**Введение в Drag and Drop**

Вы уже умеете работать с большинством пользовательских событий: кликом, наведением, изменением полей форм, отправкой формы и другими. С этими событиями могут быть связаны, например, загрузка локальных файлов с использованием File API или перетаскивание элементов интерфейса (Drag and Drop). О последнем мы расскажем в этой теме.

Drag and Drop, или dnd, встречается во многих проектах. Причём dnd может быть и основным способом взаимодействия с интерфейсом, и небольшой дополнительной функциональностью. Например, интерфейс Trello почти полностью построен на dnd. А возможность перетащить локальный файл изображения для установки нового аватара в какой-нибудь социальной сети — всего лишь небольшая функциональность dnd в огромном проекте.

Начнём изучение Drag and Drop с самых основ. Сначала вы узнаете о наборе событий, из которых состоит dnd. После — попрактикуетесь с более сложными способами перетаскивания элементов интерфейса. Всё это нужно для знакомства с одним из готовых решений — библиотекой react-dnd. Именно её вы будете использовать в своей проектной работе.

Пора переходить к первому уроку темы.

# Как работает Drag and Drop. Основы и термины

В этом уроке разберём самый простой способ перетаскивания элементов интерфейса. Мы рассмотрим механизм DnD в двух плоскостях: из каких пользовательских событий состоит перетаскивание и какие элементы в нём участвуют.

## Пользовательские события в Drag and Drop

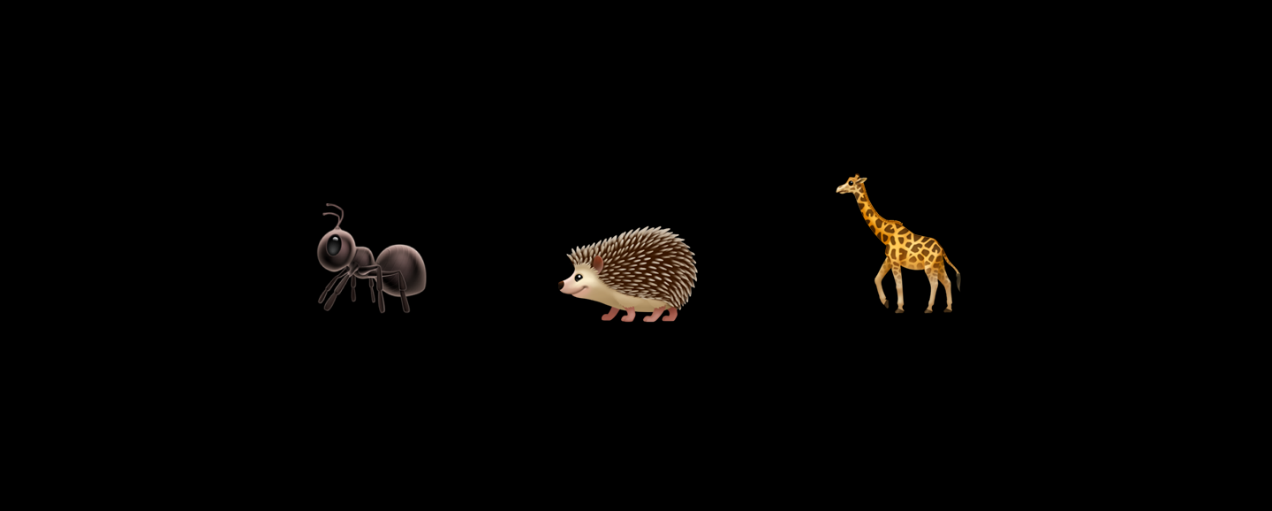
Перетаскивание — набор из трёх событий в самом простом варианте реализации. Сначала пользователь нажимает по элементу левой кнопкой мыши (ЛКМ). После этого при зажатой ЛКМ перемещает элемент по доступной области. Когда пользователю надоедает перетаскивать элемент — он отпускает ЛКМ, и элемент перестаёт реагировать на движение мыши.

Если перевести это на язык React, используются:

* onMouseDown — момент «захвата» элемента пользователем,
* onMouseMove — момент «перетаскивания» элемента,
* onMouseUp — момент «броска» элемента.

На каждом из этапов происходят вычисления и действия. В этом уроке мы разберём моменты «захвата», «перетаскивания» и немного — момент «броска».

Перетаскивать будем ёжика, жирафа и муравья:



Не задумывайтесь, почему мы выбрали именно этих животных. Возможно, это любимые животные автора

Напишем базовый компонент App, который будет рендерить список компонентов DraggableAnimal:

Скопировать кодJSX

*// app.js*

import React from 'react';

import DraggableAnimal from '../draggable-animal/draggable-animal'

import styles from './app.module.css';

const listImages = [

{

id: 1,

content: '🦔',

},

{

id: 2,

content: '🦒',

},

{

id: 3,

content: '🐜',

},

];

const App = () => {

const [elements, setElements] = React.useState([]);

React.useEffect(() => {

setElements([

...listImages

])

}, []);

return (

<section className={styles.element}>

{

elements.map((element) => <DraggableAnimal key={element.id} content={element.content}/>)

}

</section>

)

};

export default App;

Скопировать кодJSX

*// draggable-animal.js*

import React from 'react';

import styles from "./draggable-animal.module.css";

const DraggableAnimal = (props) => {

return (

<div

className={styles.animalElement}

>

{props.content}

</div>

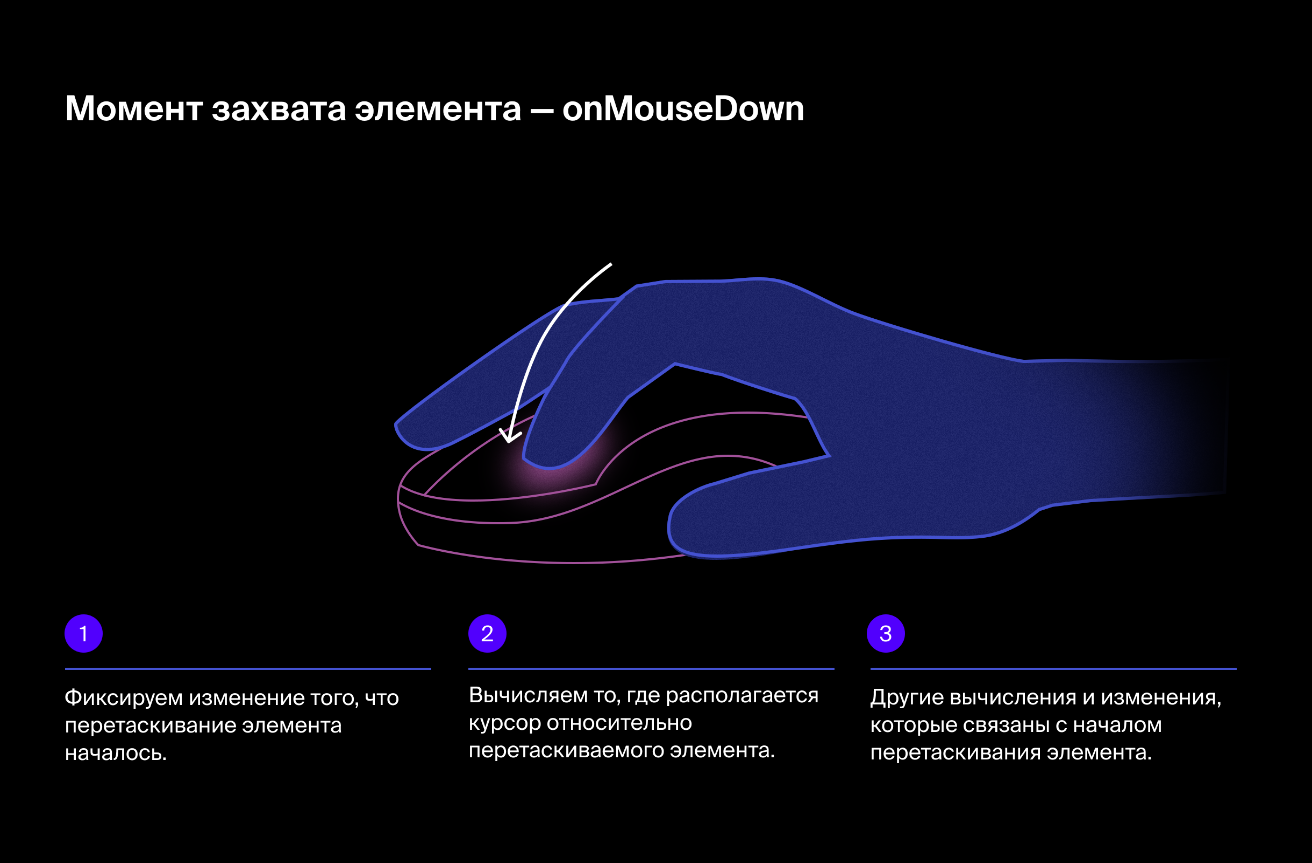
)

};

export default DraggableAnimal;

Когда базовая вёрстка готова, рассмотрим каждое из событий.

