Politechnika Świętokrzyska
Wydział Elektrotechniki, Automatyki i Informatyki

Bartłomiej Kotarski
Marcin Kroczak

CAR CLEAN

Projekt zespołowy
na studiach stacjonarnych
o kierunku Informatyka

Kielce, 2020

SPIS TREŚCI

1. OMÓWIENIE WYKORZYSTANYCH WARSTW

Aplikacja została podzielona na trzy warstwy, a konkretnie na warstwę:

- prezentacji,
- biznesową,
- danych.

Za warstwę prezentacji odpowiada aplikacja napisana za pomocą biblioteki React, która pozwala na reaktywne tworzenie stron internetowych.

Warstwie biznesowej odpowiada logika całej aplikacji, która została stworzona przy pomocy Spring Boot.

Baza danych, z która łączy się aplikacja backendowa zajmuje się warstwą danych. PostgreSQL został wybrany pod to zadanie.

1.1. Dodanie serwisu przez użytkownika

1) Użytkownik wchodzi w podstronę „Oferta” co wiąże się z automatycznym zapytaniem GET do serwera.

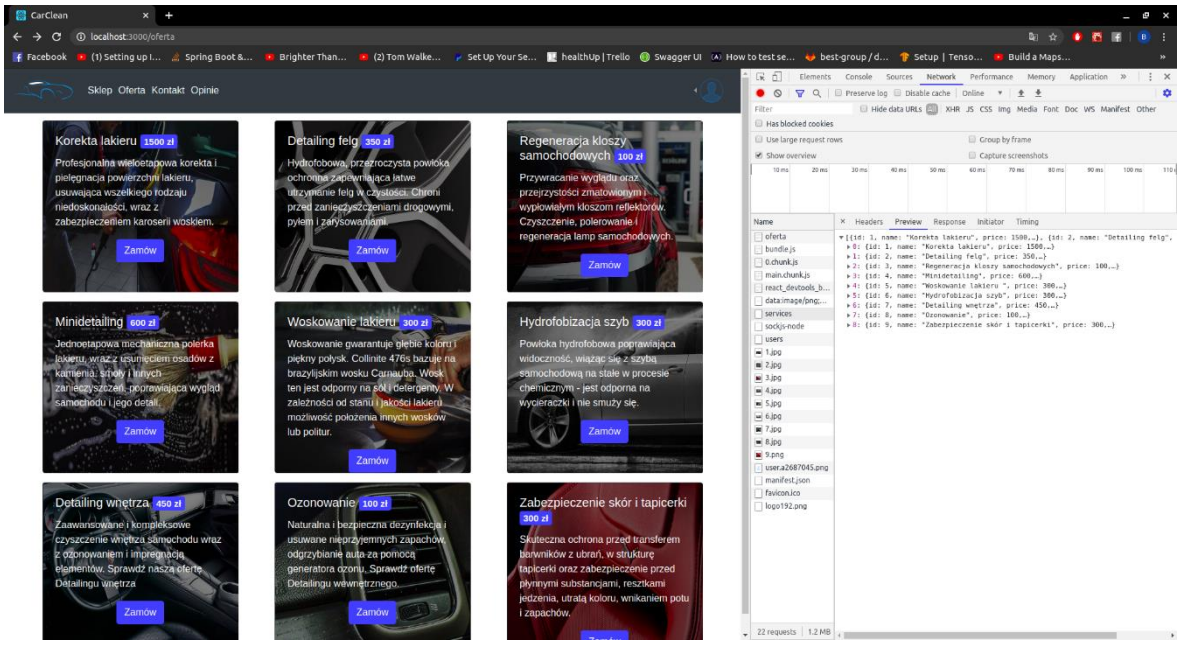
```
componentDidMount = async () => {
  try {
    const result = await getAllServices();
    this.setState({ offers: result });
  } catch (e) {
    const error = { message: 'Brak dostępnych ofert' };
    this.setState({ error: error });
  }
};
```

2) Serwer dostaje żądanie, wysyła zapytanie do bazy danych, aby pobrać wszystkie oferty dostępne w bazie.

```
@GetMapping //controllers/ServicesController.java
  public List<ServicesDto> getAllServices() {
      return servicesService.getAllServices();
  }
```

```
@Override//services/ ServicesServiceImpl.java
    public List<ServicesDto> getAllServices() {
        return servicesRepository.findAll().stream().map(
service-> ServicesDto.build(service)).collect(Collectors.toList());
    }
```

