Dish🍜

🔍 Подробный технический разбор `DishForm.tsx`

```
import { useState, type FormEvent, type JSX } from "react";
import { useDispatch } from "react-redux";
```

◾ Что происходит

- `useState` — хук React для хранения состояния внутри функционального компонента.

  Каждый вызов `useState` создаёт отдельную «ячейку памяти», связанную с компонентом.

- `FormEvent` — тип события из TypeScript, используется для типизации `onSubmit` в форме.
- `JSX` — тип возвращаемого значения (React элемент).
- `useDispatch` — хук из `react-redux`, который даёт доступ к функции `dispatch` для отправки действий (actions) в Redux.

⚙️ *Когда компонент рендерится, React создаёт для него замыкание и связывает значения всех* `useState`.

🔷 Что такое замыкание в JavaScript

Замыкание (closure) — это механизм, при котором функция «помнит» переменные из внешней области видимости, даже если эта область уже завершила выполнение.

Пример на чистом JS:

```
function counter() {
  let count = 0;
```

```
  return function () {
    count++;
    console.log(count);
  };
}

const inc = counter();
inc(); // 1
inc(); // 2
```

🧠 Здесь функция `inc` “замкнула” переменную `count` в своём внутреннем контексте.

Даже после того, как `counter()` отработала, переменная `count` не исчезла — она живёт в замыкании.

🔷 Как это связано с React и `useState`

Когда ты вызываешь в компоненте:

```
const [title, setTitle] = useState('');
```

React делает под капотом **следующее**:

1. При первом рендере компонент вызывается как функция.

   Все локальные переменные, включая `useState`, попадают в контекст этой функции.

2. React сохраняет текущее состояние `title` **внутри своей внутренней структуры (Fiber)**.

3. Возвращаемая функция `setTitle` **замыкает ссылку** на этот участок памяти, где React хранит значение `title`.

То есть `setTitle` — это функция, которая помнит, **с каким именно состоянием (ячейкой памяти)** она связана.

Эта связь работает именно благодаря **замыканию**.

🔷 Что происходит при повторных рендерах

Каждый раз, когда компонент рендерится заново:

1. React заново вызывает функцию `DishForm()`.

   То есть все локальные переменные пересоздаются.

2. Но `useState` **не создаёт новое состояние**, а достаёт старое — по порядковому индексу вызова (React запоминает, что это “второй useState”, “третий useState” и т.д.).

3. При этом каждый `set...` (например, `setTitle`) всё так же замыкает доступ к своей ячейке состояния, и знает, что обновлять именно её.

💡 Иными словами:

> Каждый вызов useState создаёт замыкание между функцией-компонентом и внутренним состоянием React,
>
> чтобы при каждом новом рендере связь между “логикой” и “данными” сохранялась.

🔷 Почему это важно понимать

Если бы не было замыканий:

- при каждом рендере React терял бы связь между `setTitle` и значением `title`;
- состояние бы сбрасывалось;
- не работали бы эффекты и обновления UI.

Замыкания обеспечивают то, что “старое состояние” доступно внутри функции,

даже после того, как компонент уже перерендерился или React пересоздал его экземпляр.

🔍 Пример для визуализации

```
function DishForm() {
  const [title, setTitle] = useState('');

  function handleChange(e) {
    // Здесь handleChange — замыкание
    // Оно “помнит” текущий title и setTitle
    setTitle(e.target.value);
  }

  console.log('render', title);
  return <input value={title} onChange={handleChange} />;
}
```

Каждый раз, когда ты вводишь текст:

1. Вызывается `handleChange` — она замкнула `setTitle`.
2. React обновляет состояние и вызывает компонент заново.
3. Создаётся новое замыкание с новым `title`.

📘 Замыкание гарантирует, что `handleChange` всегда работает с актуальной версией `setTitle`,

но “помнит”, к какой конкретно ячейке состояния оно относится.

🧠 Коротко в одну строку

> В контексте React замыкание — это механизм, связывающий каждую функцию (setState, обработчик и т.п.) с актуальным состоянием компонента, чтобы при каждом рендере они знали, какие данные обновлять.

🧩 *Redux не знает о `useState` — это локальное состояние, не глобальное.*

```
export default function DishForm(): JSX.Element {
```

- Объявляем компонент `DishForm` как **функцию**, возвращающую JSX.
- Сразу указываем тип `JSX.Element`, чтобы TypeScript понимал, что это React-компонент.

```
const [title, setTitle] = useState<string>('');
const [category, setCategory] = useState<string>('');
const [image, setImage] = useState<string>('');
const [price, setPrice] = useState<number>(0);
const [error, setError] = useState<string>('');
```

■ Что делает каждая строка:

- Создаём **5 состояний**:
  1. `title` — строка для названия блюда.
  2. `category` — выбранная категория (main, dessert...).
  3. `image` — URL картинки.
  4. `price` — число (начальное значение 0).
  5. `error` — текст ошибки для валидации формы.

🧠 При каждом изменении поля ввода React вызывает `set...`-функцию и рендерит компонент заново с новым состоянием.

```
const dispatch = useDispatch();
```

- Получаем функцию `dispatch` из Redux.
- С её помощью мы позже **отправим действие** вида `{ type: "dishes/create", payload: {...} }` в редьюсер.

⚙️ Под капотом `useDispatch` подключён к контексту `<Provider store={store}>` в `main.tsx`.

Если `Provider` не оборачивает `App`, здесь будет ошибка: *“could not find react-redux context value”*.

```
function validateInputs(): boolean {
  if (title.trim() === '') {
    setError('Название не должно быть пустым');
    return false;
  }
  if (category.trim() === '') {
    setError('Выберите категорию');
    return false;
  }
  if (image.trim() === '') {
    setError('Заполните поле картинка');
    return false;
  }
  if (price < 0) {
    setError('Цена не может быть отрицательной');
    return false;
  }
  return true;
}
```

◾ Подробно

- Функция возвращает `true`, если данные корректны.
- `trim()` убирает пробелы по краям.
- При ошибке устанавливаем `setError(...)`, что вызывает повторный рендер.
- Возвращаем `false`, чтобы `handleSubmit` не отправлял данные дальше.