### Момент захвата элемента — onMouseDown



Когда пользователь зажимает ЛКМ на элементе, например на ёжике, нужно «разблокировать» возможность перетаскивать ёжика с помощью обработчика событий onMouseMove. Это важно, потому что без возможности блокировать перемещение ёжика по onMouseMove элемент будет прилипать к курсору при наведении. Нам же нужно разблокировать «захват» только при зажатой ЛКМ.

Самый простой вариант блокировки перемещения курсора — булевое состояние. Его можно назвать isElementDragging с начальным значением false. Когда пользователь нажимает ЛКМ, состояние isElementDragging изменяется на true.

В момент срабатывания события onMouseDown нужно вычислить конкретную точку элемента (в нашем случае — ёжика), на которой пользователь зажал ЛКМ. Если этого не сделать, ёжик будет дёргаться в момент начала перетаскивания и прилипать к курсору левым верхним краем:

Для решения проблемы нужно вернуться к нативному JS и выяснить, где взять данные для вычислений и что они значат.

Для вычислений нужны эти данные:

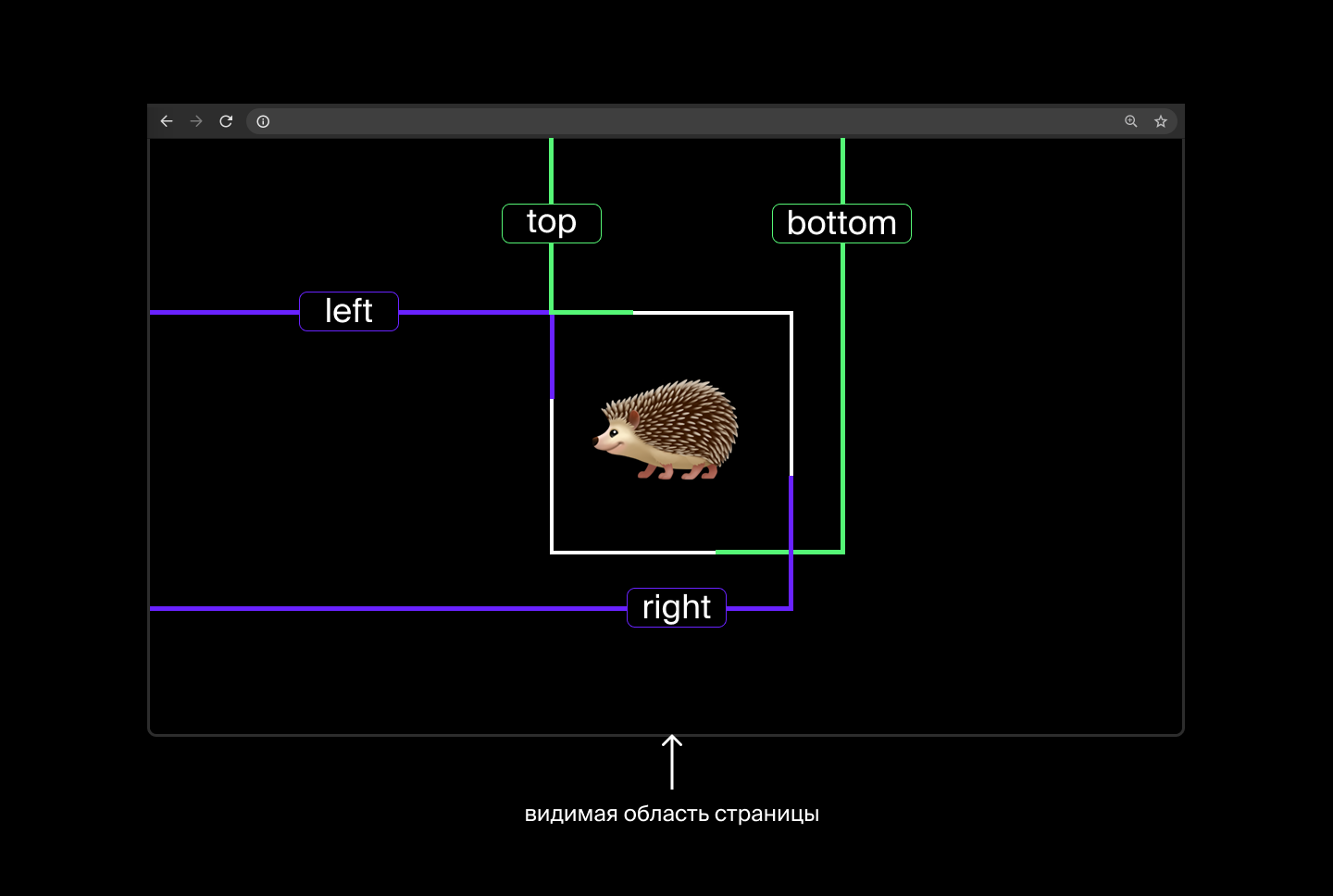
* вся видимая область страницы, в которой находится ёжик,
* расположение ёжика относительно видимой области.

Высчитывать координаты ёжика будем относительно viewport — видимой в конкретный момент времени части страницы. Координаты рассчитываются от левого верхнего угла. В самом верхнем левом углу они равны 0, 0.

В JS есть набор свойств с координатами и размерами элемента — объект DOMRect. И метод getBoundingClientRect, который возвращает свойства с координатами элемента относительно видимой части экрана:

* left — левый край элемента относительно левой видимой части окна страницы,
* top — верхний край элемента относительно верхней видимой части окна страницы,
* right — правый край элемента относительно левой видимой части окна страницы,
* bottom — нижний край элемента относительно верхней видимой части окна страницы.

Представим всё это в виде схемы:



Сначала вы изучаете Redux, а потом рассматриваете схемы с ёжиком. Такой вот интенсив

Теперь, когда координаты ёжика могут быть вычислены, надо найти конкретную часть ёжика, по которой кликнул пользователь. За это отвечают свойства объекта SyntheticEvent — clientX и clientY. Свойства возвращают текущие координаты курсора относительно левой и верхней видимой части экрана. Обратите внимание на схожие свойства: pageX и pageY, screenX и screenY. Они нам не подойдут, потому как возвращают координаты относительно всего документа и всего экрана пользователя соответственно.

Перейдём к написанию самого кода. Добавим первый слушатель и состояние для компонента DraggableAnimal:

Скопировать кодJSX

*// draggable-animal.js*

import React from 'react';

import styles from "./draggable-animal.module.css";

const DraggableAnimal = (props) => {

const [isElementDragging, setElementDrag] = React.useState(false);

const [cursorPosition, setCursorPosition] = React.useState({});

const handleMouseDown = (e) => {

setElementDrag(true);

};

return (

<div

onMouseDown={handleMouseDown}

className={styles.animalElement}

>

{props.content}

</div>

)

};

export default DraggableAnimal;

Мы объявили два состояния: isElementDragging и cursorPosition.

Состояние isElementDragging отвечает за блокирование возможности перетаскивать элемент. Мы будем изменять состояние в обработчиках событий onMouseDown и onMouseUp.

Состояние cursorPosition — объект, где мы будем сохранять положение курсора пользователя внутри перетаскиваемого элемента. В объекте будет два свойства — x и y — по этим осям мы будем отслеживать положение курсора.

