

Лекция 12

Axios. Бизнес-процесс

Разработка интернет приложений

Канев Антон Игоревич

AJAX

- **AJAX**, Ajax (Asynchronous Javascript and XML — «асинхронный JavaScript и XML») — подход к построению интерактивных пользовательских интерфейсов веб-приложений, заключающийся в «фоновом» обмене данными браузера с веб-сервером.
- В результате при обновлении данных веб-страница не перезагружается полностью, и веб-приложения становятся быстрее и удобнее.
- **JSON-RPC** (JavaScript Object Notation Remote Procedure Call — JSON-вызов удалённых процедур) — протокол удалённого вызова процедур, использующий JSON для кодирования сообщений.

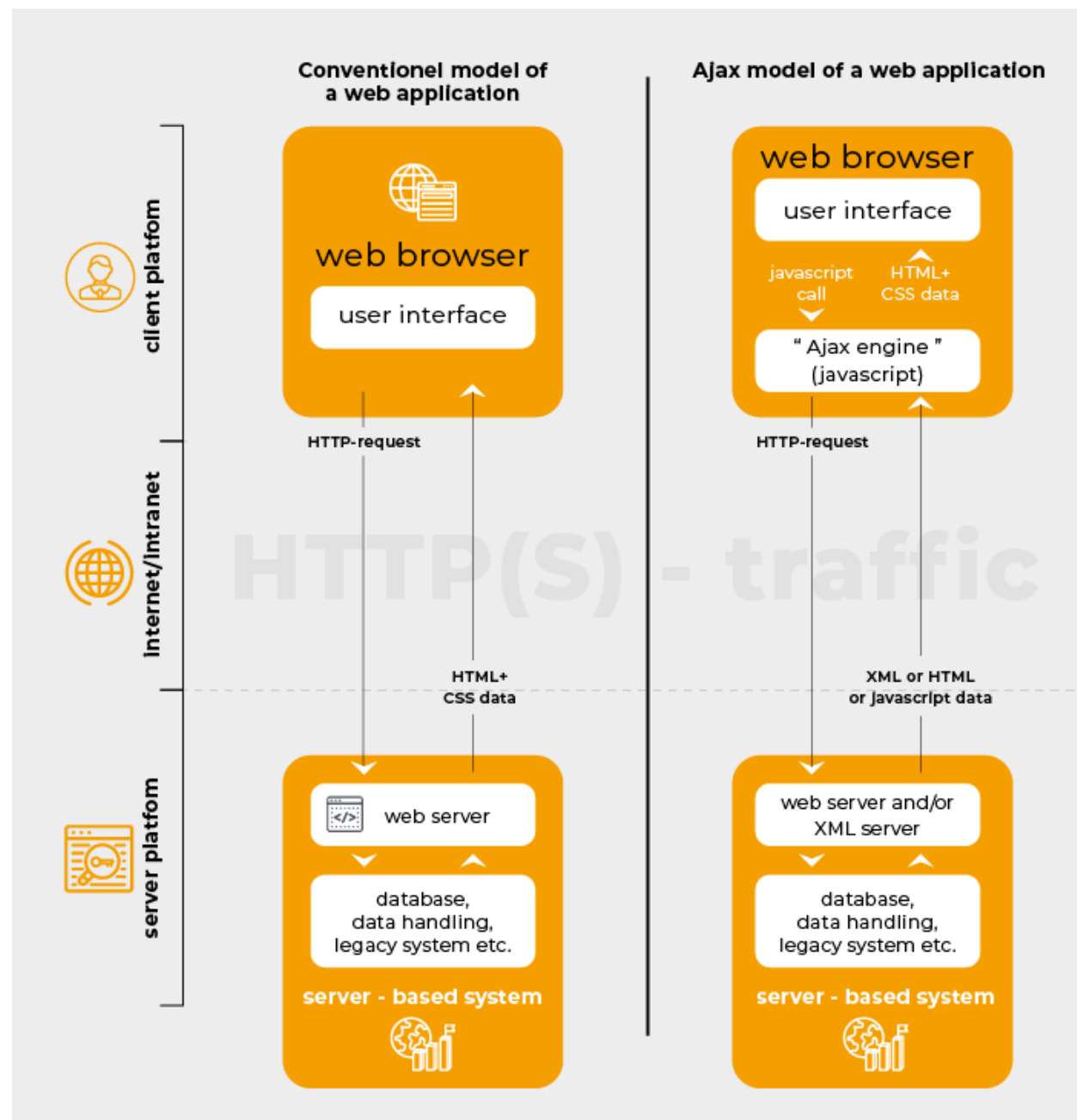
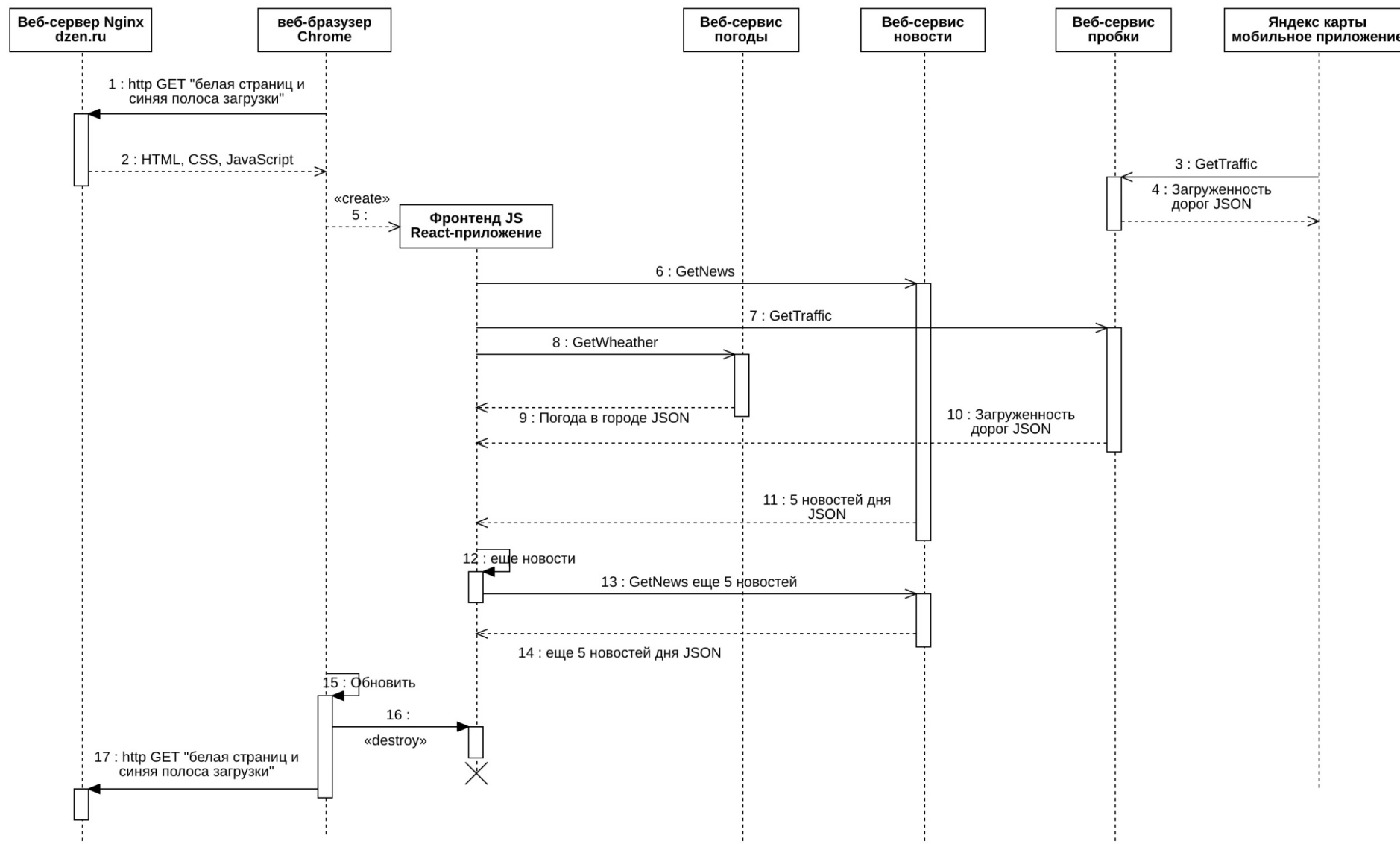