💡 Это **локальная валидация**, выполняемая до отправки action в Redux.

Если бы она была на сервере, то ошибки возвращались бы в ответе запроса.

```
function clearInputsAndError(): void {
  setCategory('');
  setTitle('');
  setPrice(0);
  setImage('');
  setError('');
}
```

- После успешного добавления блюда форма должна очиститься.
- Все состояния сбрасываются в исходные значения.
- Без этого поля сохраняли бы прошлые значения, что мешает UX.

```
function handleSubmit(e: FormEvent<HTMLFormElement>): void {
  e.preventDefault();
  if (validateInputs()) {
    dispatch({ type: "dishes/create", payload: { title, category, price, image } }
});
    clearInputsAndError();
  }
}
```

Построчно

1. `e.preventDefault()` — отменяет стандартное поведение формы (перезагрузку страницы).
2. `validateInputs()` — проверяем корректность данных.
3. Если всё ок, `dispatch()` отправляет в Redux action:

   ```
   { type: "dishes/create", payload: { title, category, price, image } }
   ```

Этот объект попадёт в `dishesReducer`, где будет добавлено новое блюдо с `id`.

4. После этого поля очищаются.

💬 *Redux не возвращает результат — он просто уведомляет store, что состояние нужно изменить.*

🧩 *Компоненты, подписанные через `useSelector`, увидят новое состояние и перерендерятся.*

```
return (
  <div>
    <h1>Форма создания меню</h1>
    <form onSubmit={handleSubmit}>
      {error && <div style={{ color: 'red' }}>{error}</div>}
```

- Если `error` не пустая строка, JSX рендерит `<div>` с сообщением об ошибке.

🔍 Что делает конструкция `error && <div>...</div>`

В JSX (и вообще в JavaScript) логический оператор `&&` работает так:

```
A && B
```

возвращает:

- `B`, если `A` **истинно** (truthy),
- `A`, если `A` **ложно** (falsy).

✅ Пример:

```
true && 'Hello'    // → 'Hello'
false && 'Hello'   // → false
```

🔷 Применим это к JSX

В JSX значение `false`, `null`, `undefined` или `''` (пустая строка) **не рендерятся вообще**.

Это удобно для условного отображения.

📘 То есть:

```
{error && <div>{error}</div>}
```

означает буквально:

> “Если error истинное значение, то отрендери ,
>
> а если `error` пустая строка, null или undefined — не показывай ничего”.

🔷 Разбор по шагам

Значение `error`	Что произойдёт	Результат в DOM
`'Ошибка ввода!'`	truthy → рендерим `<div>`	`<div>Ошибка ввода!</div>`
`''` (пустая строка)	falsy → ничего не рендерим	нет элемента
`null` или `undefined`	falsy → ничего не рендерим	нет элемента

🔶 То есть :

> “Если error не пустая строка, JSX рендерит с сообщением об ошибке.”

— абсолютно верная ✅

и **не наоборот**.

💬 Если написать "наоборот"

Если кто-то прочитает это как

> “Если error пустая строка — рендерится div”

— то это будет **ошибка**.

Пустая строка (`''`) — это falsy значение, и JSX просто ничего не покажет.

💡 Для наглядности (в консоли):

```
Boolean('Ошибка')  // true
Boolean('')        // false
```

⇒ JSX рендерит `<div>` **только если** `Boolean(error)` → `true`.

✅ **Вывод:**

> Всё верно:
>
> Когда `error` не пустая строка, JSX действительно рендерит `<div>` с сообщением об ошибке.
>
> Когда `error` пустая, ничего не отображается.

- `style={{ color: 'red' }}` — inline-стиль.
- `<form onSubmit={handleSubmit}>` привязывает обработчик.

```
<input
  type="text"
  placeholder="title"
  value={title}
  onChange={(e) ⇒ setTitle(e.target.value)}
```

```
      />
```

◾ Пошагово

- `type="text"` — поле ввода текста.
- `value={title}` — двустороннее связывание с состоянием React.
- `onChange` — обновляет состояние при каждом вводе символа.

⚙️ Когда пользователь набирает текст:

1. React вызывает `setTitle(...)`.
2. Компонент перерендеривается.
3. `value` обновляется → инпут показывает новое значение.

```
<input
  type="text"
  placeholder="image"
  value={image}
  onChange={(e) => setImage(e.target.value)}
/>
```

То же самое, только для ссылки на картинку блюда.

```
<select
  name="category"
  value={category}
  onChange={(e) => setCategory(e.target.value)}
>
  <option value="" disabled>category</option>
  <option value="main">main</option>
  <option value="dessert">dessert</option>
  <option value="snack">snack</option>
```

```
      </select>
```

Детали:

- Элемент `<select>` связан со `state` через `value={category}`.
- При выборе пункта вызывается `setCategory`.
- `disabled` у первой опции не позволяет выбрать пустое значение.
- Ошибка часто встречается у новичков: ставят `selected` на `<option>` — в React это не нужно, значение управляется через `value`.

```
      <input
        type="number"
        value={price}
        onChange={(e) => setPrice(Number(e.target.value))}
      />
```

- Преобразуем `e.target.value` (строку) в число через `Number()`.

  Иначе React выдаст предупреждение: *“A component is changing an uncontrolled input of type number to be controlled”*.

```
        <button type="submit">Создать блюдо</button>
      </form>
    </div>
  );
}
```

- Кнопка вызывает `onSubmit` у формы, который срабатывает на `handleSubmit`.
- После успешного создания блюда Redux добавит его в state, и компонент списка (`DishesList`) отобразит его мгновенно.