Координата по оси X равна разнице между e.clientX (расположение курсора пользователя) и e.target.getBoundingClientRect().left (расположение элемента). Обратите внимание, что e.clientX всегда больше или равен e.target.getBoundingClientRect().left. В противном случае сам обработчик события onMouseDown по элементу не запустится. Аналогичным образом рассчитывается значение по оси Y.

Запишем все вычисления в состояние cursorPosition:

Скопировать кодJSX

const handleMouseDown = (e) => {

setElementDrag(true);

setCursorPosition({

...cursorPosition,

x: e.clientX - e.target.getBoundingClientRect().left,

y: e.clientY - e.target.getBoundingClientRect().top

});

};

Это основные вычисления, которые нужны для перетаскивания ёжика. Вычислений может быть намного больше, именно поэтому для более сложных решений мы будем выбирать готовые библиотеки.

Теперь перейдём к работе с событием onMouseMove.

### Момент перемещения элемента — onMouseMove



Здесь всё просто. Когда курсор пользователя начинает перемещаться вместе с зажатой ЛКМ, нужно транслировать ёжику об изменяющихся координатах. Перед вычислениями следует проверять состояние isElementDragging.

Если значение false, нужно выходить из обработчика события. Если этого не сделать, перемещение элемента будет происходить спонтанно, при наведении на него.

Если значение isElementDragging равно true, можно перейти к вычислениям. Мы не будем обновлять состояние cursorPosition, а заведём новое — elementPosition. Всё дело в особенностях работы слушателя onMouseMove — он срабатывает только в случаях изменения положения курсора. Другими словами, с момента обработки слушателя событий onMouseDown до срабатывания onMouseMove значение clientX или clientY точно изменятся.

После вычислений нужно передавать их ёжику. Мы постоянно обновляем состояния, поэтому можно передавать вычисления внутри хука useEffect с использованием рефа.

Создадим реф и опишем новое состояние elementPosition в компоненте DraggableAnimal:

Скопировать кодJSX

*// draggable-animal.js*

import React from 'react';

import styles from "./draggable-animal.module.css";

const DraggableAnimal = (props) => {

const [isElementDragging, setElementDrag] = React.useState(false);

const [cursorPosition, setCursorPosition] = React.useState({});

const [elementPosition, setElementPosition] = React.useState({});

const elementRef = React.useRef();

const handleMouseDown = (e) => {

setElementDrag(true);

setCursorPosition({

...cursorPosition,

x: e.clientX - e.target.getBoundingClientRect().left,

y: e.clientY - e.target.getBoundingClientRect().top

});

};

return (

<div

ref={elementRef}

onMouseDown={handleMouseDown}

className={styles.animalElement}

>

{props.content}

</div>

)

};

export default DraggableAnimal;

Создадим обработчик событий handleMouseMove, в котором опишем все вычисления:

Скопировать кодJSX

const handleMouseMove = (e) => {

if (!isElementDragging) return;

// Отменим поведение по умолчанию и всплытие

e.stopPropagation();

e.preventDefault();

setElementPosition({

...elementPosition,

x: e.clientX - cursorPosition.x,

y: e.clientY - cursorPosition.y

});

};

Мы повторно высчитываем разницу между clientX и cursorPosition.x, и в этом месте исправляем поведение, при котором перетаскиваемый элемент прилипает левым верхним краем к курсору.

Опишем хук useEffect, где воспользуемся данными из состояния elementPosition и передадим их в атрибут style рефа:

Скопировать кодJSX

React.useEffect(() => {

if (elementRef.current) {

elementRef.current.style.transform = `translate(${elementPosition.x}px, ${elementPosition.y}px)`

}

}, [elementPosition]);

Мы используем transform, а не свойства left и top для рефа. Свойство transform более производительное и семантически лучше подходит для этой задачи. В конце добавим перетаскиваемому элементу слушатель onMouseMove:

Скопировать кодJSX

const DraggableAnimal = (props) => {

return (

<div

ref={elementRef}

onMouseMove={handleMouseMove}

onMouseDown={handleMouseDown}

className={styles.animalElement}

>

{props.content}

</div>

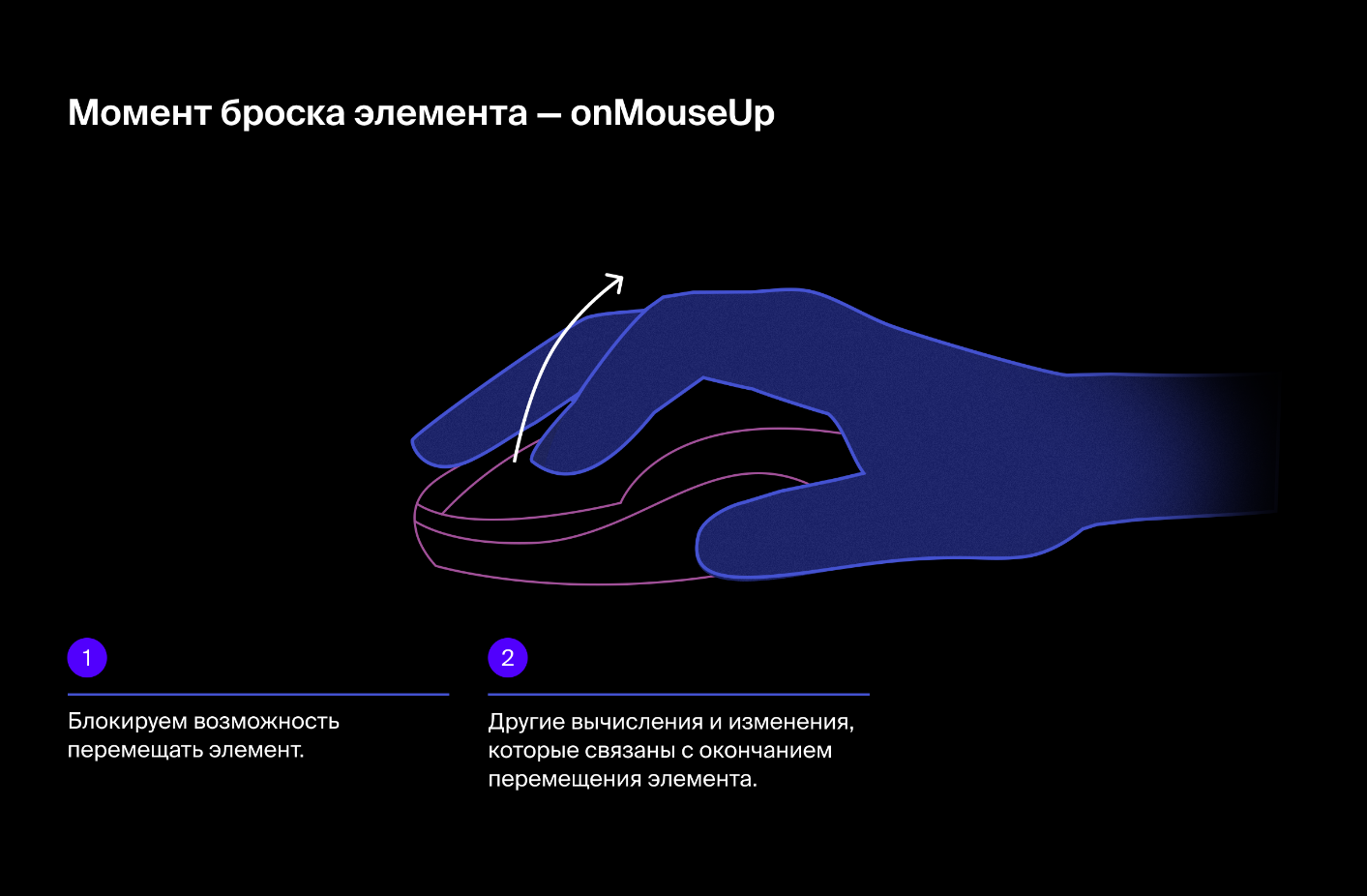
)

};

В итоге ёжик начнёт нормально ползать по экрану:

Перейдём к последнему событию — onMouseUp.