Диаграмма последовательности с AJAX

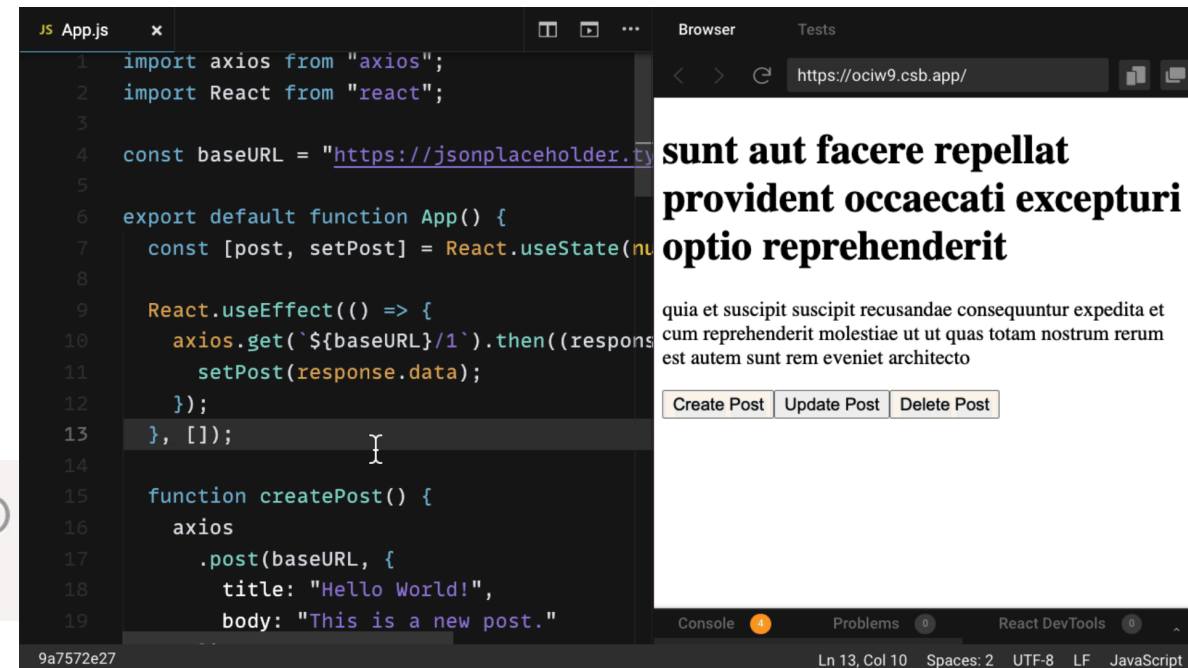


Axios vs fetch

- Fetch — нативный низкоуровневый JavaScript интерфейс для выполнения HTTP-запросов с использованием обещаний через глобальный метод `fetch()`.
- Axios — JavaScript-библиотека, основанная на обещаниях, для выполнения HTTP-запросов.

```
fetch('https://jsonplaceholder.typicode.com/posts/1')  
  .then(response => response.json())  
  .then(json => console.log(json))
```

```
axios.get('https://jsonplaceholder.typicode.com/posts/1')  
  .then(response => console.log(response));
```



Обработка ошибок

- Axios обрабатывает ошибки логично.
- Если сервер вернул ответ с HTTP статусом ошибки (например 404 или 500), то обещание будет отвергнуто.

```
fetch(url)
  .then(response => {
    return response.json().then(data => {
      if (response.ok) {
        return data;
      } else {
        return Promise.reject({status: response.status, data});
      }
    });
  })
  .then(result => console.log('success:', result))
  .catch(error => console.log('error:', error));
```

```
axios.get('https://jsonplaceholder.typicode.com/posts/1')
  .then(response => console.log(response));
```

POST

- С axios всё просто, а с fetch уже не так:
- JSON обязан быть преобразован в строку, а заголовок Content-Type должен указывать, что отправляются JSON данные,
- иначе сервер будет рассматривать их как строку.

```
axios.post('/user', {  
  firstName: 'Fred',  
  lastName: 'Flintstone'  
});
```

```
fetch('/user', {  
  method: 'POST',  
  headers: {  
    'Content-Type': 'application/json'  
  },  
  body: JSON.stringify({  
    firstName: 'Fred',  
    lastName: 'Flintstone'  
  })  
});
```