3) Baza danych zwraca na serwer wszystkie oferty wraz z własnościami jak nazwa, cena, id i opis.
4) Serwer zwraca otrzymane dane jako listę do warstwy prezentacji.
5) Warstwa prezentacji wyświetla otrzymane dane według zaprogramowanego szablonu



Rysunek 1.1.1 Odpowiedź serwera z dostępnymi ofertami

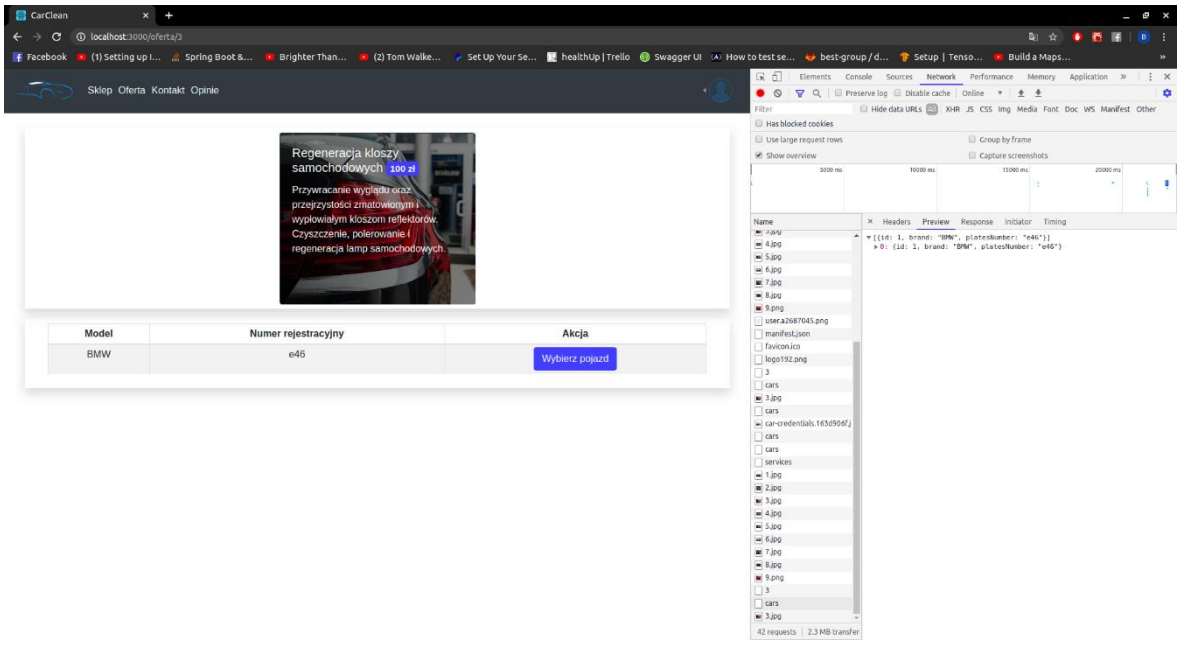
1) Użytkownik klika przycisk zamów na wybranej ofercie, wysyłając automatycznie dwa żądania na serwer. Pierwsze z nich dotyczy wybranej oferty, drugie wszystkich pojazdów użytkownika.

```
componentDidMount = async () => {//pages/ ChosenOffer.js

   try {
     const result = await getServiceById(this.props.match.params.id);
     const cars = await getAllUserCars();
     this.setState({ cars: cars.data, showModal: true, offer: result });
   } catch (e) {
     if (e.response) this.setState({ error: e.response.data });
     else this.setState({ error: 'Brak oferty' });
   }
 };
```

2) Serwer przetwarza żądania i komunikuje się z bazą danych.

```
@Override//services/ CarServicesImpl.java
   public List<CarDto> getAllUserCars(String username) {
       return carRepository.findByUserUsername(username)
               .stream()
               .map(car ->
                       CarDto.build(car.orElseThrow(() ->
                               new RuntimeException("Pojazd nie został znaleziony"))))
               .collect(Collectors.toList());
   }
```

3) Baza danych zwraca wyniki według zapytania.
4) Serwer sprawdza otrzymane dane, jeżeli użytkownik nie ma pojazdów to zwraca błąd, jeżeli ma to wysyła listę z pojazdami użytkownika.
5) Warstwa prezentacji wyświetla listę jako tabelę, a jeżeli dostanie informacje o braku pojazdów to ukazuje stosowany komunikat.