### Момент броска элемента — onMouseUp



Когда перетаскивание элемента завершается, достаточно просто заблокировать его дальнейшее перемещение. Поэтому мы изменим состояние isElementDragging на false. Так выглядит итоговый код компонента DraggableAnimal:

Скопировать кодJSX

const DraggableAnimal = (props) => {

const [isElementDragging, setElementDrag] = React.useState(false);

const [cursorPosition, setCursorPosition] = React.useState({});

const [elementPosition, setElementPosition] = React.useState({});

const elementRef = React.useRef();

React.useEffect(() => {

if (elementRef.current) {

elementRef.current.style.transform = `translate(${elementPosition.x}px, ${elementPosition.y}px)`

}

}, [elementPosition]);

const handleMouseDown = (e) => {

setElementDrag(true);

setCursorPosition({

...cursorPosition,

x: e.clientX - e.target.getBoundingClientRect().left,

y: e.clientY - e.target.getBoundingClientRect().top

});

};

const handleMouseMove = (e) => {

if (!isElementDragging) return;

e.stopPropagation();

e.preventDefault();

setElementPosition({

...elementPosition,

x: e.clientX - cursorPosition.x,

y: e.clientY - cursorPosition.y

});

};

const handleMouseUp = () => {

setElementDrag(false);

};

return (

<div

ref={elementRef}

onMouseMove={handleMouseMove}

onMouseDown={handleMouseDown}

onMouseUp={handleMouseUp}

className={styles.animalElement}

>

{props.content}

</div>

)

};

Пока этого достаточно. Но обычно обработка события onMouseUp содержит в себе логику перемещения элемента в новый контейнер.

## Элементы в Drag and Drop

Мы рассмотрели простой пример перетаскивания элемента. Теперь разберёмся с элементами, которые участвуют в этом процессе:

1. Контейнер — компонент, внутри которого может происходить перемещение элемента. В нашем случае это компонент App, размеры которого равнялись ширине и высоте окна браузера.
2. Исходный элемент — компонент, который мы перетаскиваем внутри контейнера. В нашем случае это ёжик.
3. Целевой элемент — компонент, в который мы перетаскиваем исходный элемент. Пока мы не добавили такой компонент, но сделаем это позже.

Почти все популярные DnD-библиотеки в React используют как минимум три этих элемента. Конечно, их может быть и больше. Более подробно поговорим о библиотеках через урок.

## Что ещё важно знать о Drag and Drop

Существует отдельный интерфейс Drag and Drop в браузерах. Вот основные события, из которых он состоит:

* ondragstart,
* ondrag,
* ondragend,
* ondrop.

Список можно дополнить и некоторыми другими событиями. Этот интерфейс по умолчанию предоставляет возможности, о которых мы говорили выше, но имеет и ряд ограничений, не позволяющих полноценно кастомизировать процесс перемещения элементов. С этим набором событий мы будем работать в следующих уроках.

Не забывайте и о том, что события onMouseDown, onMouseUp и onMouseMove не предназначены для работы с тачскрином смартфонов или планшетов. Для реализации DnD на таких устройствах используются touch-события. Их много, вот основные:

* onTouchStart — может быть использован аналогично onMouseDown,
* onTouchMove — используется аналогично onMouseMove,
* onTouchEnd — используется аналогично onMouseUp.

На этом введение в DnD подходит к концу. Ёжик двигается, и вам тоже пора двигаться к следующему уроку.

**Drag and Drop. Ближе к реальности**

Шутки кончились. В предыдущем уроке мы перетаскивали ёжика. Но простое перемещение ёжика по экрану вряд ли можно отнести к бизнес-логике настоящего приложения. Так что в этом уроке мы погрузимся в сложные концепции DnD. И будем перетаскивать муравьёв.

Мы разобрали принципы перетаскивания элементов с помощью обработки событий группы onMouse. Также неоднократно обращались к вычислениям: расположению курсора и перетаскиваемого элемента. Если немного усложнить задачу, вычислений станет ещё больше. Например, сделаем небольшой контейнер для животных и место, куда их нужно перетащить:



*«Откуда же появилась рыбка-колючка?», — спросите вы. «Понятия не имеем», — ответим мы*

Муравьёв нужно перетащить из левой области в правую. Если делать эту функциональность на событиях onMouse, то простое перетаскивание обрастёт огромным количеством вычислений. Нам потребуется:

* возвращать элемент в исходную позицию,
* определять положение перемещаемого элемента и контейнера, в который этот элемент переносится,
* обрабатывать все коллизии, которые возникают при «броске» элемента на краю целевого контейнера.

Использование таких вычислений в нагруженном бизнес-логикой приложении приводит к росту компонентов. А сам механизм перетаскивания элементов — лишь визуализация процесса, который может совершить пользователь.

Чтобы не делать лишних вычислений там, где это не нужно — используем другой инструмент. Это события группы onDrag. А уже в следующем уроке изучим библиотеку для перетаскивания элементов.

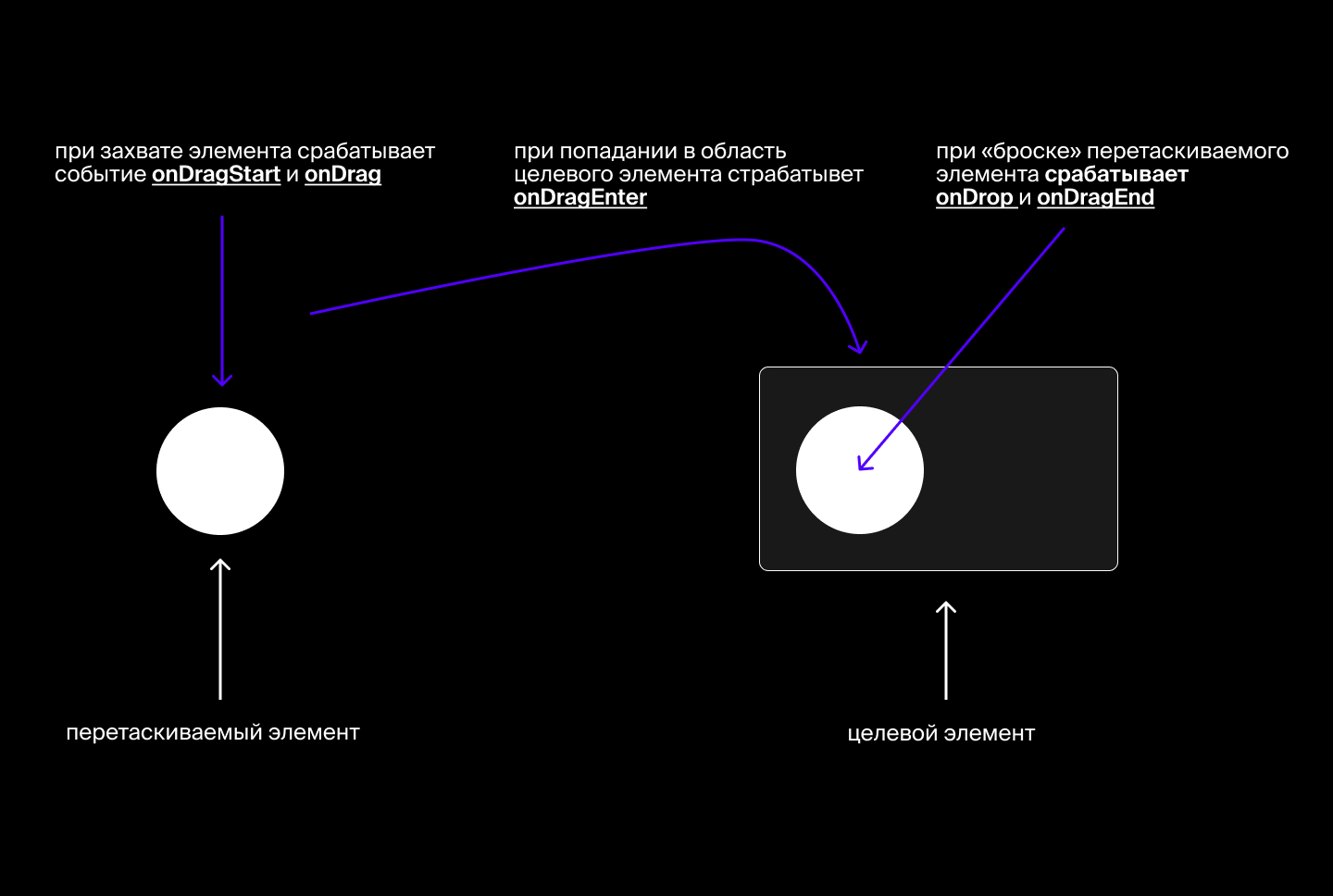
Вы уже знаете три основных элемента, которые участвуют в процессе перетаскивания и события DnD:

* onDrag — событие возникает в начале перетаскивания элемента. Слушатель добавляется тому элементу, который перетаскивают.
* onDrop — событие возникает в момент «броска» перетаскиваемого элемента в целевой элемент. Слушатель добавляется тому элементу, в который перетаскивают.
* onDragOver — событие возникает в момент, когда перетаскиваемый элемент находится над целевым. Обладает поведением по умолчанию, которое запрещает перетаскивание. Слушатель добавляется тому элементу, в который перетаскивают.