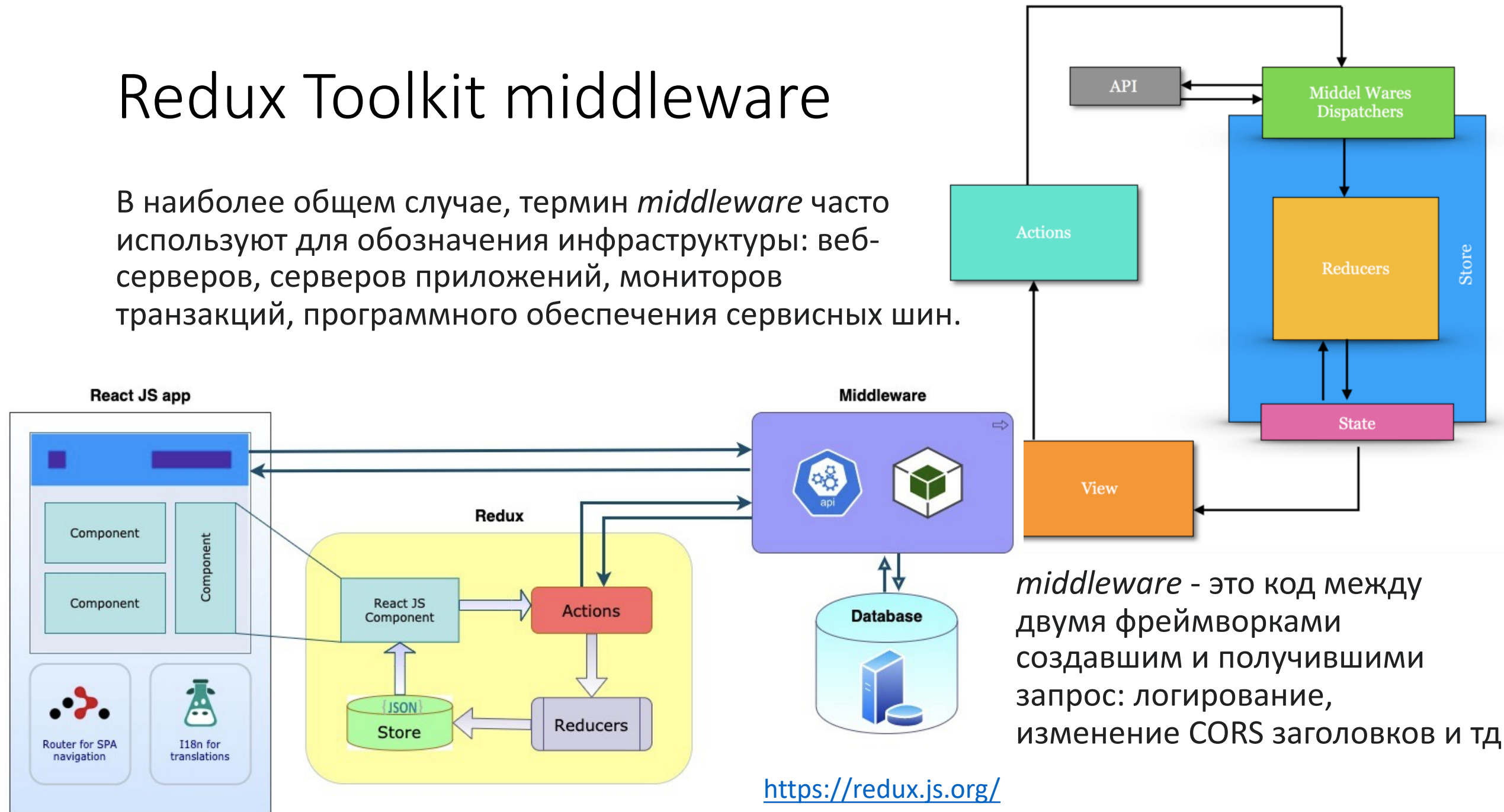
Базовые значения для запросов

- fetch это явный API, вы ничего не получаете, если об этом не просите.
- Если используется аутентификация, основанная на сохранении сессии пользователя, то надо явно указывать куку.
- Если сервер расположен на поддомене, то надо явно прописывать CORS.
- Эти опции надо прописывать для всех вызовов сервера и у fetch нет механизма для установки значений по-умолчанию, а у axios есть.

```
axios.defaults.baseURL = 'https://api.example.com';  
axios.defaults.headers.common['Accept'] = 'application/json';  
axios.defaults.headers.post['Content-Type'] = 'application/json';
```

Redux Toolkit middleware

В наиболее общем случае, термин *middleware* часто используют для обозначения инфраструктуры: веб-серверов, серверов приложений, мониторов транзакций, программного обеспечения сервисных шин.



Redux-middleware

- **Мидлвары** (middlewares) — это функции, которые последовательно вызываются в процессе обновления данных в хранилище.

Мидлвары используются в задачах:

- Логирование
- Оповещение об ошибках
- Работа с асинхронным API
- Маршрутизация

```
const logger = store => next => action => {  
  let result;  
  console.groupCollapsed("dispatching", action.type);  
  console.log("prev state", store.getState());  
  console.log("action", action);  
  result = next(action);  
  console.log("next state", store.getState());  
  console.groupEnd();  
  return result;  
};
```

```
const middleware = applyMiddleware(logger);  
const store = createStore(reducers, middleware);
```

Redux Toolkit fetch

- Ранее мы уже рассмотрели простой вариант создания action с данными, полученными из API
- Таким образом Redux и обращение к API непосредственно друг с другом не связаны. Их объединяет обработчик события

```
initialState: {  
  loading: 'idle',  
  users: [],  
},  
reducers: {  
  usersLoading(state, action) {  
    // Используем подход "state machine"  
    if (state.loading === 'idle') {  
      state.loading = 'pending';  
    }  
  },  
  usersReceived(state, action) {  
    if (state.loading === 'pending') {  
      state.loading = 'idle';  
      state.users = action.payload;  
    }  
  },  
},
```

```
// Деструктурируем и экспортируем обычных создателей  
export const {  
  usersLoading,  
  usersReceived,  
} = usersSlice.actions;
```

```
// Определяем `thunk`, отправляющего создателей  
const fetchUsers = () => async (dispatch) => {  
  dispatch(usersLoading());  
  const response = await usersAPI.fetchAll();  
  dispatch(usersReceived(response.data));  
};
```

Redux Thunk middleware

- Рассмотрим пример обращения к API через thunk
- В таком примере само обращение к API у нас скрыто в action
- В коде обработчика мы просто создаем действие с нужными параметрами, а средствами redux toolkit выполняется запрос и заполняется payload

