1. ГЛОБАЛЬНАЯ СХЕМА ПРИЛОЖЕНИЯ

```
┌───────────────┐
│ Browser    │
│ React SPA   │
└───────┬───────┘
      │ HTTP (Axios)
      ▼
┌──────────────────────────┐
│ API Gateway (Spring) │
│ DispatcherServlet    │
└───────┬──────────────────┘
      │
      ▼
┌──────────────────────────────────────────┐
│ SECURITY LAYER (Filter Chain)       │
│ - CORS                              │
│ - JwtAuthFilter                     │
│ - SecurityContext                  │
└───────┬──────────────────────────────────┘
      │
      ▼
┌──────────────────────────────────────────┐
│ CONTROLLER LAYER                    │
│ (@RestController)                   │
└───────┬──────────────────────────────────┘
      │
      ▼
┌──────────────────────────────────────────┐
│ SERVICE LAYER                       │
│ (Business logic, rules, transactions) │
└───────┬──────────────────────────────────┘
      │
      ▼
┌──────────────────────────────────────────┐
│ REPOSITORY LAYER (JPA / Hibernate)  │
└───────┬──────────────────────────────────┘
      │
      ▼
┌──────────────────────────┐
│ MySQL DB   │
└──────────────────────────┘
```

2. ПОЛНЫЙ ЖИЗНЕННЫЙ ЦИКЛ ЗАПРОСА

Разберём реальный пример: GET /api/projects/1

ЭТАП 1. FRONTEND (React)

Что происходит:

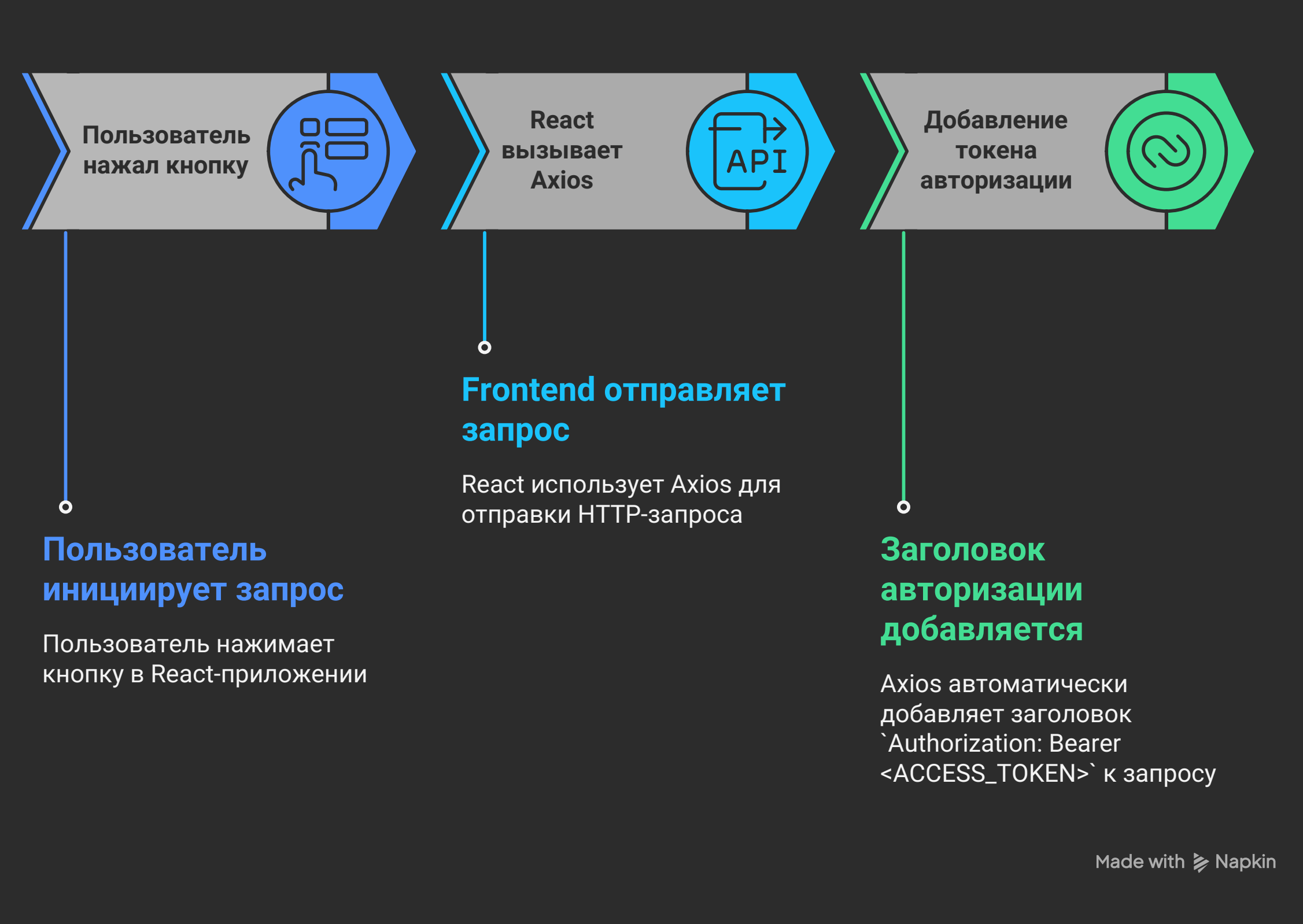
- Пользователь нажал кнопку
- React вызывает Axios
- В headers автоматически добавляется:

```
Authorization: Bearer <ACCESS_TOKEN>
```

