Документация

По разработке и использованию веб приложения, мобильного приложения, а также серверной части

Тема приложения: «Игра угадай покемона»

Логачева С.А.

Чеперегина А.С.

Санкт-Петербург

2023

Оглавление

[Front разработка 2](#_Toc153561732)

[Общее описание принципа работы программы 2](#_Toc153561733)

[Описание построения кода 4](#_Toc153561734)

[Back разработка 7](#_Toc153561735)

[Описание построения кода 7](#_Toc153561736)

[Описание базы данных 9](#_Toc153561737)

# Front разработка

## Общее описание принципа работы программы

Используемые технологии: React

Задумка: написать игру про покемонов.

Правила игры: пользователя будут выводится вопросы на карточке (рисунок 1.1). На каждой такой карточке есть название покемона вверху, ниже 4 картинки, одна из которых соответствует названию.

Задача: выбрать правильный ответ.

Вопрос-карточка включает в себя 4 картинки и название покемона, которого надо найти (рис.1.1)

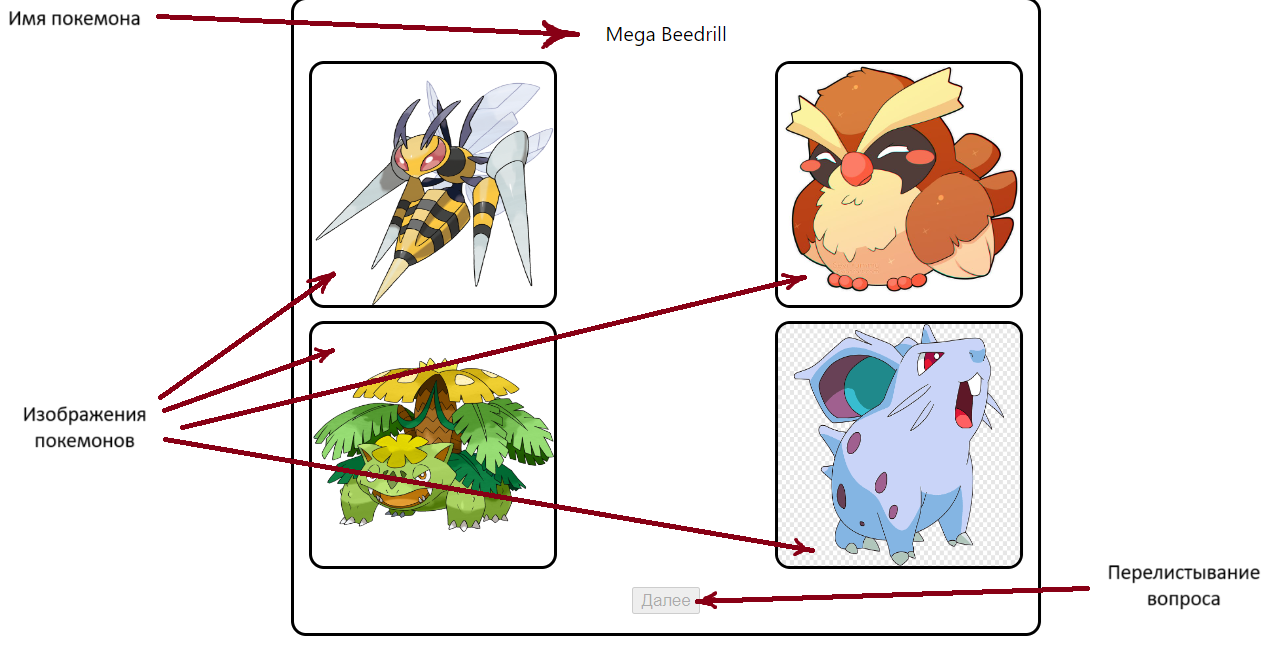


Рисунок 1.1. Описание расположения элементов в компоненте Card

Для перелистывания вопросов используется кнопка «Далее», которая также обозначена на рисунке 1.1. После выбора правильного ответа пользователю выскакивает alert с сообщением “Верно!” (рисунок 1.2).