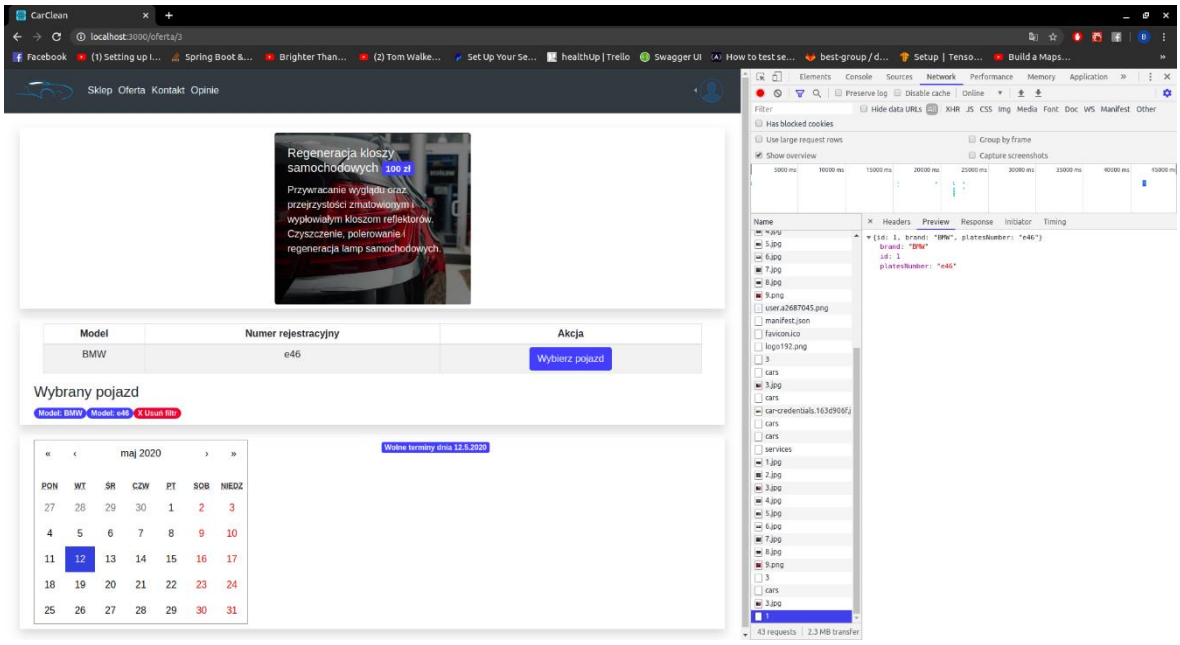


Rysunek 1.1.2 Odpowiedź zawierająca listę pojazdów użytkownika.

1) Użytkownik klika wybierz pojazd po stronie warstwy prezentacji wysyłając żądanie na serwer sprawdzające czy pojazd na pewno istnieje i należy do niego.
2) Serwer dostaje nazwę użytkownika oraz dane pojazdu. Wysyła zapytanie do bazy danych wyciągające pojazd o tych danych dla tego rodzaju użytkownika.

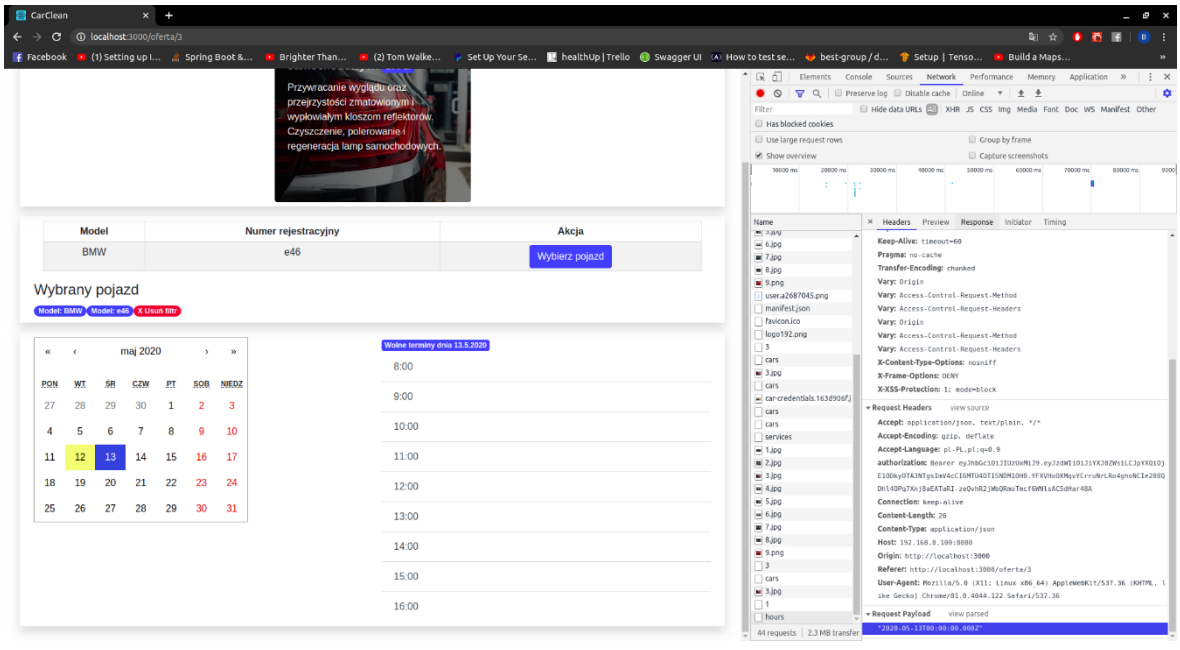
```
@Override//services/ CarServicesImpl.java
    public CarDto getUserCar(String username, int carId) {
        Car car = carRepository.findByUserUsernameAndId(username, carId).orElseThrow(() -
> new RuntimeException("Pojazd nie został znaleziony"));
        return CarDto.build(car);
    }
```