🧭 Логика выполнения по этапам

Этап	Что делает React/Redux	Результат
1	Пользователь вводит данные	`useState` хранит значения
2	Нажимается кнопка “Создать блюдо”	вызывается `handleSubmit`
3	`validateInputs` проверяет поля	ошибка или продолжение
4	`dispatch()` отправляет action	редьюсер добавляет элемент
5	Redux обновляет `state`	`DishesList` перерендеривается
6	`clearInputsAndError()` сбрасывает форму	готово к следующему блюду

✏️ Подробный разбор `DishEditForm.tsx`

```
import { useState, type FormEvent, type JSX } from 'react';
import EditNoteIcon from '@mui/icons-material/EditNote';
import { useDispatch } from 'react-redux';
import type Dish from './types/Dish';
```

◾ Объяснение:

- Импортируем `useState`, `FormEvent` и `JSX`, как и в `DishForm` — они нужны для управления состоянием и типизации формы.
- `EditNoteIcon` — компонент-иконка из библиотеки MUI. Она будет кнопкой, открывающей/закрывающей форму редактирования.
- `useDispatch` — подключение к Redux store.
- Тип `Dish` — нужен, чтобы обозначить, что этот компонент редактирует именно блюдо, а не произвольный объект.

```
export default function DishEditForm(props: { dish: Dish }): JSX.Element {
```

- Объявляем компонент, принимающий через `props` одно свойство `dish` типа `Dish`.
- Это одно блюдо, данные которого приходят из списка (`DishesList`).
- Типизация `props: { dish: Dish }` даёт студентам уверенность, что React передаст объект строго той структуры, что описана в `Dish.ts`.

```
const { dish } = props;
```

- Деструктурируем `props` для удобства.
- Теперь можно обращаться к `dish.title`, `dish.category` и т.д., без `props.dish`.

```
const [toggle, setToggle] = useState<boolean>(false);
const handleToggle = (): void => {
  setToggle(!toggle);
};
```

◾ Объяснение:

- `toggle` управляет видимостью формы.
- При значении `false` — форма скрыта.
- При клике по ✏️ (`EditNoteIcon`) вызывается `handleToggle`, меняющий `toggle` на противоположное.
- `useState<boolean>` уточняет тип, чтобы исключить случайные строковые значения.

📘 React рендерит JSX заново, и если `toggle === true`, то условный рендер `{toggle && <form>...</form>}` покажет форму.

```
const [title, setTitle] = useState<string>(dish.title);
const [category, setCategory] = useState<string>(dish.category);
const [image, setImage] = useState<string>(dish.image);
const [price, setPrice] = useState<number>(dish.price);
const [error, setError] = useState<string>('');
```

Объяснение:

- Инициализируем поля формы текущими данными блюда — это **редактирование**, не создание.
- `error` изначально пустая строка.
- При вводе данных значения будут меняться в локальном состоянии (а не сразу в Redux).

```
const dispatch = useDispatch();
```

- Подключаем `dispatch` для отправки действия `dishes/edit` при сохранении изменений.

```
function validateInputs(): boolean {
  if (title.trim() === '') {
    setError('Название не должно быть пустым');
    return false;
  }
  if (category.trim() === '') {
    setError('Выберите категорию');
    return false;
  }
  if (image.trim() === '') {
    setError('Заполните поле картинка');
    return false;
```

```
  }
  if (price < 0) {
    setError('Цена не может быть отрицательной');
    return false;
  }
  return true;
}
```

Объяснение:

- Такая же логика, как в `DishForm`, но используется для проверки перед редактированием.
- Возвращает `false`, если ошибка, и сохраняет текст ошибки для отображения.

```
function resetInputsAndError(): void {
  setCategory(dish.category);
  setTitle(dish.title);
  setPrice(dish.price);
  setImage(dish.image);
  setError('');
}
```

Объяснение:

- Если пользователь отменил или сохранил изменения, форма сбрасывается к исходным значениям блюда.

```
function handleSubmit(e: FormEvent<HTMLFormElement>): void {
  e.preventDefault();
  if (validateInputs()) {
```

```
    dispatch({
      type: 'dishes/edit',
      payload: {
        title, category, image, price, id: dish.id
      }
    });
    resetInputsAndError();
  }
}
```

Объяснение:

1. `preventDefault()` — предотвращает перезагрузку формы.
2. Проверяем данные.
3. Если всё корректно, отправляем action в Redux:

   ```
   { type: 'dishes/edit', payload: { ...обновлённое блюдо... } }
   ```

4. Редьюсер (`dishesReducer`) ищет блюдо по `id` и заменяет старый объект на новый.
5. После этого поля сбрасываются.

```
return (
  <div>
    <EditNoteIcon onClick={handleToggle} />
    {toggle && (
      <form onSubmit={handleSubmit}>
        {error && <div style={{ color: 'red' }}>{error}</div>}
```

- `EditNoteIcon` — иконка-«карандаш», клик по ней показывает/скрывает форму.

- `{toggle && (...)}` — условный рендеринг.
- При ошибке появляется `<div>` с красным текстом.

Остальные `<input>` и `<select>` работают точно как в `DishForm`, только значения инициализированы существующими данными блюда.

После сохранения изменений список блюд (`DishesList`) автоматически перерендерится, потому что Redux обновит `state`.

 Подробный разбор `DishesList.tsx`

```
import type { JSX } from "react";
import styles from './DishesList.module.css';
import DishEditForm from "./DishEditForm";
import { useDispatch, useSelector } from "react-redux";
import selectDishes from "./selectors";
import ClearIcon from '@mui/icons-material/Clear';
import type { DishId } from "./types/Dish";
```

■ Объяснение:

- `useSelector` — хук для получения данных из Redux store.
- `useDispatch` — отправка действий (`delete`).
- `selectDishes` — селектор, возвращающий массив блюд.
- `ClearIcon` — иконка «крестик» для удаления блюда.
- `DishEditForm` — форма редактирования, используется внутри списка.
- Подключаем стили из модуля CSS, чтобы классы не пересекались между компонентами.

```
export default function DishesList(): JSX.Element {
  const dishes = useSelector(selectDishes);
```

```
const dispatch = useDispatch();
```

- `useSelector(selectDishes)` берёт список блюд из Redux state.