Код:

```
request("GET", "/api/projects/1")
```

Этап 1: Frontend Request

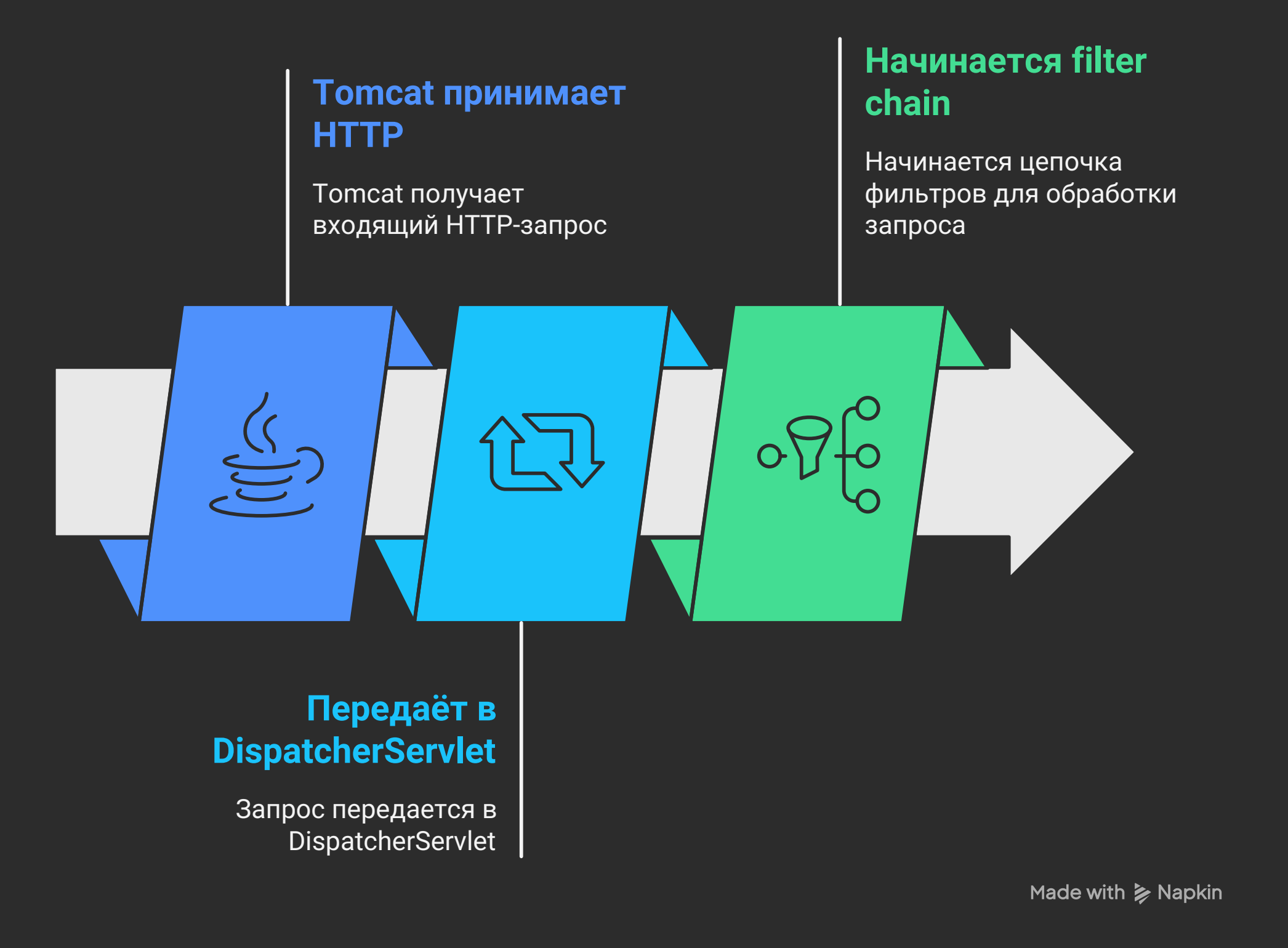


ЭТАП 2. DISPATCHER SERVLET (Spring)

Что происходит:

- Tomcat принимает HTTP
- Передаёт в DispatcherServlet
- Начинается filter chain

Этап 2: Dispatcher Servlet



ЭТАП 3. SECURITY FILTER CHAIN (КРИТИЧЕСКИЙ)

3.1 CORS Filter

✓ Проверка origin ✓ OPTIONS запросы ✗ Если не проходит — 403 ДО контроллера

3.2 JwtAuthFilter

Что делает:

1. Читает Authorization header
2. Проверяет Bearer
3. Декодирует JWT
4. Достаёт sub
5. Загружает пользователя из БД
6. Проверяет:
    - isActive
    - isBlocked
    - isAccountExpired
7. Создаёт Authentication
8. Кладёт в SecurityContext

Если ошибка →

```
✗ 401 Unauthorized
```

3.3 SecurityContext

Результат:

```
SecurityContextHolder:
  principal = CrmUserauthorities = ROLE_ADMIN
```

ЭТАП 4. AUTHORIZATION (Spring Security)

4.1 URL rules (HttpSecurity)

```
/api/projects/** → authenticated
```

✓ Проверка пройдена

4.2 Method-level security

```
@PreAuthorize("hasRole('USER')")
```

✓ ADMIN подходит (через role hierarchy)

ЭТАП 5. CONTROLLER

```
@GetMapping("/api/projects/{id}")public ProjectDto get(@PathVariable Long id) {
```

Контроллер:

- НЕ думает о безопасности
- НЕ лезет в БД напрямую
- Делегирует сервису

ЭТАП 6. SERVICE LAYER (МОЗГ)

```
