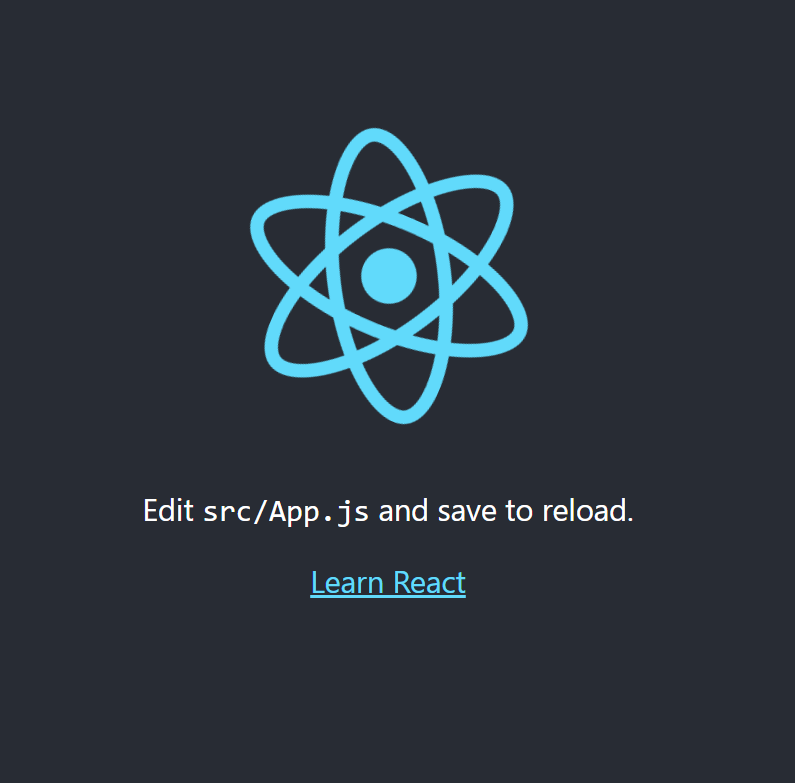
Влизаме в директорията на приложението:

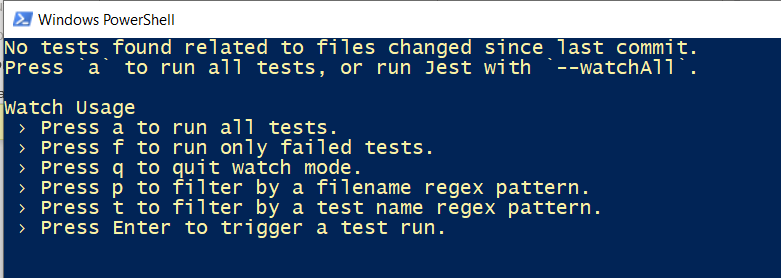
cd .\stargame\ или просто **cd stargame**.

Стартираме чисто новото ни приложение с:

**yarn start**

Очакваният резултат е този:

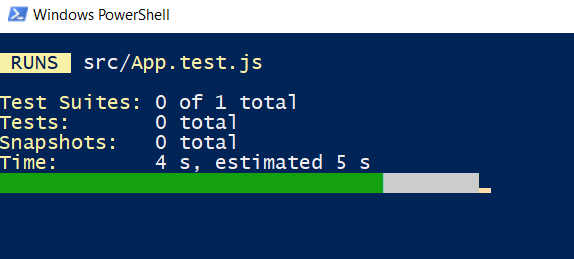


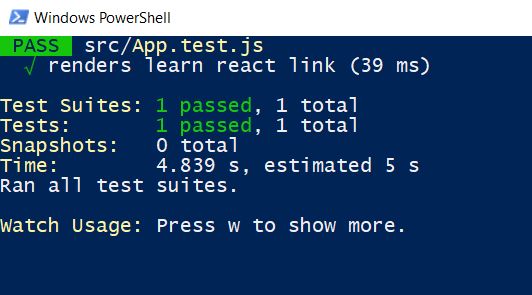
Прекъсваме процеса чрез командата от клавиши **Ctrl+C**, за да изпълним компонентните тестове:

**yarn test**

След това ще получиш следното съобщение:

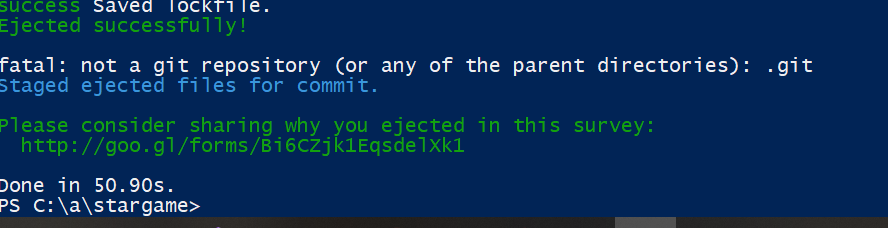
Въвеждаш **a** в конзолата.