  Теперь `dishes` — это массив объектов `Dish`.
- `dispatch` нужен, чтобы отправлять действия при удалении блюда.

```
const handleDelete = (id: DishId): void => {
  dispatch({ type: "dishes/delete", payload: id });
};
```

- При клике на крестик передаётся `id` блюда.
- `dispatch` вызывает `dishesReducer`, который фильтрует массив, удаляя блюдо с этим `id`.

```
return (
  <ul className={styles.list}>
    {dishes.map((dish) => (
      <li key={dish.id} className={styles.dishCard}>
        <h3 className={styles.heading}>{dish.title}</h3>
        <p className={styles.category}>{dish.category}</p>
        <img className={styles.image} src={dish.image} alt={dish.title} />
        <p className={styles.price}>{dish.price} €</p>
        <div className={styles.icons}>
          <ClearIcon onClick={() => handleDelete(dish.id)} />
          <DishEditForm dish={dish} />
        </div>
      </li>
    ))}
  </ul>
);
```

```
}
```

◾ Объяснение:

1. `dishes.map()` — создаёт массив JSX-элементов, по одному `<li>` на блюдо.
2. Каждый элемент должен иметь `key={dish.id}` — это помогает React эффективно обновлять DOM.
3. Отображаются поля блюда (`title`, `category`, `price`, `image`).
4. `ClearIcon` вызывает `handleDelete`.
5. `DishEditForm` получает текущее блюдо через пропс `dish` и отображает ✏️ для редактирования.
6. В `className` используются стили из CSS-модуля — изолированные классы (например `list_abc123`).

⚙️ Подробный разбор `dishesReducer.ts`

```
import type Dish from "./types/Dish";
import type { Action } from "./types/Action";
import { uid } from "uid";
```

- Импортируем типы и функцию `uid()` — генератор уникальных ID.
- `uid()` создаёт короткие строки вроде `"a3f9k2n"`.

```
const initialState: Dish[] = [
  {
    id: uid(),
    title: 'Pie',
    category: 'dessert',
    price: 12,
    image: 'https://fsd.multiurok.ru/html/2018/05/03/s_5aeb2280cc6d6/89406
```

```
5_15.jpeg'
  }
];
```

- Это **начальное состояние Redux**: один пример блюда (`Pie`).
- Удобно для теста — пользователь видит список не пустым.

```
export default function dishesReducer(
  state: Dish[] = initialState,
  action: Action
): Dish[] {
```

- Основная функция редьюсера.
- Принимает два аргумента:
  - `state` — текущее состояние (`Dish[]`);
  - `action` — объект `{ type, payload }`.
- Если `state` не передан (первый запуск), используется `initialState`.

```
  switch (action.type) {
```

- `switch` определяет, какое действие выполняется.

```
    case 'dishes/create':
      return [...state, { ...action.payload, id: uid() }];
```

- Добавляем новое блюдо:
  - `...state` копирует старые элементы;

- `{ ...action.payload, id: uid() }` добавляет новый объект (данные из формы + id).

- Возвращаем новый массив (immutability).

```
  case 'dishes/delete':
    return state.filter((dish) ⇒ dish.id !== action.payload);
```

- Удаление: оставляем все блюда, у которых `id` ≠ `payload`.

```
  case 'dishes/edit':
    return state.map((dish) ⇒
      dish.id === action.payload.id ? action.payload : dish
    );
```

- Редактирование: заменяем блюдо, у которого совпадает `id`.
- Остальные элементы массива не трогаем.

```
  default:
    return state;
 }
}
```

- Если действие не распознано, возвращаем состояние без изменений.
- Это защита от случайных dispatch.

🧩 Подробный разбор `Action.ts`

```
import type Dish from "./Dish";
import type { DishId } from "./Dish";
import type DishDto from "./DishDto";
```

```
export type Action =
  | { type: "dishes/create", payload: DishDto }
  | { type: "dishes/delete", payload: DishId }
  | { type: "dishes/edit", payload: Dish };
```

◾ Пояснение:

- Здесь мы описываем **все возможные Redux-действия**, которые reducer может обрабатывать.
- Каждое действие строго типизировано:
  - `"dishes/create"` ожидает данные без id (DishDto);
  - `"dishes/delete"` — только id;
  - `"dishes/edit"` — полное блюдо (Dish).
- Это помогает TypeScript предупреждать ошибки: если кто-то забудет передать `image` — компилятор выдаст ошибку.

🍽️ Подробный разбор `Dish.ts` и `DishDto.ts`

```
export default interface Dish {
  id: string;
  title: string;
  category: string;
  price: number;
  image: string;
}

export type DishId = Dish['id'];
```

- Интерфейс описывает структуру одного блюда.

- `DishId` вытаскивает тип поля `id` из `Dish` — полезно для точности типов.

```
export default interface DishDto {
  title: string;
  category: string;
  price: number;
  image: string;
}
```

- `DishDto` используется в `DishForm` — это данные до присвоения id.
- Такой подход имитирует логику REST API: клиент отправляет DTO, сервер добавляет id и возвращает готовый объект.

🧠 Подробный разбор `selectors.ts`

```
import type { RootState } from "../../store";
import type Dish from "./types/Dish";

const selectDishes = (state: RootState): Dish[] => state.dishes;

export default selectDishes;
```

◾ Пояснение:

- Селектор — функция, которая извлекает нужный кусок состояния из Redux Store.
- Здесь она возвращает `state.dishes` — массив всех блюд.
- `RootState` — общий тип состояния всего приложения (определён в `store.ts`).
- Используется в `DishesList`:

```
const dishes = useSelector(selectDishes);
```

🧭 Финальная схема данных

```
DishForm  → dispatch({ type: 'dishes/create' }) ┐
DishEditForm → dispatch({ type: 'dishes/edit' }) │
DishesList   → dispatch({ type: 'dishes/delete' }) │
            ↓
        dishesReducer()
            ↓
       обновлённый state
            ↓
    useSelector(selectDishes)
            ↓
    перерендер компонента списка
```

⚙️ Подробный технический разбор `store.ts` (чистый Redux + combineReducers)

📄 Код

```
import { combineReducers, createStore } from "redux";
import dishesReducer from "./components/dishes/dishesReducer";

const store = createStore(combineReducers(
   {
      dishes: dishesReducer
```

```
    }
));

export default store;
export type RootState = ReturnType<typeof store.getState>;
```

🔍 Построчное объяснение

1️⃣

```
import { combineReducers, createStore } from "redux";
```

- Импортируются две базовые функции из библиотеки **Redux**:
  - `createStore()` — создаёт объект *Store* (центральное хранилище состояния приложения).