<https://github.com/reduxjs/redux-thunk>

- Обратите внимание на код справа: для выполнения запроса используется сгенерированный на основе swagger код, который мы импортируем из userAPI

```
import {
  createAsyncThunk,
  createSlice,
} from '@redux/toolkit';
import { userAPI } from '../userAPI';

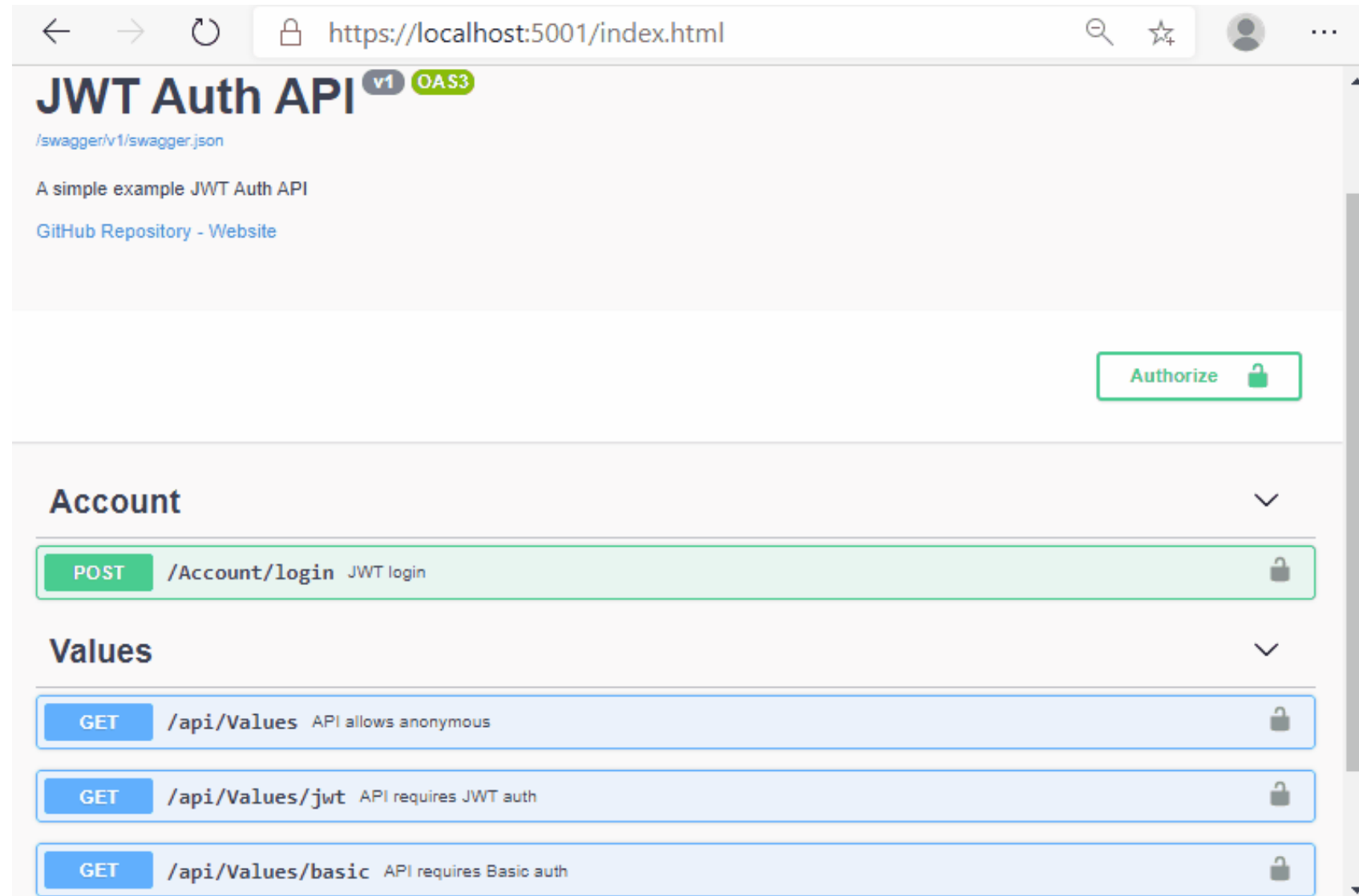
// Создаем преобразователя
const fetchUserById = createAsyncThunk(
  'user/fetchByIdStatus',
  async (userId, thunkAPI) => {
    const response = await userAPI.fetchById(userId);
    return response.data;
  }
);

// Обрабатываем операции в редукторах
const usersSlice = createSlice({
  name: 'users',
  initialState: { entries: [], loading: 'idle' },
  reducers: {
    // стандартная логика редуктора с авто-генерируемыми
  },
  extraReducers: {
    [fetchUserById.fullfilled]: (state, action) => {
      // Добавляем пользователя в массив состояния
      state.entries.push(action.payload);
    },
  },
});

// Позже, отправляем `thunk`
dispatch(fetchUserById(123));
```

Swagger

- Swagger – это фреймворк для спецификации RESTful API.
- Swagger UI позволяет интерактивно просматривать спецификацию и отправлять запросы
- Полученное на бэке описание можно использовать для генерации кода фронтенда



Кодогенерация для фронтенда из swagger

- У нас есть файл от Django по ссылке <http://127.0.0.1:8000/swagger/?format=openapi>
- Для кодогенерации скачаем библиотеку swagger-typescript-api
`npm i --save-dev swagger-typescript-api`
- Добавим в package.json команду

```
"scripts": {  
  "generate-backend-types": "swagger-typescript-api -p http://127.0.0.1:8000/swagger/?format=openapi --no-client -o ./types/autogenerated -n backend.ts" }
```
- Выполним
`npm run generate-backend-types`
- Получим в папке types/autogenerated файл backend.ts внутри которого будут сгенерированы интерфейсы повторяющие сущности из бэкенда
- Можно сгенерировать сразу методы обращения к API. Нужно их использовать в коде TS

Пример кодогенерации

- В примере FSD при генерации мы получаем методы для выполнения наших запросов к API
- Далее в коде наших обработчиков событий или thunk мы используем эти методы

```
import { createAsyncThunk } from '@reduxjs/toolkit';
```

```
import { api } from '@api';
```

```
import type { LoginData, LoginBody } from './fetchLogin.types';
```

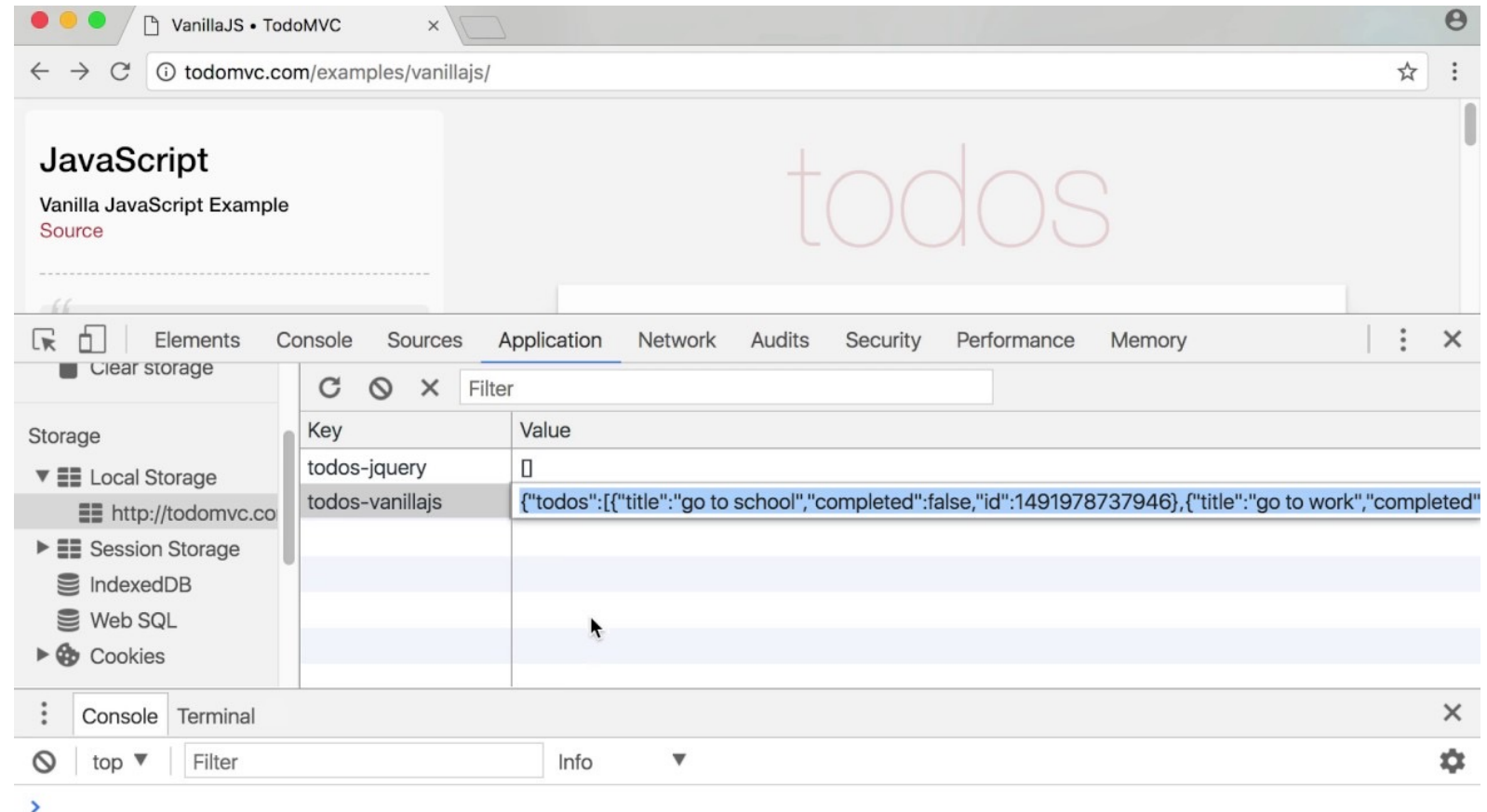
```
export const fetchLogin = createAsyncThunk<LoginData, LoginBody>('login/fetchLogin', async (body) => {  
  const { data } = await api.authenticate.authenticateCreate(body);
```

```
  return data.role;  
});
```

```
export class Api<SecurityDataType extends unknown> extends HttpClient<SecurityDataType> {  
  authenticate = {  
    /**  
     * @description Login  
     *  
     * @tags authenticate  
     * @name AuthenticateCreate  
     * @request POST:/authenticate/  
     * @secure  
     */  
    authenticateCreate: (data: Login, params: RequestParams = {}) =>  
      this.request<UserSwagger, any>({  
        path: `/authenticate/`,  
        method: 'POST',  
        body: data,  
        secure: true,  
        type: ContentType.Json,  
        format: 'json',  
        ...params,  
      }),  
  };  
}
```