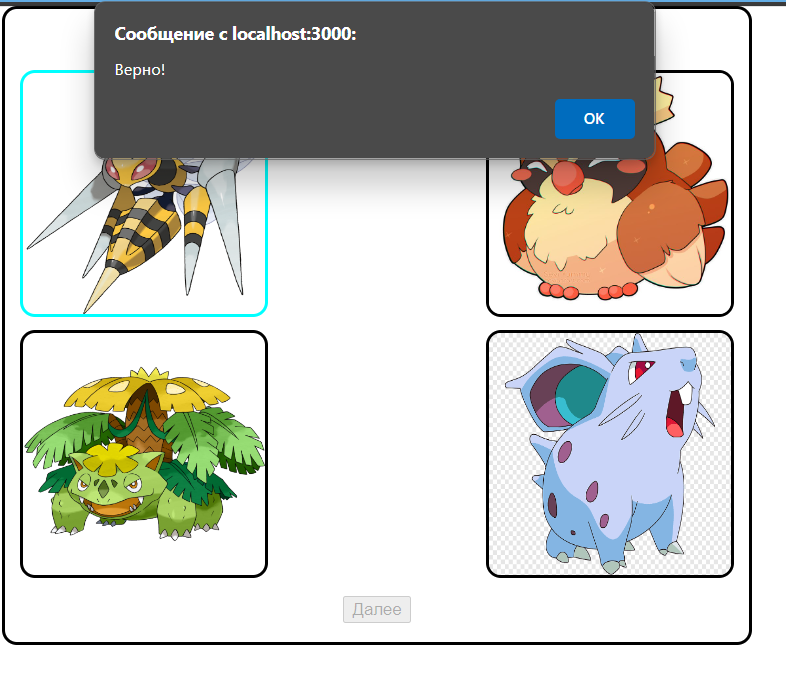


Рисунок 1.2. Сообщение пользователю о выборе правильного ответа

После того, как пользователь нажимает «ок», выбранный вариант становится зелёным, и кнопка «Далее» становится активной, как показано на рисунке 1.3.

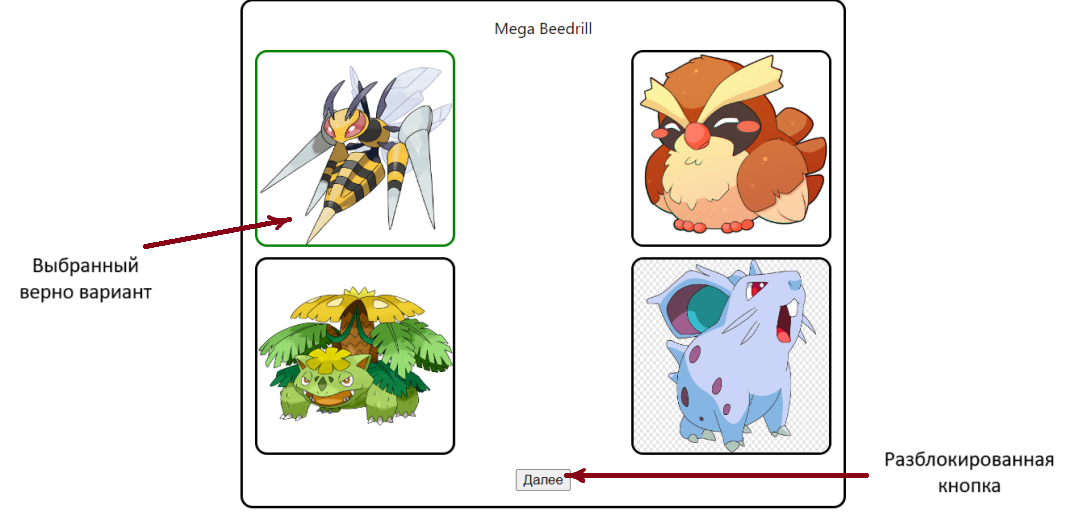


Рисунок 1.3. Поведение программы после выбранного правильного ответа.

## Описание построения кода

Было создано два компонента App и Card. App – стандартный компонент, который появляется автоматически при создании react приложения.

Компонент Card содержит в себе 4 картинки, название покемона, которого надо найти и кнопку «Далее», которая отвечает за перелистывание (рисунок 1).

Для генерации этого компонента, необходимо в пропсах передать структуру вопроса, а также передать функцию смены вопроса (будет описано далее).

Структура вопроса выглядит следующим образом:

{

currectAnswer: 1,

title: “Pikachu”,

data: [

{id: 1, src: “pikachu.png”},

{id: 5, src: “bird.png”},

{id: 3, src: “cat.png”},

{id: 4, src: “dog.png”}

]

}

CurrectAnswer — содержит id правильного ответа из массива data. Title — имя покемона, которого надо найти. Data — массив картинок с уникальными идентификаторами.

Таким образом, верстка для карточки-вопроса с вышеизложенной структурой выглядит, как показано на рисунке 1.4.