public Project getProject(Long id) {
```

Что происходит:

1. Загружается проект
2. Проверяется бизнес-доступ:
    - владелец?
    - участник?
    - админ?
3. Если нельзя → 403
4. Если можно → вернуть DTO

! Это главный уровень защиты

ЭТАП 7. REPOSITORY

```
projectRepository.findById(id)
```

- Hibernate формирует SQL
- Отправляет в MySQL

ЭТАП 8. DATABASE

```
SELECT * FROM projects WHERE id = 1;
```

- Возвращает строки
- Hibernate → Entity

ЭТАП 9. RESPONSE PIPELINE

```
Entity → DTO → JSON
```

Spring:

- сериализует
- пишет HTTP 200
- отправляет в React

ЭТАП 10. FRONTEND

React:

- получает JSON
- кладёт в state
- рендерит UI

3. СХЕМА АУТЕНТИФИКАЦИИ (LOGIN)

```
POST /auth/login
```

Backend:

1. AuthenticationManager
2. PasswordEncoder
3. generateAccessToken()
4. generateRefreshToken()
5. refresh → DB
6. access → response
7. refresh → HttpOnly cookie

4. REFRESH FLOW

```
Browser
  ↓ (cookie)
POST /auth/refresh
  ↓
validate refresh token (DB)
  ↓
rotate refresh token
  ↓
new access token
```

! Access token НИКОГДА не хранится в БД

5. /auth/me FLOW (ОБЯЗАТЕЛЬНО)

```
GET /auth/me
```

Что делает:

- Берёт current user из SecurityContext
- Возвращает:

```
{"login": "admin","role": "ADMIN","permissions": [...]}
```

Frontend:

- не доверяет JWT
- доверяет серверу

Глобальная схема приложения