Вкладка Application. Local Storage

- Что делать с данными приложения если нажали F5?
- В варианте с JWT мы используем Local Storage (локальное хранилище) для хранения JWT
- **Не используем** Local Storage в нашем курсе кроме как для хранения JWT

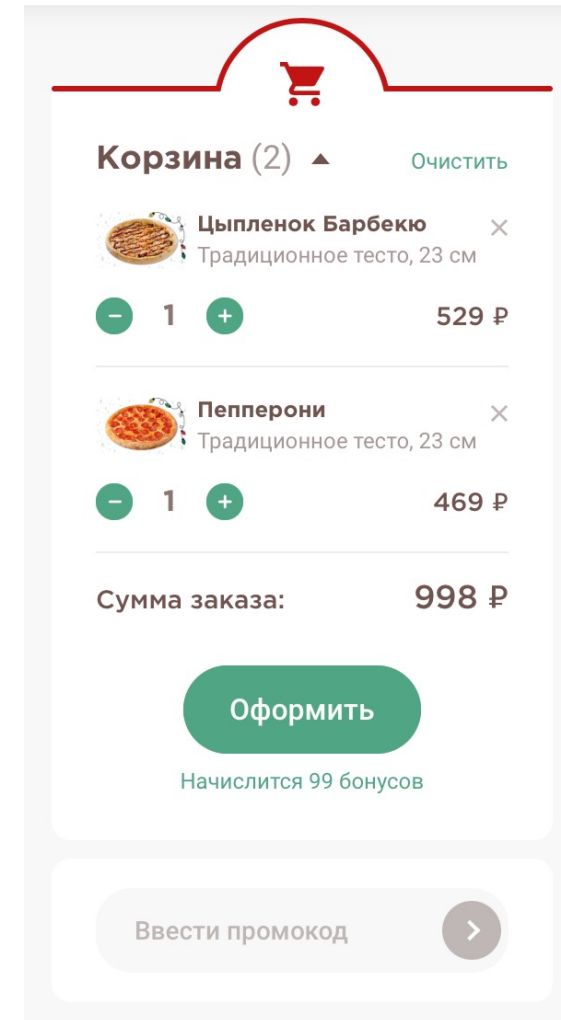
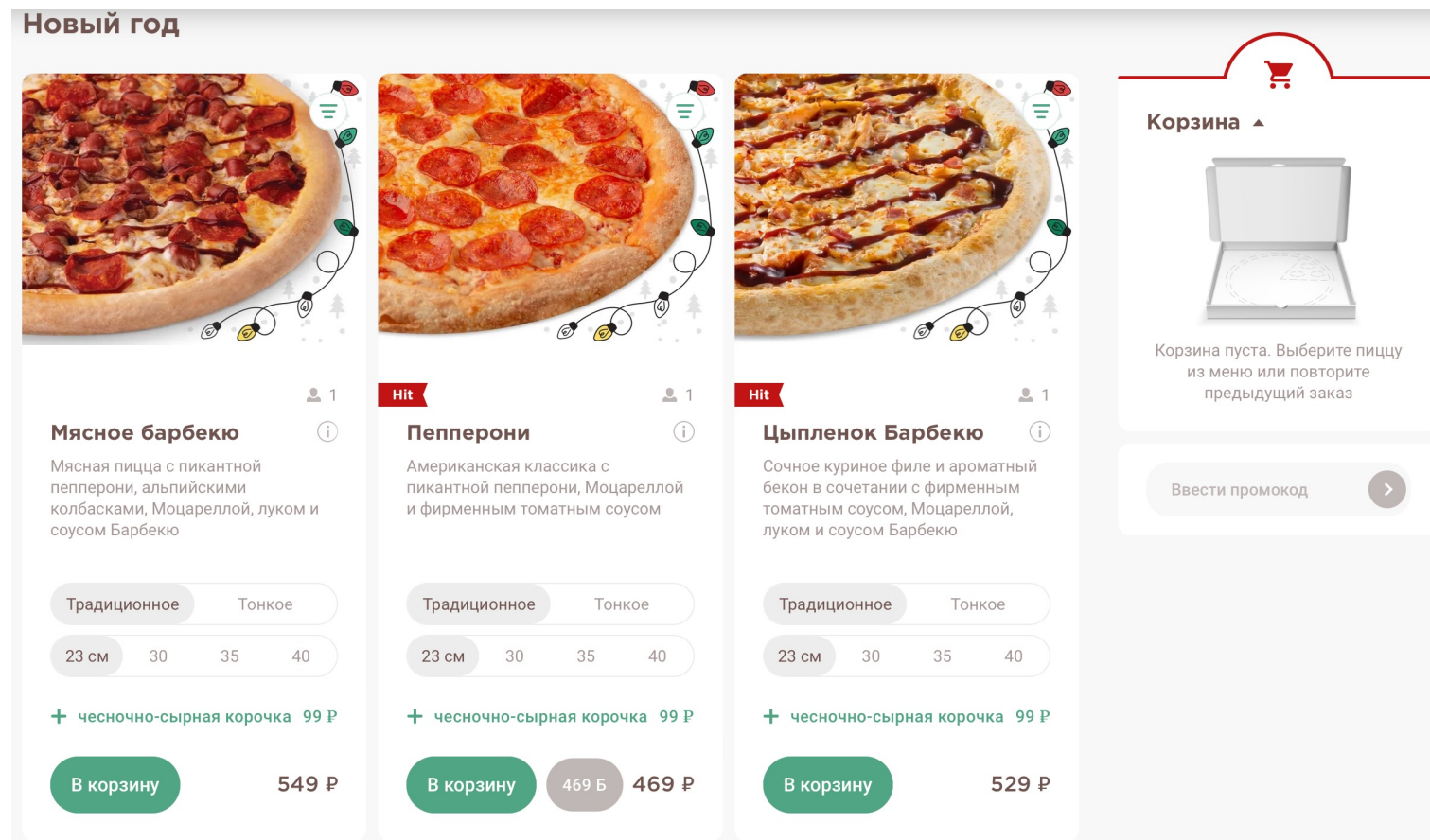


```
const storedSession = localStorage.getItem("session");

localStorage.setItem("session", JSON.stringify(state));
localStorage.removeItem("session");
```