3) Baza danych zwraca stosowane dane.
4) Serwer porównuje dane i wysyła pojazd wraz z potwierdzeniem na warstwę prezentacji.
5) Warstwa prezentacji dostaje status 200 oraz pojazd, więc wyświetla kalendarz z wyborem daty.



Rysunek 1.1.3 Odpowiedź zawierająca wybrany pojazd użytkownika.

1) Użytkownik po stronie prezentacji klika w wybrany dzień na kalendarzu, co wiąże się z wysłaniem tej daty na serwer.

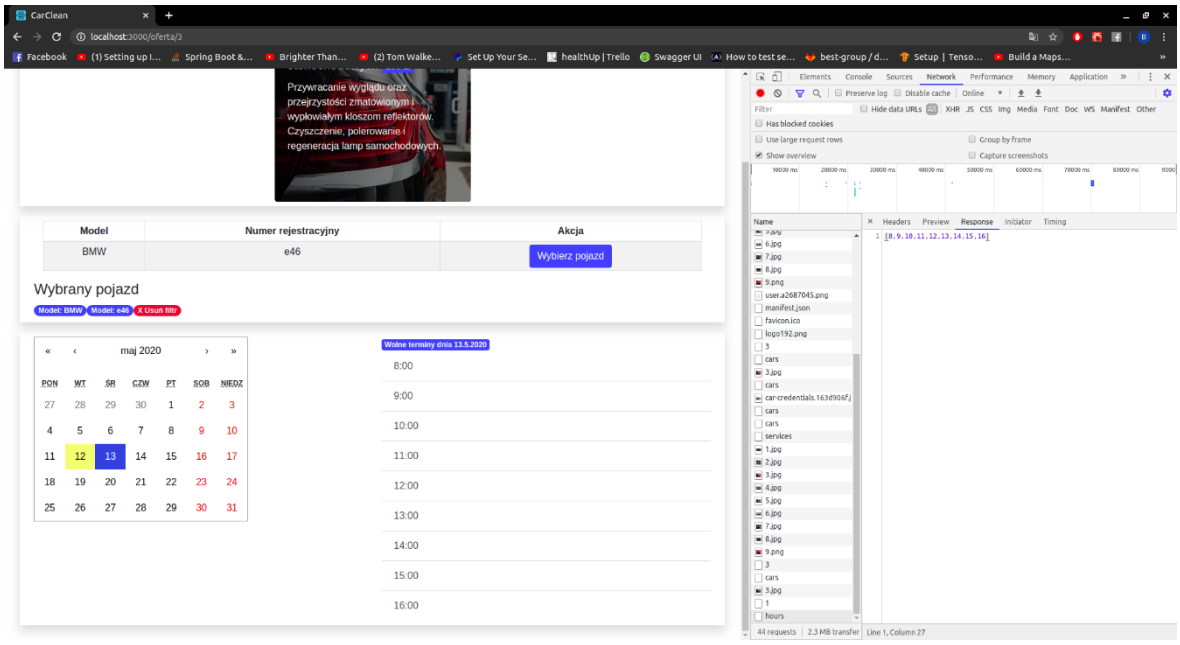


Rysunek 1.1.4 Zapytanie zawierające wybraną datę.