    Он хранит текущее состояние и позволяет вызывать `dispatch(action)` для его изменения.
  - `combineReducers()` — объединяет несколько редьюсеров в один “корневой” редьюсер.

    Это необходимо, чтобы приложение могло иметь разные “участки состояния” (state slices), каждый из которых управляется своим редьюсером.

📘 Даже если сейчас редьюсер один (`dishesReducer`), использование `combineReducers` — **архитектурно правильное решение**, потому что потом легко добавить другие: `users`, `tasks`, `cart`, и т. д.

2️⃣

```
import dishesReducer from "./components/dishes/dishesReducer";
```

- Импортируется редьюсер `dishesReducer`, который отвечает за состояние, связанное с блюдами (массив блюд, добавление, редактирование, удаление и т.д.).

🧠 В Redux редьюсер — это чистая функция:

```
(state, action) ⇒ newState
```

которая получает текущее состояние и объект `action`, а возвращает новое состояние.

3️⃣

```
const store = createStore(combineReducers(
   {
      dishes: dishesReducer
   }
));
```

🔷 Что здесь происходит:

1. `combineReducers({ dishes: dishesReducer })`
   - Создаёт **единый корневой редьюсер**, который объединяет все “под-редьюсеры”.
   - Ключ `dishes` определяет *имя ветки состояния*, под которой будет храниться состояние блюд.
   - В итоге структура состояния (`state`) будет выглядеть так:

     ```
     {
       dishes: [ /* массив блюд */ ]
     ```

```
}
```

- Если бы добавили ещё редьюсеры, например:

```
combineReducers({
  dishes: dishesReducer,
  users: usersReducer,
  cart: cartReducer
})
```

то структура `state` выглядела бы как:

```
{
  dishes: [...],
  users: {...},
  cart: {...}
}
```

2. **`createStore(...)`**
   - Получает результат работы `combineReducers` — то есть **общий редьюсер**, который умеет управлять всеми ветками состояния.
   - Возвращает объект `store`, который:
     - хранит всё состояние приложения;
     - предоставляет методы:
       - `getState()` — получить текущее состояние;
       - `dispatch(action)` — отправить действие (action) для обновления;
       - `subscribe(listener)` — подписаться на обновления состояния.
3. **Итог**

- Переменная `store` теперь содержит *единое хранилище Redux* для всего приложения.

4️⃣

```
export default store;
```

- Экспортируем `store`, чтобы использовать его в `main.tsx` (или `index.tsx`):

  ```
  <Provider store={store}>
    <App />
  </Provider>
  ```

  Это делает хранилище доступным для всех компонентов через хуки `useSelector()` и `useDispatch()`.

5️⃣

```
export type RootState = ReturnType<typeof store.getState>;
```

🔷 Что делает эта строка (TypeScript-магия):

- `store.getState` — это функция, возвращающая текущее состояние Redux Store.
- `ReturnType<typeof store.getState>` — извлекает тип возвращаемого значения этой функции.
- Таким образом, `RootState` становится типом, описывающим **всю структуру состояния приложения**.

📘 Пример:

```
// Автоматически типизируется как { dishes: Dish[] }
const dishes = useSelector((state: RootState) ⇒ state.dishes);
```

💡 Преимущество:

Если ты добавишь в combineReducers ещё редьюсер, тип `RootState` обновится автоматически.

🧩 Как всё работает в памяти

```
createStore(
   combineReducers({
      dishes: dishesReducer
   })
)
```

Создаёт цепочку:

```
Store
  └── Root Reducer (из combineReducers)
       └── dishesReducer
            └── управляет state.dishes
```

Когда компонент вызывает:

```
dispatch({ type: 'ADD_DISH', payload: {...} });
```

Redux делает:

1. Передаёт action в root reducer.
2. `combineReducers` вызывает `dishesReducer` для ветки `dishes`.
3. `dishesReducer` возвращает новое состояние.
4. Redux обновляет `store`.

5. Все компоненты, использующие `useSelector((state) => state.dishes)`, получают новое состояние.

✅ Итоговое понимание

Компонент кода	Что делает	Ключевые слова
`combineReducers()`	объединяет редьюсеры	rootReducer
`createStore()`	создаёт Store	state, dispatch, subscribe
`dishesReducer`	управляет одной частью состояния	slice reducer
`RootState`	описывает всю структуру Store	типизация

📘 Структура состояния после запуска

```
{
  dishes: [
    { id: 1, title: "Soup", price: 12 },
    { id: 2, title: "Salad", price: 9 }
  ]
}
```

⚛️ Подробный технический разбор `main.tsx` (чистый Redux)

📄 Код

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import { Provider } from 'react-redux';
import App from './App';
import store from './store';
```

```
ReactDOM.createRoot(document.getElementById('root')!).render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>
);
```

🔍 Построчное объяснение

```
import React from 'react';
import ReactDOM from 'react-dom/client';
```

◾ Что происходит

- Импортируются базовые модули React:
  - `React` — для поддержки JSX (хотя напрямую не используется, без него JSX не скомпилируется);
  - `ReactDOM` — отвечает за отрисовку компонентов React в HTML-документе.

📘 В React 18 используется новый API `createRoot()` (а не старый `ReactDOM.render()`), поэтому импорт идёт из `'react-dom/client'`.

```
import { Provider } from 'react-redux';
```

◾ Что делает `Provider`

- Это компонент из пакета `react-redux`, который **связывает Redux и React**.

- Он создаёт *контекст Redux* — невидимый “глобальный объект”, через который все дочерние компоненты могут обращаться к Store с помощью:
  - `useSelector()` — чтобы читать данные;
  - `useDispatch()` — чтобы отправлять действия (actions).

💡 Без `<Provider>` React-компоненты не смогут работать с Redux — вызовы `useSelector` и `useDispatch` вызовут ошибку.