Заявка-черновик

- Страница списка услуг включает сам список услуг и кнопку перехода в «корзину» - на страницу с заявкой-черновиком



- При нажатии на кнопку «Корзины» мы открываем страницу конструктора заявки
- Там кнопки Сформировать, Удалить, удал./ред. м-м

Диаграмма деятельности

- Также в 7 лабораторной нужно описать бизнес процесс для ура-сценария в вашей системе
- Его описываете на диаграмме деятельности или в BPMN 2.0
- У вас будет 2-3 дорожки: создатель, модератор, выделенный сервис (например оплата). Дорожки называйте по вашей теме

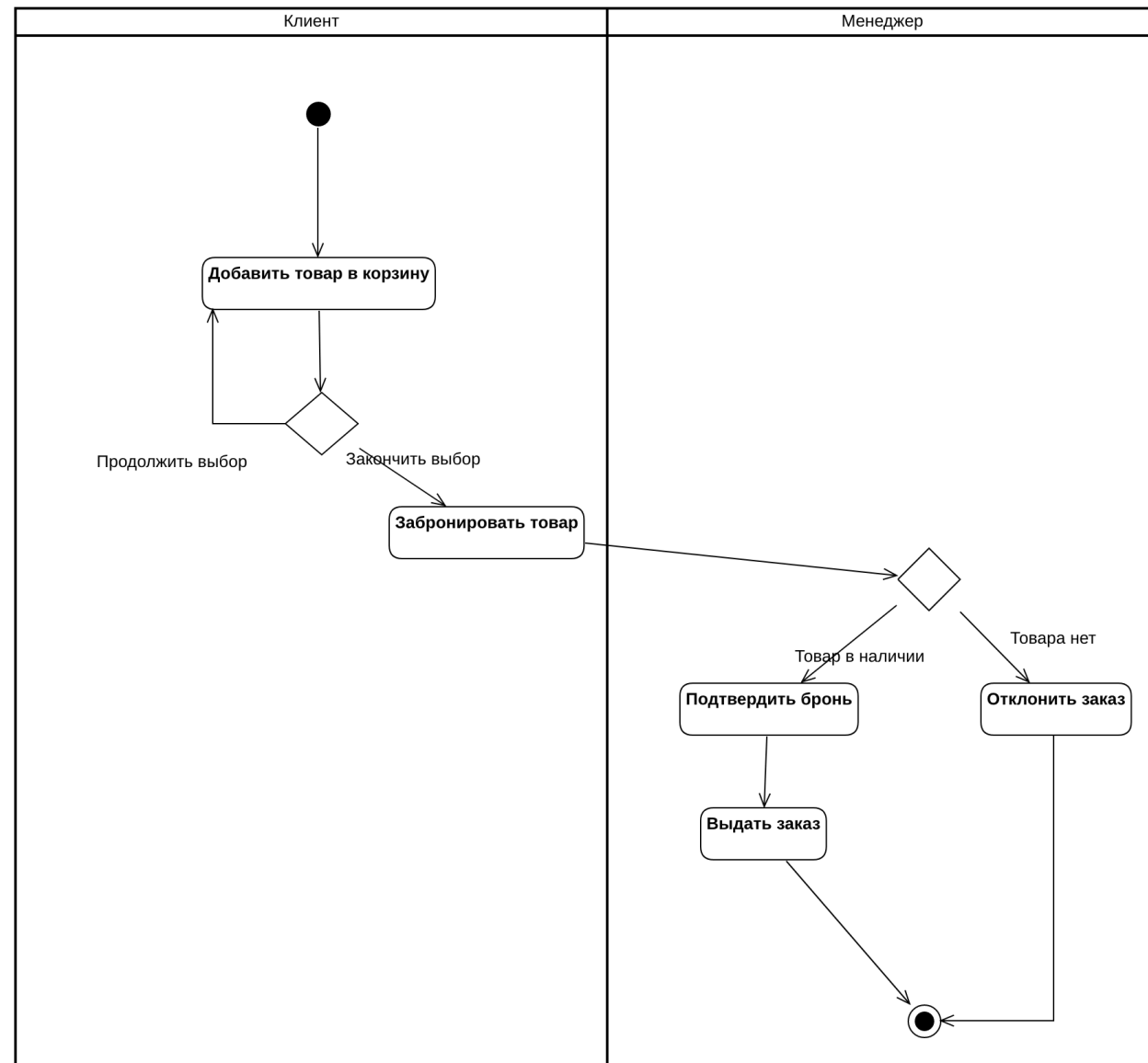


Диаграмма состояний

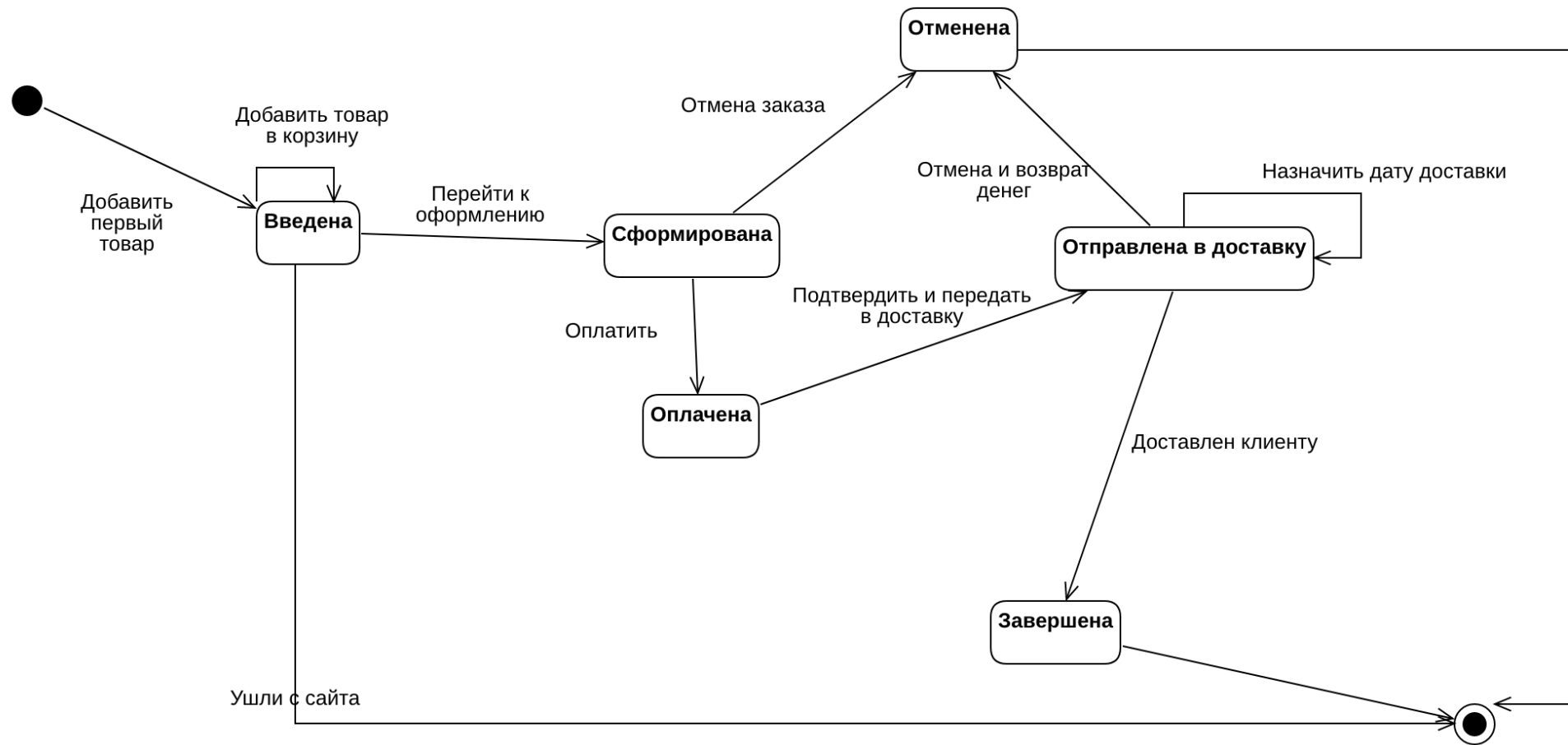
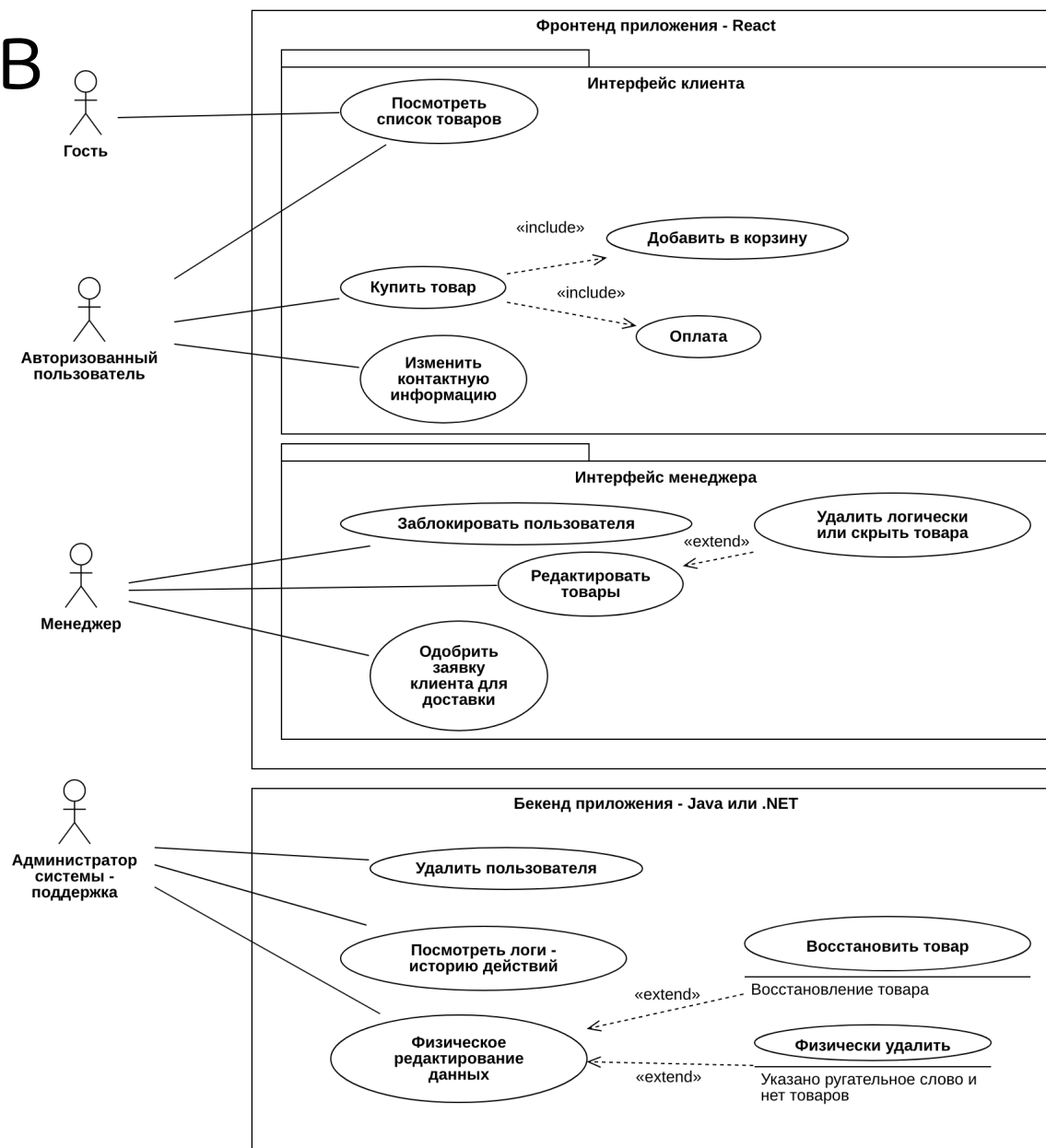