Это обязательные события, без которых не получится работать с интерфейсом DnD.

Также для перетаскиваемого элемента существуют события onDragStart и onDragEnd. События возникают в момент начала и завершения перетаскивания элемента. Их можно использовать для дополнительной визуализации перетаскивания, например, чтобы выделить перетаскиваемый элемент или воспроизвести анимацию.

Для целевого элемента есть события onDragEnter и onDragLeave. События возникают в момент входа или выхода перетаскиваемого элемента в область целевого. Эти события можно использовать, чтобы изменить интерфейс целевого элемента.



*Цепочка событий при перетаскивании элементов интерфейса*

С применением встроенного интерфейса для перетаскивания мы сильно облегчаем процесс разработки, ведь Drag and Drop вычислит всё за нас. А мы тем временем можем сконцентрироваться, например, на взаимодействии с хранилищем и его отладке.

Попробуем переписать код из предыдущего урока. Воспользуемся тремя компонентами:

* DragAndDropContainer — в нём хранятся состояние и обработчики событий перетаскиваемого и целевого элемента;
* DraggableAnimal — перетаскиваемый элемент;
* DropTarget — целевой элемент для DraggableAnimal.

Опишем состояние в контейнере:

Скопировать кодJSX

const DragAndDropContainer = () => {

const [elements, setElements] = React.useState([]);

const [draggedElements, setDraggedElements] = React.useState([]);

const [draggedElement, setDraggedElement] = React.useState({});

React.useEffect(() => {

setElements([

*// При монтировании компонента добавляем в elements захардкоженные данные*

...listImages

])

}, []);

return (

<article className={styles.container}>

</article>

)

};

В коде выше мы создали такие состояния:

* elements — исходный массив перетаскиваемых элементов;
* draggedElements — массив элементов, которые перетащили в целевой элемент;
* draggedElement — объект с перетаскиваемым элементом.

Теперь займёмся перетаскиваемыми элементами. По умолчанию для них надо обрабатывать событие onDrag. Когда начинается перетаскивание, нужно добавить в состояние draggedElement «захваченный» элемент. Для этого можно просто передать в обработчик конкретный элемент массива elements, на котором и сработал обработчик. Чтобы элемент вообще можно было перетаскивать, ему нужно добавить атрибут draggable. Только так браузер поймёт, что событие onDrag должно сработать.

Добавим слушатель onDrag и атрибут draggable:

Скопировать кодJSX

import React from 'react';

import styles from "./draggable-animal.module.css";

const DraggableAnimal = (props) => {

const { data, onDragHandler } = props;

return (

<div

className={styles.animalElement}

draggable

onDrag={(e) => onDragHandler(e, data)}

>

{data.content}

</div>

);

};

export default DraggableAnimal

Сам обработчик событий будет приходить в компонент DraggableAnimal из контейнера DragAndDropContainer:

Скопировать кодJSX

const DragAndDropContainer = () => {

const [elements, setElements] = React.useState([]);

const [draggedElements, setDraggedElements] = React.useState([]);

const [draggedElement, setDraggedElement] = React.useState({});

React.useEffect(() => {

setElements([

*// При монтировании компонента добавляем в elements захардкоженные данные*

...listImages

])

}, []);

const handleDrag = (e, draggedElement) => {

e.preventDefault();

setDraggedElement(draggedElement);

};

return (

<article className={styles.container}>

<div className={styles.animals}>

{

elements.map(animal =>

<DraggableAnimal key={animal.id} data={animal} onDragHandler={handleDrag} />

)

}

</div>

</article>

)

};

Конечно, весь список элементов можно вынести в компонент DragList. Но чтобы не практиковать props drilling, просто отрисуем список внутри контейнера. Теперь посмотрим на результат работы:

Не так круто, как с ёжиком в предыдущем уроке. Но тоже сойдёт.

Часть задания готова. Остаётся наладить работу в целевом элементе. Он состоит из контейнера со слушателем и элементов, которые передаются в компонент. В момент «броска» перетаскиваемого элемента в целевой нужно обработать событие onDrop: удалить перетаскиваемый элемент из состояния elements и добавить его в draggedElements.

Создадим обработчики handleDrop и handleDragOver в компоненте DragAndDropContainer:

Скопировать кодJSX

const handleDragOver = e => e.preventDefault();

const handleDrop = () => {

setDraggedElements([

...draggedElements,

draggedElement

]);

setElements([

...elements.filter(element => element.id !== draggedElement.id)

]);

setDraggedElement({});

};

return (

*// Другие элементы в DragAndDropContainer*

<DropTarget onDragOverHandler={handleDragOver} onDropHandler={handleDrop} >

{draggedElements.map(item => (

<div key={item.id} className={styles.item}>

<span className={styles.animalItem}>

{item.content}

</span>

<p>

{item.description}

</p>

</div>

))}

</DropTarget>

*// Другие элементы в DragAndDropContainer*

)

И отредактируем сам компонент DropTarget:

Скопировать кодJSX

import React from 'react';

import styles from './drop-target.module.css'

const DropTarget = (props) => {

const { children, onDropHandler, onDragOverHandler } = props;

return (

<div

className={styles.target}

onDrop={onDropHandler}

onDragOver={onDragOverHandler}

>

{children}

</div>

);

};

export default DropTarget;

Теперь базовая функциональность готова, и мы наконец-то можем отсадить всех муравьёв от ежа, чтобы он их не съел:

Всех муравьёв — в отдельный формикарий.

Потребовалось всего несколько простых слушателей событий, а вся работа легла на состояние. Но есть проблема: ёжик сможет легко пробраться в формикарий и съесть всех муравьёв, а вытащить его обратно мы не сможем. Эта функциональность и сортировка элементов отсутствует внутри DragTarger. Займёмся дополнительной функциональностью в следующем уроке, а после перенесём всю основную логику в Redux.

А пока перейдём к заданиям.

# Drag and Drop. Реальность. Библиотека React-DND

В предыдущем уроке мы много говорили про использование событий из группы onDrag, другими словами — о нативном браузерном dnd Api. Оно достаточно функциональное, но многословное — нужно описывать много обработчиков событий и везде добавлять слушатели. В большом приложении это дополнительный и зачастую сложный код, ведь обработчики событий onDrag редко состоят только из нескольких изменений состояний. В этом уроке мы полностью переложим работу с событиями на стороннюю библиотеку и будем заниматься только мониторингом состояния и визуальной частью интерфейса.

React — популярная библиотека, поэтому в ней есть библиотеки для работы с dnd. Перечислим три самые распространённые:

* react-dnd,
* react-draggable,
* react-beautiful-dnd.

Мы выберем react-dnd, потому что разработчики этой библиотеки предоставляют широкий спектр возможностей для работы с dnd и немногословный код.

## Начало работы с react-dnd

Для начала работы с библиотекой установим два пакета и расскажем о них:

Скопировать кодBASH

npm i react-dnd react-dnd-html5-backend

Сам пакет react-dnd предоставляет функциональность библиотеки. Её мы разберём чуть позже, а сначала поговорим о react-dnd-html5-backend — провайдере для dnd. Другими словами, это браузерный API, который используется непосредственно для перетаскивания элементов.

Провайдер можно заменить на другой. В этом и есть одно из преимуществ react-dnd. Если react-dnd-html5-backend — провайдер для событий группы onDrag, то пакет react-dnd-touch-backend — провайдер для touch событий. Можно написать и свой собственный провайдер. Но мы не будем это делать.