Отново прекъсваме процеса. (Ctrl+C)

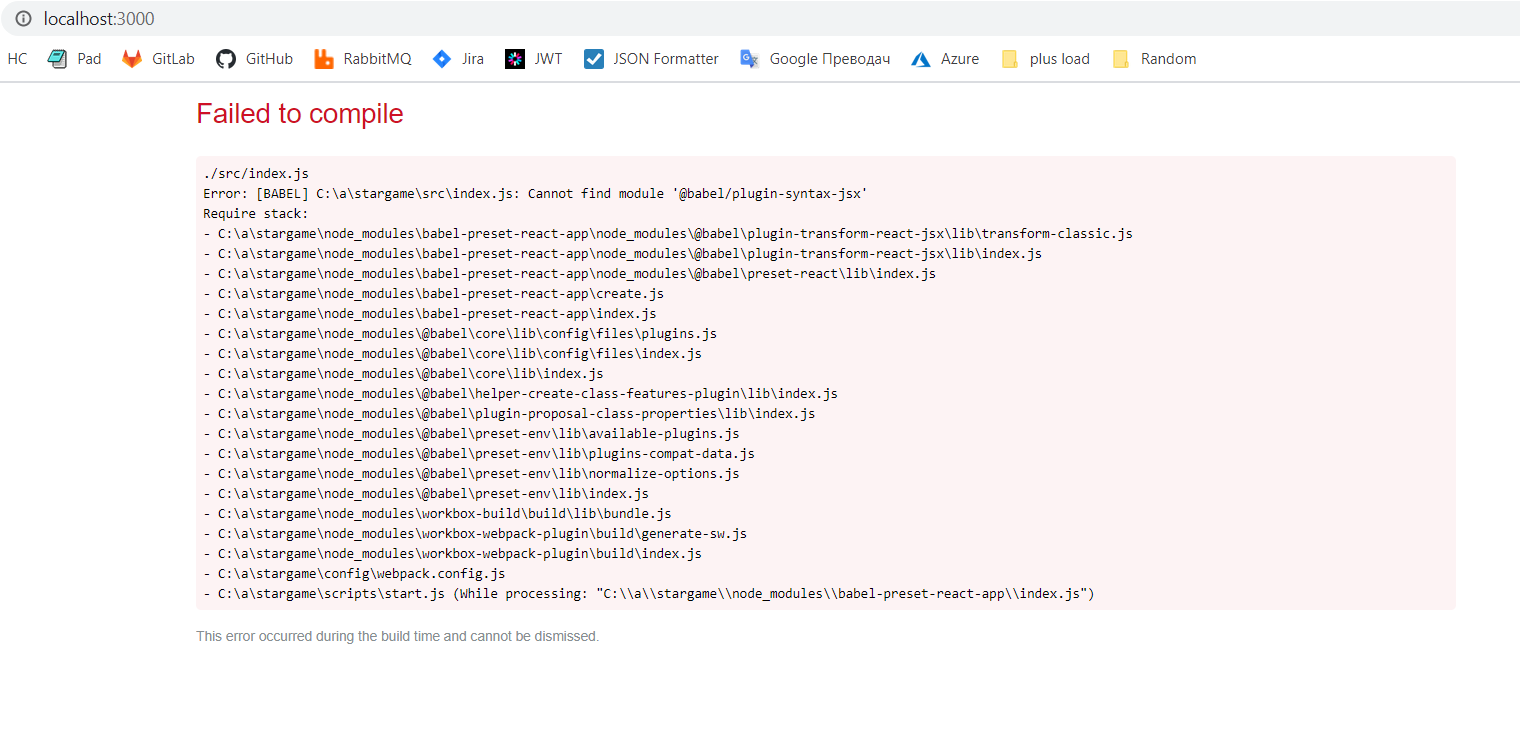
Изпълняваме командата :

**yarn eject**

С очакван резултат:

След това се уверете, че всичко е инсталирано правилно. (**yarn start**)

Възможно е, да се появи някой проблем като следния:



Грешката е добре описана. В случая е необходимо да изпълним:

**yarn add @babel/plugin-transform-react-jsx**

сададас