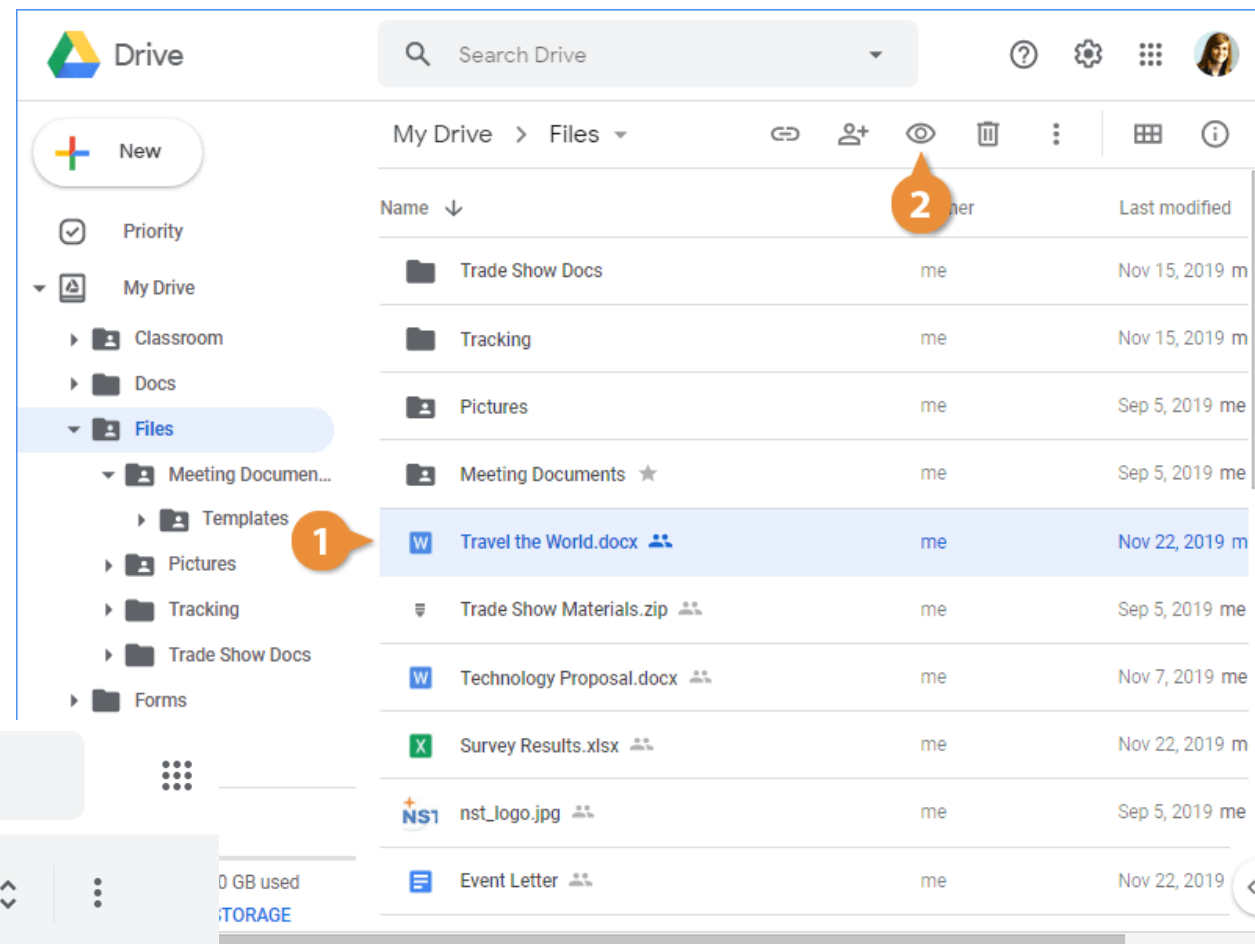
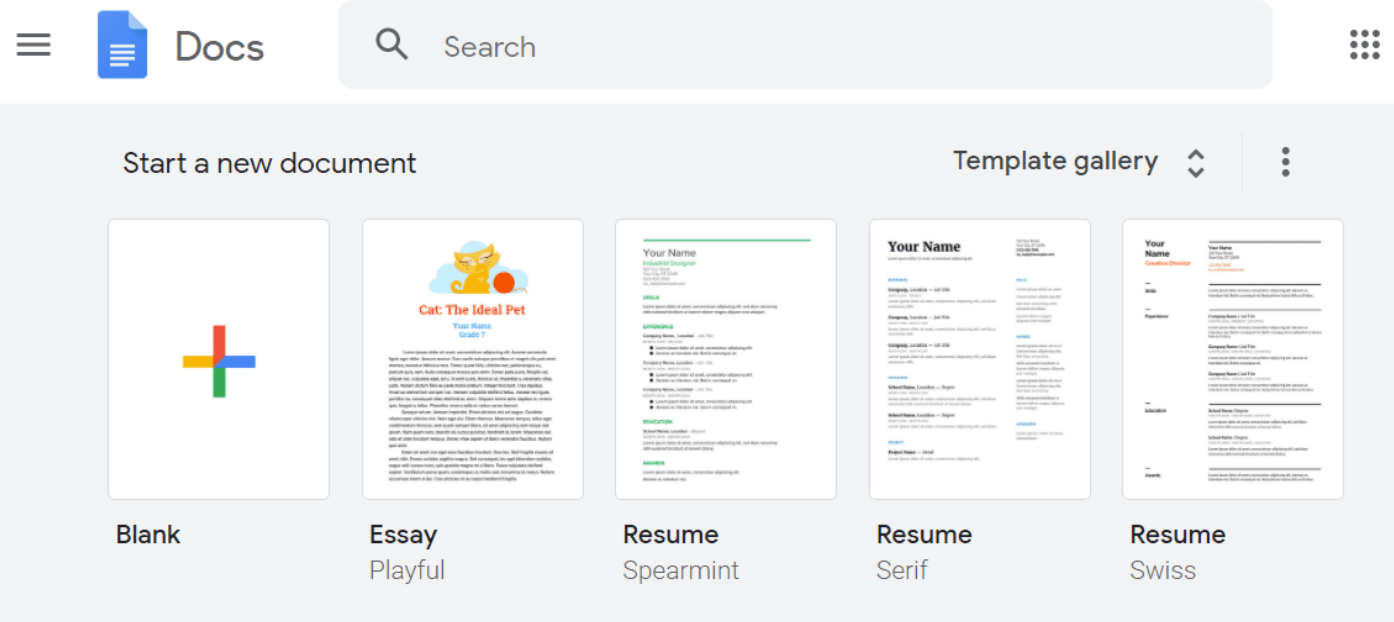
Диаграмма прецедентов

- В отчет вы также добавляете диаграмму прецедентов и диаграмму состояний
- На диаграмме прецедентов указываете роли пользователей вашей системы и действия, которые они могут в ней выполнить, как в функциональных требованиях
- Также требуется обновить и исправить все старые диаграммы и включить их в РПЗ



Два списка услуг

- Ваша страница списка услуг для гостя и создателя заявки представлена карточками
- Тут вы добавляете виджет корзины и кнопку добавления услуги в заявку



- Но список услуг для редактирования модератором должен быть в виде таблицы
- Тут будут кнопки добавления, редактирования и удаления