Когда библиотека установлена, разберёмся, как включить react-dnd в проекте. В предыдущем уроке мы рассматривали компоненты DragAndDropContainer, DragSource, DropTarget, но работали только с перетаскиваемыми и целевыми элементами. При работе с react-dnd нужно явно обозначить контейнер, в котором будут происходить все перетаскивания, и обернуть его в провайдер. Провайдер предоставит возможности dnd.

Вернёмся к коду из предыдущего урока, где мы работали с муравьями, рыбкой, жирафом и ёжиком. Удалим из компонента DragAndDropContainer все обработчики и слушатели. И добавим провайдер для dnd:

Скопировать кодJSX

import { DndProvider } from "react-dnd";

import { HTML5Backend } from "react-dnd-html5-backend";

const DragAndDropContainer = () => {

return (

<section className={styles.app}>

<DndProvider backend={HTML5Backend}>

<article className={styles.element}>

<div className={styles.animals}>

{

elements.map( animal => <DraggableAnimal key={animal.id} data={animal}/>)

}

</div>

<DropTarget onDropHandler={handleDrop} >

{

draggedElements.map(item => (

<div key={item.id} className={styles.item}>

<span className={styles.animalItem}>

{item.content}

</span>

<p>

{item.description}

</p>

</div>

))

}

</DropTarget>

</article>

</DndProvider>

</section>

)

}

export default DragAndDropContainer

Мы обернули всю область, в которой будем перетаскивать элементы, в компонент DndProvider с атрибутом backend. В атрибут передали API установленного пакета react-dnd-html5-backend.

Теперь разберёмся с перетаскиваемым и целевым элементом. DraggableAnimal — перетаскиваемый компонент, или DragSource в системе dnd, а DropTarget — целевой компонент, или DropTarget в системе dnd. Работа с компонентами в react-dnd происходит через два кастомных хука:

* useDrag — для перетаскиваемого элемента,
* useDrop — для целевого элемента.

Рассмотрим эти хуки подробнее.

### Перетаскивание элемента. Хук useDrag

useDrag позволяет добавлять элементам функциональность перетаскивания. Хук возвращает массив из трёх значений:

* CollectedProps — объект, который предоставляет другим частям компонента доступ к вычислениям функции collect внутри хука useDrag;
* dragRef — реф, который добавляется перетаскиваемому элементу или компоненту;
* dragPreviewRef — реф, который указывается для элемента, используемого в качестве превью, или компонента во время перетаскивания. dragPreviewRef использовать необязательно.

Звучит сложно. Чтобы разобраться в этом, заглянем внутрь хука useDrag. Сам хук принимает два параметра:

* объект или функцию;
* массив deps, который используется для мемоизации вычислений.

Если с deps вы знакомы, то первый параметр содержит объект с уникальными для react-dnd полями:

* Обязательное свойство type. Это строка, благодаря которой целевой элемент понимает, какие элементы в него можно перетащить. Это поле мы укажем и в хуке useDrag, и в useDrop.
* Обязательное свойство item. Это данные о перетаскиваемом элементе. Они используются в обоих хуках — useDrag и useDrop. Поскольку react-dnd использует HTML5 dnd api в качестве провайдера — важно помещать в item только ключевую информацию о перетаскиваемом объекте. Почти всегда можно использовать только id конкретного элемента.
* Необязательный метод collect. Об этой функции мы говорили выше — это набор вычислений для работы с пропсами. Метод принимает два параметра: monitor и пропсы.

Возможно, понятнее не стало. Начнём писать код и разберёмся на практике. Перейдём в компонент DraggableAnimal, который отрисовывает иконки с животными. Импортируем и используем хук useDrag в компоненте:

Скопировать кодJSX

import React from 'react';

import styles from "./draggable-animal.module.css";

import { useDrag } from "react-dnd";

const DraggableAnimal = ({data}) => {

const {id, content} = data;

const [] = useDrag({});

return (

<div

className={styles.animalElement}

>

{content}

</div>

)

;

};

export default DraggableAnimal

В самой простой реализации react dnd нам нужно только одно возвращаемое значение — dragRef. Этот реф мы используем на div-элементе. А чтобы все участники dnd обладали информацией о перетаскиваемом элементе — укажем в первом параметре useDrag поля type и item:

Скопировать кодJSX

import React from "react";

import styles from "./draggable-animal.module.css";

import { useDrag } from "react-dnd";

const DraggableAnimal = ({data}) => {

const {id, content} = data;

const [, dragRef] = useDrag({

type: "animal",

item: {id}

});

return (

<div

ref={dragRef}

className={styles.animalElement}

>

{content}

</div>

)

;

};

export default DraggableAnimal

Указав атрибут type со значением animal, мы сможем перетащить элемент только в элемент с таким же type. В качестве параметра item мы указали id — у каждого животного есть id, а нам требуются только эти данные.

Мы реализовали базовое немодное перетаскивание элемента — DragSource. Но интерфейс должен быть дружелюбнее — хорошо бы подсвечивать пользователю, какого именно муравья он пытается перетащить. Тут-то и нужно воспользоваться методом collect с параметром monitor. Параметр monitor — собственная реализация хранилища в React dnd. В объекте monitor 11 методов, но мы разберём только основные из них:

* canDrag — возвращает булевое значение true в случае, если в этот момент никакой элемент не перетаскивается.
* isDragging — возвращает true, если происходит перетаскивание.
* didDrop — возвращает true, если на целевом элементе срабатывает событие drop.
* getItemType — возвращает type перетаскиваемого элемента.
* getItem — возвращает объект item. Это те данные, которые мы указали в ключе item хука useDrag.

Попробуем применить метод collect для кастомизации перетаскиваемого элемента. Например, мы можем полностью скрыть перетаскиваемый элемент из основного контейнера:

Скопировать кодJSX

import React from "react";

import styles from "./draggable-animal.module.css";

import { useDrag } from "react-dnd";

const DraggableAnimal = ({data}) => {

const {id, content} = data;

const [{isDrag}, dragRef] = useDrag({

type: "animal",

item: {id},

collect: monitor => ({

isDrag: monitor.isDragging()

})

});

return (

!isDrag &&

<div

ref={dragRef}

className={styles.animalElement}

>

{content}

</div>

)

;

};

export default DraggableAnimal

В результате на время перетаскивания рыбка-колючка и жираф пропадут из исходного контейнера:

С помощью методов monitor можно легко модифицировать как состояния, так и сам интерфейс приложения на время перетаскивания элемента. Теперь попробуем настроить работу целевого элемента с использованием хука useDrop.

### Работа с целевым элементом. Хук useDrop

Этот хук похож на useDrag структурой и методами объекта monitor, но сам объект внутри хука несколько отличается. Разберём основные параметры и возвращаемые значения этого хука.

Хук возвращает массив из двух значений:

* CollectedProps — объект, который предоставляет другим частям компонента доступ к вычислениям функции collect внутри хука useDrop;
* DropTargetRef — реф, который указывает на целевой элемент.

Внутри объекта хука useDrop наиболее важны:

* Обязательное свойство accept. Его значение — строка, которая должна быть аналогична свойству type перетаскиваемого компонента.
* Необязательные методы hover и drop. Оба принимают в качестве параметра item перетаскиваемого компонента и monitor. Первый метод срабатывает, когда перетаскиваемый элемент попадает в зону целевого. Второй метод — при «броске» перетаскиваемого элемента в целевой.
* Необязательный метод collect. Аналогичен методу collect хука useDrag. Единственная разница — наличие метода isOver и отсутствие метода isDragging.