 <div className="card">

            <label>{question?.title}</label>

            <div className="container">

                {question?.data?.map((elem) =>

                    <img key={elem.id} id={elem.id} src={elem.src} onClick={handleClick} className={failClick.id == elem.id ? "fail" : currentClick.id==elem.id ? "correct" : ""}></img>

                )}

            </div>

            <button disabled={next} onClick={handleClickNext}>Далее</button>

        </div>

Рисунок .1.4. Верстка компонента Card

Следующим этапом необходимо прописать функцию обработки нажатия на картинки. Для этого напишем функцию handleClick. Её задача будет заключаться в том, что нужно проверить, является ли данная нажатая картинка правильной или нет. Также необходимо подсветить пользователю, что картинка выбрана или нет. Для этого будем использовать css классы fail и correct. Чтобы отслеживать и запоминать уже нажатые картинки воспользуемся хуком reacta UseState для правильного нажатия и для неправильного. Для отслеживания активности кнопки также будем использовать useState для переменной next. Вышеописанное представлено на рисунке 1.5.

    const [failClick, setFailClick] = useState({ id: null })

    const [currentClick, setCurrentClick] = useState({id: null})

Рисунок 1.5. Использование UseState.

Вернемся к функции handleClick. Как было упомянуто выше – её задача проверять и запоминать в useState необходимые параметры. Код данной функции показан на рисунке 1.6.

    const handleClick = (elem) => {

        const cur\_id = elem.currentTarget.getAttribute("id");

        if (cur\_id == question?.correctAnswer) {

            alert("Верно!")

            setCurrentClick((prevState) => ({

                ...prevState,

                id: cur\_id,

            }));

            setFailClick({ id: null })

            setNext(false)

        }

        else {

            setFailClick((prevState) => ({

                ...prevState,

                id: cur\_id,

            }));

        }

    }

Рисунок 1.6. Код функции handleClick для обработки нажатия на картинку

Следующая задача — это при нажатии на кнопку «далее», чтобы появлялся новый вопрос. Для этого как раз и надо было передать в пропсах функцию для перелистывания. Сама функция реализована в компоненте App. Её код представлен на рисунке 1.7.

  const getQuestion = async() =>{

    await axios.get(`${apiUrl}/getQuestion`).then((resp) => {

      console.log(resp)

      setQuestion(resp.data)

     });

  }

Рисунок 1.7. Функция перелистывания.

В этой функции происходит запрос на сервер. Адрес сервера хранится в переменной apiUrl. После получения результата в useState для question записывается ответ, который приходит с сервера. Далее эти данные уходят в пропсы к компоненту Card. Основное свойство реакта заключается в том, что при изменении любого состояния, компонент перерисовывается. Таким образом, App будет перерисован, а соответственно и Card с новыми пропсами тоже, поскольку вызывается в первом компоненте. apiUrl был взят http://192.168.0.104:5000, потому что на этом адресе поднимается сервер (по его конфигу).

При открытии веб приложения сразу формируется карточка-вопрос. Чтобы это сделать, необходимо использовать хук useEffect, в зависимость которых не будет ничего входить (это означает, что вызов этого useEffect будет происходить единожды при рендере компонента). Код вызова useEffect представлен на рисунке 1.8.

  useEffect(() => {

    getQuestion();

  }, [])

Рисунок 1.8. Вызов генерации вопроса.

Ранее упоминали о функции для перелистывания

Мы описали все функции, вызываемые в данном приложении. Осталось лишь верстка для компонента App, которая представлена на рисунке 1.9.

    <div className="App">

      <Card question={question} changeIndex={getQuestion} />

    </div>

Рисунок 1.9. Верстка для компонента App.

# Back разработка

## Описание построения кода

Используемые технологии: NodeJS

Для поднятия сервера будем использовать библиотеку express. Для подключения необходимо использовать следующую команду: const express = require('express'). Порт, на котором будет разворачиваться сервер, задается с помощью переменной port=5000. Далее происходит инициализация сервера на необходимом порту. Вышеописанные действия представлены на рисунке 2.1.

const express = require('express');

const port = 5000;

const app = express();

app.listen(port, () => {

    console.log(`Server started on port ${port}`);

});

Рисунок 2.1 Инициализация сервера на порту 5000

Для работы с базой будем использовать библиотеку mysql. Для подключения к ней необходимо создать конфиг, который бы хранил в себе имя пользователя, пароль, хост и подключаемую схему данных. Далее необходимо инициализировать соединение. Вышеописанные действия представлены на рисунке 2.2.

const mysql = require('mysql');

// Set database connection credentials

const config = {

    host: 'localhost',

    user: 'root',

    password: '12TKL3212tkl32',

    database: 'finalProject',

};

const pool = mysql.createPool(config);

Рисунок 2.2. Инициализация соединения с базой

Далее для работы с веб приложением необходимо разобраться с политикой Cors. Для этого была установлена библиотека cors, также был создан так называемый белый лист. Белый лист — это массив адресов хостов, который не будут блокироваться. Поскольку по дефолту реакт приложение запускается на 3000 порту, то в белый лист был записан адрес http://localhost:3000. Далее с использованием вышеупомянутой библиотеки можно обойти эту политику cors. Обойти получится только тем хостам, которые прописаны в белом листе. Для вышеописанных действий был написан код, представленный на рисунке 2.3.