```
import App from './App';
```

■ Что здесь

- Импортируем **корневой компонент** приложения.
- Внутри `App` находятся все остальные компоненты:

  `DishForm`, `DishesList`, `DishEditForm` и т.д.

📦 Этот компонент будет “входной точкой” интерфейса, куда передаются все контексты, в том числе Redux.

```
import store from './store';
```

■ Что делает

- Импортируем объект `store`, созданный в `store.ts`.
- Этот объект хранит состояние и редьюсер (`dishesReducer`).
- Мы передадим его в `Provider`, чтобы всё приложение имело к нему доступ.

```
ReactDOM.createRoot(document.getElementById('root')!).render(
```

■ Объяснение

- `ReactDOM.createRoot()` — метод React 18 для инициализации приложения.
- `document.getElementById('root')` ищет в `index.html` элемент:

```
<div id="root"></div>
```

- Символ `!` (non-null assertion) говорит TypeScript:

  “я точно знаю, что этот элемент существует, не проверяй на null”.
- Затем вызывается `.render(...)`, чтобы “вмонтировать” React-приложение внутрь этого `<div>`.

```
<React.StrictMode>
```

■ Что делает

- Включает режим строгой проверки в React:
  - помогает находить потенциальные ошибки и побочные эффекты;
  - в режиме разработки (`npm start`) вызывает компоненты дважды, чтобы проверить чистоту рендера;
  - в продакшене (`npm run build`) этот режим не влияет на работу приложения.

💡 Для учебных целей можно убрать StrictMode, чтобы избежать “двойного вызова” эффектов, который иногда путает студентов.

```
<Provider store={store}>
  <App />
</Provider>
```

■ Подробно

- Здесь происходит **главное подключение Redux к React**:
  1. `<Provider>` получает `store` как проп.
  2. Создаёт контекст (Context API).
  3. Все компоненты внутри `App` теперь могут:
     - читать данные: `useSelector((state) => state.dishes)`;
     - изменять их: `useDispatch()` → `dispatch({ type: 'dishes/create', payload: ... })`.

📘 Именно этот участок кода превращает “просто React” в “React + Redux”.

```
  </React.StrictMode>
);
```

- Закрываем строгий режим и заканчиваем вызов `.render()`.
- Приложение полностью загружено, Redux подключён, можно начинать работу с интерфейсом.

🧩 Как работает связка на практике

```
main.tsx
 │
 ├── <Provider store={store}>       ← подключает Redux ко всему приложе
нию
 │    └── <App />                  ← корневой компонент React
 │        ├── DishForm             ← добавляет блюда через dispatch()
 │        ├── DishesList           ← получает список через useSelector()
 │        └── DishEditForm         ← изменяет блюда через dispatch()
 │
 └── store.ts                      ← создаёт Store через createStore()
      │
      └── dishesReducer            ← управляет состоянием блюд
```

🧭 Что происходит при запуске

Этап	Описание	Что делает
1	React запускает `main.tsx`	Создаёт root и монтирует приложение
2	`<Provider store={store}>`	Подключает Redux через Context
3	Redux вызывает редьюсер	Получает `initialState`
4	Компоненты внутри `App` используют `useSelector`	Получают актуальное состояние
5	При действиях (dispatch)	Redux вызывает редьюсер и обновляет state
6	React видит изменение	Перерендеривает нужные компоненты

⚠️ Возможные ошибки

Ошибка	Причина	Решение
`Could not find react-redux context value`	`Provider` не оборачивает `<App />`	Обернуть App в `<Provider store={store}>`
`Cannot read properties of undefined (reading 'map')`	`useSelector` вернул `undefined` — редьюсер не подключён	Проверить, что `store` создан с правильным редьюсером
Ничего не рендерится	Неверный `id` в `getElementById('root')`	Убедиться, что `<div id="root"></div>` есть в `index.html`

✅ Итоговое понимание

- **React** отвечает за интерфейс.
- **Redux Store** хранит состояние.
- **Provider** делает Store доступным для всех компонентов.
- **useSelector** — “чтение” данных из Store.
- **useDispatch** — “запись” (отправка actions).

- Всё приложение связано одной цепочкой:

```
Компонент → dispatch(action)
      ↓
  dishesReducer(state, action)
      ↓
    Новый state
      ↓
 useSelector → обновление UI
```

🚀 Пошаговая последовательность создания проекта React + Redux (чистый Redux)

🔷 Этап 1. Создание проекта React

Команда:

```
npx create-react-app dishes --template typescript
```

🧩 Что происходит:

- `create-react-app` создаёт структуру проекта React с настройками Webpack, Babel и TypeScript.
- После завершения в папке `dishes` появятся стандартные файлы:

  `src/`, `public/`, `package.json`, `tsconfig.json`.

📘 **Важно:** шаблон TypeScript (`--template typescript`) автоматически добавит строгую типизацию, что помогает студентам видеть ошибки ещё при

написании кода.

🔷 Этап 2. Установка Redux и React-Redux

Команда:

```
npm install redux react-redux
```

🧩 Что происходит:

- `redux` — ядро системы управления состоянием.
- `react-redux` — “мост” между Redux и React, содержит `Provider`, `useDispatch`, `useSelector`.

💡 В этом курсе используется **чистый Redux**, поэтому **Redux Toolkit не устанавливается**.

🔷 Этап 3. Создание структуры проекта

Создай папки:

```
src/
 ├── components/
 │    ├── DishForm.tsx
 │    ├── DishEditForm.tsx
 │    └── DishesList.tsx
 ├── features/
 │    └── dishes/
 │         ├── dishesReducer.ts
 │         └── actions.ts
 ├── store.ts
 └── App.tsx
```

💡 Комментарий:

Такое разделение отражает “архитектуру по фичам” (feature-based structure)
—

каждый функциональный блок (например, `dishes`) имеет свой редьюсер, экшены и типы данных.

🔷 Этап 4. Написание редьюсера (`dishesReducer.ts`)

Пример базового редьюсера:

```
import { Dish } from '../../types/Dish';
import { CREATE_DISH, DELETE_DISH, UPDATE_DISH } from './actions';

const initialState: Dish[] = [];

const dishesReducer = (state = initialState, action: any): Dish[] => {
  switch (action.type) {
    case CREATE_DISH:
      return [...state, action.payload];
    case DELETE_DISH:
      return state.filter(dish => dish.id !== action.payload);
    case UPDATE_DISH:
      return state.map(dish =>
        dish.id === action.payload.id ? action.payload : dish
      );
    default:
      return state;
  }
};

export default dishesReducer;
```

🔍 Что здесь важно:

- **Редьюсер** — чистая функция:

  получает текущее состояние (`state`) и действие (`action`) → возвращает новый `state`.

- **Нельзя** мутировать состояние напрямую (`push`, `splice`),

  всегда возвращается новый массив.

- Redux использует этот редьюсер, чтобы обновлять состояние при `dispatch()`.

🔷 Этап 5. Создание Store (`store.ts`)

```
import { createStore } from 'redux';
import dishesReducer from './features/dishes/dishesReducer';

const store = createStore(dishesReducer);

export default store;
```

🔍 Что делает:

1. Импортирует `createStore` — функцию ядра Redux.
2. Подключает `dishesReducer` как “главного управляющего состоянием”.
3. `createStore()` создаёт объект Store с методами `getState()`, `dispatch()`, `subscribe()`.

💡 После этой строки у нас уже есть централизованное хранилище, которое знает, какие блюда есть в приложении.

🔷 Этап 6. Подключение Store к React через Provider (`main.tsx`)

```
import React from 'react';
import ReactDOM from 'react-dom/client';
```

```
import { Provider } from 'react-redux';
import App from './App';
import store from './store';

ReactDOM.createRoot(document.getElementById('root')!).render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>
);
```

🔍 Что делает:

- Компонент `<Provider>` передаёт `store` всему приложению через контекст.
- Теперь каждый компонент внутри `<App />` может использовать Redux:
  - `useSelector((state) => state)` — читать данные;
  - `useDispatch()` — отправлять actions.
- Без Provider Redux просто не будет работать в React.

🔷 Этап 7. Создание Action Creators (`actions.ts`)

```
export const CREATE_DISH = 'CREATE_DISH';
export const DELETE_DISH = 'DELETE_DISH';
export const UPDATE_DISH = 'UPDATE_DISH';

export const createDish = (dish) => ({ type: CREATE_DISH, payload: dish });
export const deleteDish = (id) => ({ type: DELETE_DISH, payload: id });
export const updateDish = (dish) => ({ type: UPDATE_DISH, payload: dish });
```

🔍 Что делает:

- Определяет **типы действий** (`type`) и **создателей действий** (action creators).
- Это стандарт Redux — каждое изменение состояния описывается объектом:

```
{ type: 'CREATE_DISH', payload: {...} }
```

- Эти объекты будут передаваться в `dispatch()` из компонентов.

🔷 Этап 8. Создание компонентов

🥣 `DishForm.tsx` — форма добавления блюда

Использует `useDispatch()` для создания нового блюда.

✏️ `DishEditForm.tsx` — форма редактирования

Получает данные блюда, редактирует и вызывает `dispatch(updateDish(...))`.

📋 `DishesList.tsx` — список всех блюд

Использует `useSelector()` для отображения состояния из Store.

💡 Таким образом, у нас получается полная цепочка:

```
Компонент → dispatch(action) → reducer → обновление store → useSelector → обновление UI
```

🔷 Этап 9. Проверка работы в DevTools

Шаги:

1. Установи расширение **Redux DevTools** в браузер.
2. Включи приложение и добавь блюдо.
3. Открой Redux DevTools → вкладка “Actions”.

Ты увидишь:

```
{ type: "CREATE_DISH", payload: { title: "Soup", price: 10, ... } }
```

и в State появится новое блюдо.

🔷 Этап 10. Закрепление: основная логика Redux

Команда	Назначение
`createStore(reducer)`	создаёт хранилище
`store.dispatch(action)`	отправляет действие
`reducer(state, action)`	возвращает новый state
`store.getState()`	получает текущее состояние
`<Provider store={store}>`	делает store доступным компонентам
`useSelector(fn)`	достаёт данные из store
`useDispatch()`	возвращает функцию для вызова `dispatch()`

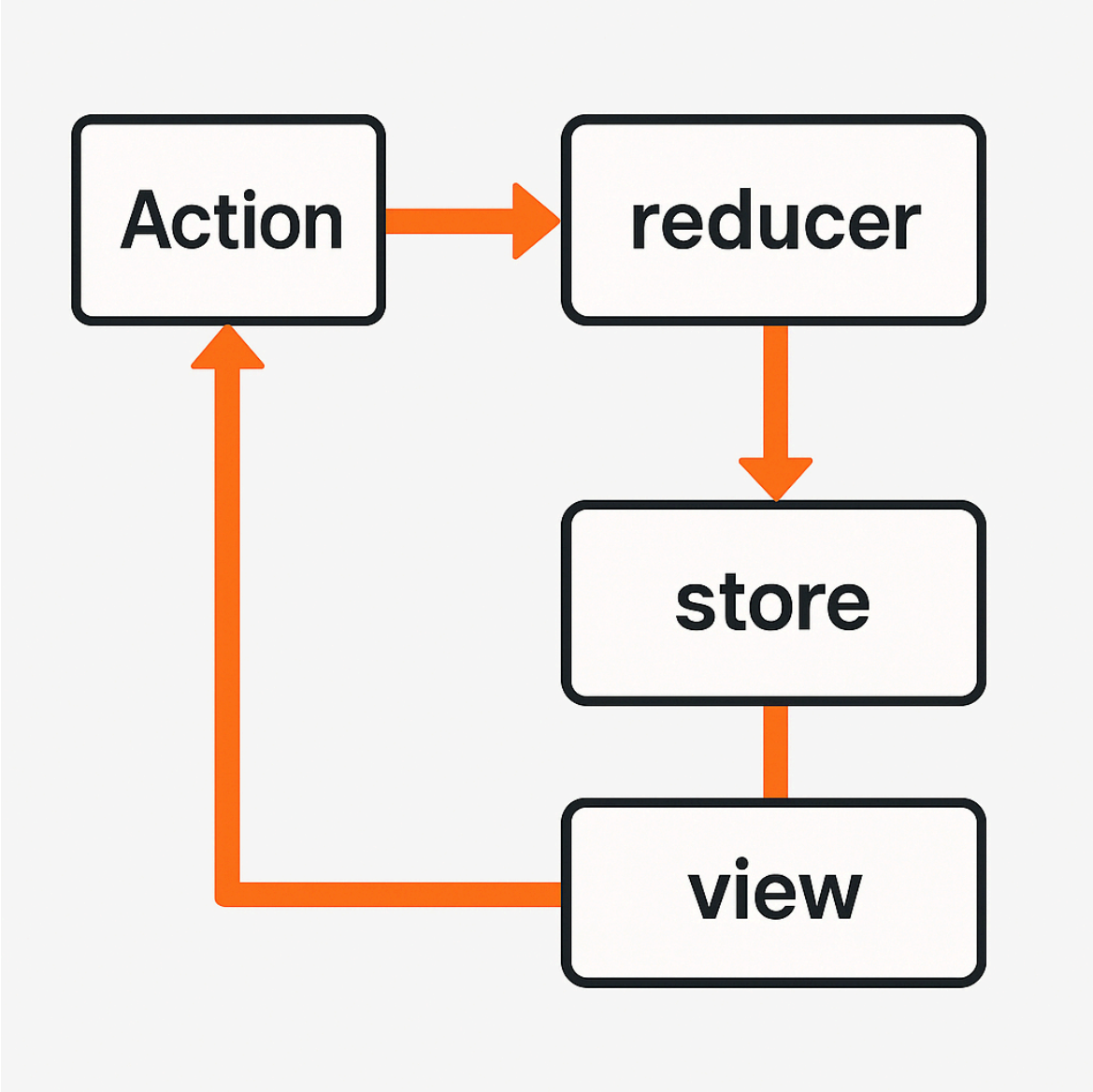
🧭 Цепочка данных в приложении