По аналогии с компонентом DraggableAnimal напишем новый код в компоненте DragTarget:

Скопировать кодJSX

import React from 'react';

import styles from './drop-target.module.css'

import { useDrop } from "react-dnd";

const DropTarget = ({children, onDropHandler}) => {

const [, dropTarget] = useDrop({

accept: "animal",

drop(itemId) {

onDropHandler(itemId);

},

});

return (

<div

ref={dropTarget}

className={styles.target}

>

{children}

</div>

);

};

export default DropTarget;

В этом компоненте впервые появились наши вычисления. React-dnd берёт на себя всё, что касается событий dnd, но работу с вычислениями и изменением состояния должен описывать сам разработчик. Опишем изменения состояния в компоненте DragAndDropContainer и передадим их компоненту DropTarget в качестве пропса:

Скопировать кодJSX

import { DndProvider } from "react-dnd";

import { HTML5Backend } from "react-dnd-html5-backend";

const DragAndDropContainer = () => {

const [elements, setElements] = React.useState([]);

const [draggedElements, setDraggedElements] = React.useState([]);

const handleDrop = (itemId) => {

setElements([

...elements.filter(element => element.id !== itemId.id)

]);

setDraggedElements([

...draggedElements,

...elements.filter(element => element.id === itemId.id)

]);

};

return (

*// Другая возвращаемая разметка и компоненты*

<DropTarget onDropHandler={handleDrop} />

)

}

export default DragAndDropContainer

Теперь вся функциональность восстановлена, и мы можем переносить ёжика и муравьёв в итоговый фрейм:

Стоит подсказать пользователю, что элемент интерфейса можно перетащить в целевой элемент. Например, окрашивая рамку целевого элемента. Для этого можно пойти двумя путями: использовать метод hover хука useDrop или написать собственный метод внутри collect. Воспользуемся вторым подходом: мы хотим получать значения из useDrop в других местах. Обратимся к методу isOver объекта monitor, он ведёт себя схожим образом с hover:

Скопировать кодJSX

import React from 'react';

import styles from './drop-target.module.css'

import { useDrop } from "react-dnd";

const DropTarget = ({children, onDropHandler}) => {

const [{isHover}, dropTarget] = useDrop({

accept: "animal",

drop(itemId) {

onDropHandler(itemId);

},

collect: monitor => ({

isHover: monitor.isOver(),

})

});

const borderColor = isHover ? 'lightgreen' : 'transparent';

return (

<div

ref={dropTarget}

className={styles.target}

style={{borderColor}}

>

{children}

</div>

);

};

export default DropTarget;

Теперь интерфейс заиграет новыми красками. А обратная связь от интерфейса — всегда хорошо.

Дальше дорабатывать интерфейс очень просто. Например, чтобы ёжик не оказался с муравьями в одном фрейме, можно добавить всем животным отдельный атрибут type и определять, каких животных можно перетаскивать, а каких — нет. Тогда ёжик останется голоден, зато муравьи — живы.

## Заключение

В этом уроке мы разобрали основы библиотеки react-dnd и попробовали её в деле. Библиотека облегчает работу с интерфейсом и берёт на себя всю работу с событиями. Что важно, код становится чище и проще. И у нас появляется больше времени на работу с ёжиками, муравьями, рыбкой-колючкой и жирафом. Сейчас мы до сих пор передаём все вычисления и работу с состоянием в качестве пропса onDropHandler. И вы наверняка понимаете, к чему всё идёт — эту часть можно спрятать в Redux.

Но прежде чем совмещать react-dnd и Redux — перейдём к заданиям.

**Связь React-DND с Redux**

В предыдущем уроке мы разобрали основы работы с библиотекой react-dnd. Этих знаний достаточно для реализации сложных механизмов перетаскивания. В этом уроке мы реализуем перетаскивание животных на Redux и попробуем включить в методы DnD работу с экшенами.

Ранее мы хранили животных в двух разных состояниях: в elements — массиве с ещё не перенесёнными животными — и в draggedElements — массиве с животными, которых мы уже перетащили. Разделение данных на две категории встречается часто. Например, в интернет-магазине, где у пользователя могут быть избранные и отложенные товары. В этом случае состояния удобнее разделять.

Но иногда важно хранить данные в одном состоянии. Например, на доске планирования может быть несколько колонок со статусами задач. В этом случае удобнее хранить все задачи в одном состоянии. А колонка, в которой оказывается задача, определяется полем внутри объекта.

Попробуем взглянуть на DnD с другой стороны — когда все животные хранятся в одном состоянии. Добавим ещё несколько фреймов для хранения животных и реализуем возможность перетаскивания между разными целевыми элементами. Для решения этой задачи воспользуемся Redux.



*Четыре фрейма — второе дыхание для проекта*

Организуем хранилище. Создадим директорию services/, а ещё — файлы редьюсера и экшена для каждого компонента: DropTarget и DraggableAnimal. Редьюсер DropTarget будет содержать только информацию о названии каждой из досок:

Скопировать кодJSX

*// /services/reducers/drop-target.js*

const initialState = {

boards: ["default", "fish", "mammals", "insects"]

};

export const dropTargetReducer = (state = initialState, action) => {

return state;

};

Редьюсер DraggleAnimal будет содержать исходное состояние с массивом животных animals. У каждого животного будет атрибут board с начальным значением default. В сам редьюсер draggableAnimalReducer будет записана обработка только одного экшена — UPDATE\_TYPE:

Скопировать кодJSX

*// /services/reducers/draggable-animal.js*

import { UPDATE\_TYPE } from "../actions/draggable-animal";

const initialState = {

animals: [

{

id: 12,

content: "🦔",

description: "Ёжики очень классные. И являются настоящими звёздами в Instagram.",

board: "default"

},

*// Другие животные*

]

};

export const draggableAnimalReducer = (state = initialState, action) => {

switch (action.type) {

case UPDATE\_TYPE: {

return {

...state,

animals: state.animals.map(animal =>

animal.id === action.id ? {...animal, board: action.board} : animal

)

};

}

default:

return state;

}

}

*// /services/reducers/index.js*

import { combineReducers } from "redux";

import { draggableAnimalReducer } from "./draggable-animal";

import { dropTargetReducer } from "./drop-target";

export const rootReducer = combineReducers({

animalList: draggableAnimalReducer,

boardList: dropTargetReducer

});

Подключим стор к проекту и переделаем компонент DragAndDropContainer. Теперь у нас есть несколько целевых элементов для перетаскивания, поэтому заберём из хранилища массив boards и отрисуем по компоненту DropTarget:

Скопировать кодJSX

const DragAndDropContainer = () => {

*// Получим все доски из хранилища*

const boards = useSelector(state => state.boardList.boards)

return (

<section className={styles.app}>

<DndProvider backend={HTML5Backend}>

<article className={styles.element}>

{

// Отрисуем каждую доску и передадим её название в качестве пропса

boards.map((item, i) => (

<DropTarget key={i} board={item} />

))

}

</article>

</DndProvider>

</section>

)

};

export default DragAndDropContainer;

Теперь в компоненте DragAndDropContainer не нужно работать с изменениями состояния и передавать их в компонент DropTarget. Из компонента DropTarget потребуется отправлять экшен на обновление состояния animals. Сделать это можно в методе drop хука useDrop. А все необходимые для отправки экшена данные у нас будут:

* название доски, в которую мы переносим элемент, получаем из пропса board;
* идентификатор перетаскиваемого элемента получим из хранилища react-dnd.

Обновим DropTarget и хук useDrop в нём:

Скопировать кодJSX

const DropTarget = ({ board }) => {

const dispatch = useDispatch();

*// Получим массив всех животных из хранилища*

const animals = useSelector(state => state.animalList.animals)