// use cors for connect back and front

const cors = require("cors")

const whitelist = ["http://localhost:3000"]

const corsOptions = {

    origin: function (origin, callback) {

        if (!origin || whitelist.indexOf(origin) !== -1) {

            callback(null, true)

        } else {

            callback(new Error("Not allowed by CORS"))

        }

    },

    credentials: true,

}

app.use(cors(corsOptions))

Рисунок 2.3. Обхождение Cors для взаимодействия React приложения и сервера.

Далее необходимо создать endpoint, который был генерировал структуру вопроса, описанную в главе 1 на основе данных в базе.

Поскольку вопросы должны быть рандомными, то при запуске сервера будем использовать дополнительную переменную maxCount, которая будет отвечать за максимальный id в таблице с покемонами.

В самом начале запуска сервера получим количество покемонов в таблице – это и будет maxCount. Сделать это можно с помощью кода, представленного на рисунке 2.4.

let maxCount = null;

pool.query(`SELECT count(\*) as count FROM POKEMONS`, (error, result) => {

    maxCount = result[0].count;

})

Рисунок 2.4. Получения количества покемонов в таблице

После получения данных для maxCount можем спокойно использовать рандом. С помощью рандома получаем id покемона, на основе которого будет строится весь вопрос. С помощью запроса к базе берем покемона, id которого вычислили в рандома. Полученные результаты после выполнения запроса — correctAnswer, title. Также введем дополнительную переменную data, в которую в последствии будем записывать оставшиеся 3 картинки (не 4, потому что один слот уже занят правильной картинкой). Вышеописанные действия представлены на рисунке 2.5.

const min = 1;

const rand = Math.floor(Math.random() \* (maxCount - min + 1)) + min;

pool.query(`SELECT \*  FROM POKEMONS where id=${rand}`, (error, result) => {

            if (error) { console.log(error) }

            else {

                let question = { correctAnswer: result[0]?.id,

                    title: result[0]?.name

                }

                data = []

                data.push({id: result[0]?.id, src: result[0]?.img})

            }

Рисунок 2.5. Получение покемона из базы по выбранному id

Далее необходимо выбрать 3 случайные картинки из таблицы с покемонами. Для этого нужно выбрать всех покемонов, у которых id не соответствует ранее выбранному покемону (чтобы не было повторений и не оказалось два правильных варианта, вместо одного). После выборки мы получим массив всех записей из таблицы, кроме одной записи. Из этого массива нужно выбрать случайным образом 3 индекса (причем таким образом, чтобы индексы не повторялись), и добавить их в ранее созданную переменную data. После этого полученную структуру можно отправлять клиенту. Вышеописанные действия представлены на рисунке 2.6.

 pool.query(`SELECT \*  FROM POKEMONS where id!=${rand}`, (error, result) => {

                    if (error) {

                        console.log(error)

                        response.send({ status: "error", e: error })

                    }

                    else {

                        let uniqueNumbers = [];

                        // генерируем 4 уникальных случайных числа от 1 до 10

                        while (uniqueNumbers.length < 3) {

                            let randomNumber = Math.floor(Math.random() \* result.length-1) + 1;

                            if (!uniqueNumbers.includes(randomNumber)) {

                                uniqueNumbers.push(randomNumber);

                            }

                        }

                        for (let i=0;i<3;i++){

                            data.push({ id: result[uniqueNumbers[i]].id, src: result[uniqueNumbers[i]].img })

                        }

                        question.data = data;

                        response.send(question)

                    }

                })

Рисунок 2.6. Выбор случайных картинок из базы данных.

## Описание базы данных

Поскольку игра связана с покемонами, то была создана таблица pokemons, которая содержит в себе id — уникальный идентификатор, name — имя покемона, img — ссылка на картинку в просторах Интернета.

# Инструкция по запуску

Для запуска сервера необходимо перейти в папку /server и в командной строке по очереди прописать две команды:

npm install

npm start

После успешного запуска в терминале будет выведено сообщение «Server started on port 5000». Если необходимо изменить порт, на котором будет работать порт, необходимо перейти в папку server/config/default.json. В данном файле будет json объект, содержащий поля port и whitelist. Их можно настроить под себя.

Для запуска react приложения, необходимо перейти в папку /front и по очереди прописать две команды:

npm install

npm start

Если был изменен адрес сервера, то необходимо открыть файл front/src/App.js и в переменной apiUrl поменять адрес сервера на нужный.