```
DishForm (useDispatch)
    ↓ dispatch(action)
dishesReducer(state, action)
    ↓ возвращает новый state
store обновляется
    ↓
DishesList (useSelector)
    ↓
React перерисовывает компонент
```

✅ Итог

После выполнения всех шагов студент должен:

1. Понимать, что такое Store и Reducer.
2. Уметь отправлять actions и читать state.
3. Разбираться, как React взаимодействует с Redux через Provider.
4. Видеть, как изменения данных автоматически обновляют интерфейс.



🔄 Схема жизненного цикла Redux

🧭 Общее описание

Эта диаграмма показывает **основной цикл данных** в приложении React + Redux.

Все обновления состояния (state) проходят по одному направлению — **односторонний поток данных**.

Такое поведение делает код предсказуемым, а логику — легко отлаживаемой.

🧩 Этап 1. React-компонент (View)

🔷 Компоненты — это **интерфейс** (UI), где пользователь вводит данные или выполняет действия.

Например, в `DishForm.tsx` пользователь нажимает кнопку “Создать блюдо”.

📘 Когда это происходит, компонент вызывает функцию:

```
dispatch({ type: 'CREATE_DISH', payload: {...} });
```

🔶 Таким образом, компонент “запускает” процесс обновления данных, но **сам ничего не меняет напрямую**.

⚙️ Этап 2. Action (действие)

🔷 **Action** — это обычный объект с двумя свойствами:

```
{ type: 'CREATE_DISH', payload: { title: 'Soup', price: 12 } }
```

- `type` — тип действия (определяет, что происходит: создание, удаление, обновление);

- `payload` — данные, которые нужны редьюсеру.

📘 Action описывает “намерение”, но не знает, *как именно* состояние изменится — этим занимается редьюсер.

🧠 Этап 3. Reducer (редьюсер)

🔷 **Reducer** получает текущее состояние и действие, и возвращает новое состояние:

```
function dishesReducer(state, action) {
  switch (action.type) {
    case 'CREATE_DISH':
      return [...state, action.payload];
    default:
      return state;
  }
}
```

🧩 Это чистая функция:

- не изменяет исходное состояние напрямую;
- не имеет побочных эффектов;
- возвращает *новый* объект или массив.

💡 Каждый `dispatch()` вызывает редьюсер → Redux создаёт новое состояние.

💾 Этап 4. Store (хранилище)

🔷 **Store** — это “центр Redux-вселенной”.

Он содержит текущее состояние всего приложения.

Когда редьюсер возвращает новое состояние:

- Store обновляет свои данные;
- уведомляет все подписанные компоненты (`useSelector`).

📘 В чистом Redux Store создаётся через:

```
const store = createStore(combineReducers(
   {
      dishes: dishesReducer
   }
));
```

👁️ Этап 5. useSelector → UI обновляется

🔷 Компоненты, использующие `useSelector`, автоматически “подписаны” на обновления Store.

Например:

```
const dishes = useSelector((state) ⇒ state);
```

Как только Store обновляется, React вызывает повторный рендер компонента, и пользователь сразу видит изменения в интерфейсе.

✅ Теперь данные на экране соответствуют текущему состоянию Store.

🔁 Итоговый цикл данных

№	Этап	Действие
1	**Пользователь** взаимодействует с компонентом	Нажимает кнопку, вводит данные
2	**Компонент** вызывает `dispatch(action)`	Отправляет действие в Redux
3	**Reducer** получает `state` и `action`	Возвращает новое состояние
4	**Store** сохраняет это состояние	Обновляет глобальное хранилище
5	**React-компоненты** с `useSelector`	Получают новое состояние и перерисовываются

💬 Главное правило Redux:

> Состояние в Redux изменяется только через actions и reducer.
>
> Компоненты не меняют данные напрямую, они лишь “запрашивают изменения”.

🧠 Ассоциация для запоминания

Представь, что Redux — это ресторан:

Элемент Redux	Аналогия	Роль
Компонент (DishForm)	Официант	Принимает заказ
Action	Бланк заказа	Передаёт, что нужно приготовить
Reducer	Повар	Готовит блюдо (новое состояние)
Store	Кухня	Хранит все готовые блюда
useSelector	Клиент	Видит обновлённое меню