const [{ isHover } , drop] = useDrop({

accept: "animal",

collect: monitor => ({

Организуем хранилище. Создадим директорию `services/`, а ещё — файлы редьюсера и экшена для каждого компонента: `DropTarget` и `DraggableAnimal`. Редьюсер `DropTarget` будет содержать только информацию о названии каждой из досок:

```jsx

// /services/reducers/drop-target.js

const initialState = {

boards: ["default", "fish", "mammals", "insects"]

};

export const dropTargetReducer = (state = initialState, action) => {

return state;

};

Редьюсер DraggleAnimal будет содержать исходное состояние с массивом животных animals. У каждого животного будет атрибут board с начальным значением default. В сам редьюсер draggableAnimalReducer будет записана обработка только одного экшена — UPDATE\_TYPE:

Скопировать кодJSX

*// /services/reducers/draggable-animal.js*

import { UPDATE\_TYPE } from "../actions/draggable-animal";

const initialState = {

animals: [

{

id: 12,

content: "🦔",

description: "Ёжики очень классные. И являются настоящими звёздами в Instagram.",

board: "default"

},

*// Другие животные*

]

};

export const draggableAnimalReducer = (state = initialState, action) => {

switch (action.type) {

case UPDATE\_TYPE: {

return {

...state,

animals: [...state.animals].map(animal =>

animal.id === action.id ? {...animal, board: action.board} : animal

)

};

}

default:

return state;

}

}

*// /services/reducers/index.js*

import { combineReducers } from "redux";

import { draggableAnimalReducer } from "./draggable-animal";

import { dropTargetReducer } from "./drop-target";

export const rootReducer = combineReducers({

animalList: draggableAnimalReducer,

boardList: dropTargetReducer

});

Подключим стор к проекту и переделаем компонент DragAndDropContainer. Теперь у нас есть несколько целевых элементов для перетаскивания, поэтому заберём из хранилища массив boards и отрисуем по компоненту DropTarget:

Скопировать кодJSX

const DragAndDropContainer = () => {

*// Получим все доски из хранилища*

const boards = useSelector(state => state.boardList.boards)

return (

<section className={styles.app}>

<DndProvider backend={HTML5Backend}>

<article className={styles.element}>

{

// Отрисуем каждую доску и передадим её название в качестве пропса

boards.map((item, i) => (

<DropTarget key={i} board={item} />

))

}

</article>

</DndProvider>

</section>

)

};

export default DragAndDropContainer;

Теперь в компоненте DragAndDropContainer не нужно работать с изменениями состояния и передавать их в компонент DropTarget. Из компонента DropTarget потребуется отправлять экшен на обновление состояния animals. Сделать это можно в методе drop хука useDrop. А все необходимые для отправки экшена данные у нас будут:

* название доски, в которую мы переносим элемент, получаем из пропса board;
* идентификатор перетаскиваемого элемента получим из хранилища react-dnd.

Обновим DropTarget и хук useDrop в нём:

Скопировать кодJSX

const DropTarget = ({ board }) => {

const dispatch = useDispatch();

*// Получим массив всех животных из хранилища*

const animals = useSelector(state => state.animalList.animals)

const [{ isHover } , drop] = useDrop({

accept: "animal",

collect: monitor => ({

isHover: monitor.isOver(),

}),

drop(itemId) {

*// Отправим экшен с текущим перетаскиваемым элементом и названием доски*

dispatch({

type: UPDATE\_TYPE,

...itemId,

board

});

},

});

const borderColor = isHover ? "lightgreen" : "transparent";

};

export default DropTarget

Потребуется отрисовать только тех животных, значение свойства board которых соответствует текущему перетаскиваемому элементу DropTarget. Можно воспользоваться методами filter и map:

Скопировать кодJSX

const DropTarget = ({ board }) => {

const dispatch = useDispatch();

const animals = useSelector(state => state.animalList.animals)

const [{ isHover } , drop] = useDrop({

accept: "animal",

collect: monitor => ({

isHover: monitor.isOver(),

}),

drop(itemId) {

dispatch({

type: UPDATE\_TYPE,

...itemId,

board

});

},

});

const boardClass = board === 'default' ? styles.animalsPull : styles.animals;

const borderColor = isHover ? 'lightgreen' : 'transparent';

return (

<div ref={drop} className={boardClass} style={{borderColor}}>

{animals

// Получим массив животных, соответствующих целевому элементу

.filter(animal => animal.board === board)

// Отрисуем массив

.map(animal => <DraggableAnimal key={animal.id} data={animal} />)

}

</div>

)

};

export default DropTarget

Теперь каждое животное можно перетаскивать между досками. Но все данные о них хранятся в одном источнике хранилища — animals:

Последнее, что требуется сделать для достижения такого результата, — модифицировать разметку DraggableAnimal в зависимости от того, в каком целевом элементе располагается перетаскиваемый. Самое простое решение — реализовать эту логику представления внутри самого компонента DraggableAnimal. Это верное решение, ведь мы никак не модифицируем данные в зависимости от представления карточки животного. Разместим отображение в компоненте DraggableAnimal:

Скопировать кодJSX

const DraggableAnimal = ({ data }) => {

const { id, board } = data;

const [{ isDrag }, drag] = useDrag({

type: "animal",

item: { id },

collect: monitor => ({

isDrag: monitor.isDragging()

})

});

*// Отображение DraggableAnimal в целевом элементе "default"*

const draggableAnimalPreview = (

<div ref={drag} className={styles.animalElement}>

{data.content}

</div>

);

*// Отображение DraggableAnimal в других целевых элементах*

const draggableAnimalCard = (

<div ref={drag} className={styles.item}>

<span className={styles.animalItem}>

{data.content}

</span>

<p>

{data.description}

</p>

</div>

);

*// Проверяем, где сейчас находится карточка*

const draggableAnimalContent = (

board === "default" ? draggableAnimalPreview : draggableAnimalCard

);

return (

!isDrag && draggableAnimalContent

);

};

export default DraggableAnimal

Обратите внимание: в поле item хука useDrag мы сохраняем именно объект, хотя фактически это всего лишь числовое значение id. Лучше заранее закладывать объект в поле item. Позже в поле могут добавиться ещё идентификаторы или данные, и такие изменения повлияют на другие участки кода: редьюсер или целевой элемент.

На этом изучение основ dnd и второй спринт подходят к концу. Ёжика можно отсадить от муравьёв, а рыбку-колючку поместить в отдельный аквариум.

В интерфейсе всегда есть места для улучшений с помощью dnd: например, работа с сортировкой перетаскиваемых элементов внутри целевого. В основе этого лежат по большей части навыки работы с массивами. Кажется, что после Redux никакие Array.splice не страшны.

Прежде чем переходить к финальной части проектной работы, попрактикуйтесь с dnd в заданиях.

# Drag and Drop. Заключение

В этой теме вы изучили мощную составляющую интерфейса — Drag and Drop. Сначала познакомились с базовыми концепциями: событиями onMouse и определением положения элемента и курсора относительно экрана пользователя. Затем перешли к более сложному и удобному инструменту — библиотеке react-dnd.

Вы узнали, как можно реализовать DnD на одних событиях onMouse. Писать вычисления для такого DnD долго и достаточно сложно, зато результатом этих инвестиций станет функциональность, которая не ограничена HTML5 DnD API. Это полезный, но редко используемый навык.

После вы познакомились с событиями onDrop — API для DnD, встроенным в браузер. Именно эти события лежат в основе библиотеки react-dnd. Работать с браузерным API проще, чем с событиями мыши — требуется меньше кода. Но при этом остаётся ряд ограничений, в том числе и визуальных.

Затем вы освоили мощное оружие — библиотеку react-dnd. Она предоставляет самый быстрый способ реализации перетаскивания частей интерфейса и хорошо подходит для React-Redux проектов. К тому же, для работы библиотека использует собственное Redux- хранилище.

Сложный месяц подходит к концу. Пора поговорить о проекте.

## Что можно сделать в проекте

Теперь у вас получится выполнить 5 и 6 пункты из [финальной части проектной работы.](https://practicum.yandex.ru/trainer/web-plus/lesson/c4f0f268-8ff6-49f1-a491-41bd20ef4ba9/) Это непростая задача, потому что в рамках курса (за исключением проектного месяца) вы не работали с дополнительными библиотеками, которые помогают в реализации функциональности для улучшения пользовательского опыта. Если Redux выводит на новый уровень Developer Experience, то react-dnd выведет на новый уровень взаимодействие с приложением.

Мы рекомендуем подойти к работе с react-dnd следующим образом:

* Выделите новый простой компонент в Stellar Burger и реализуйте в нём функциональность перетаскивания в связке с Redux.
* Попробуйте реализовать шестой пункт из проектной работы на этом же тестовом компоненте. Про внутреннюю сортировку мы не рассказывали в уроках, потому что её пример есть в официальной документации. Чтобы облегчить поиск, [вот ссылка на реализацию сортировки](https://react-dnd.github.io/react-dnd/examples/sortable/simple).
* Как только у вас будет готов прототип для перетаскивания ингредиентов с их сортировкой — переходите к «чистовой» разработке и реализации этой функциональности в ингредиентах, то есть переходите непосредственно к пунктам 5-6.

На этом проектные работы на этот месяц завершены. Следующая тема не понадобится вам в проекте этого месяца, но понадобится в проекте следующего месяца.