2) Serwer dostaje żądanie i formułuje zapytanie do bazy danych, aby zwróciła wszystkie serwisy tego dnia.
3) Baza danych zwraca serwisy według polecenia.
4) Serwer iteruje po liście i sprawdza występujące godziny pracy warsztatu. Oznacza to, że sprawdza czy jest wystąpienie godziny 8, jeżeli nie to dodaje 8 do

listy wolnych godzin i sprawdza dla 9, sytuacja powtarza się aż do ostatniej określonej godziny pracy. Godziny pracy są określane we właściwościach aplikacji, dzięki czemu są łatwe to zmiany. Serwer wysyła listę wolnych godzin na warstwę prezentacji.

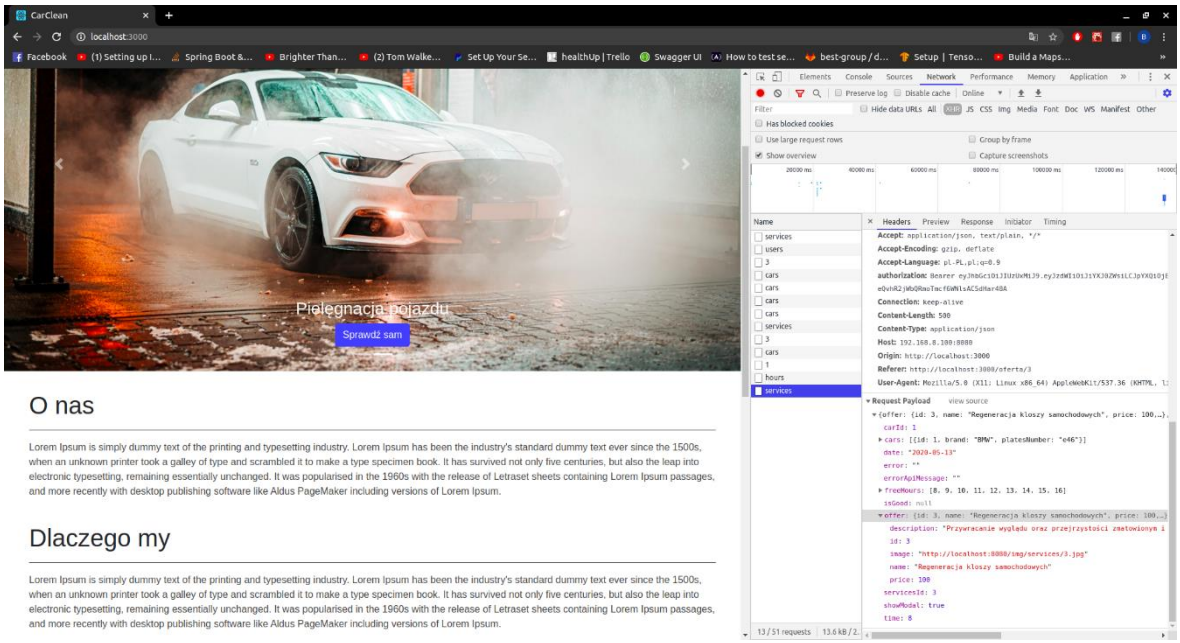
```
@Override//services/ OrderServiceServiceImpl.java
    public ResponseEntity<List<Integer>> getFreeHoursByDay(LocalDate localDate) {
        List<Integer> freeHours = new ArrayList<>();
        try {
            List<OrderService> orderServiceList = orderServiceRepository.findByDate(localDate).orEl
seThrow(() -> new Exception("Wybrany dzień nie ma zaplanowanych wizyt"));
            for (int i = startWorkHour; i <= endWorkHour; i++) {
                int temp = i;
                boolean decision = orderServiceList.stream().anyMatch(el -> el.getTime() == temp);
                if (!decision)
                    freeHours.add(i);
            }
        } catch (Exception e) {
            logger.info("{}", e.getMessage());
            for (int i = startWorkHour; i <= endWorkHour; i++)
                freeHours.add(i);
        }

        if (freeHours.isEmpty()) {
            return ResponseEntity.notFound().build();
        }
        return ResponseEntity.ok(freeHours);
    }
```



Rysunek 1.1.5 Odpowiedź w formie listy ukazująca dostępne godziny wybranego dnia.

5) Warstwa prezentacji dostaje wynik i wyświetla w postaci tabelki.

1) Użytkownik wybiera interesująca go datę i zatwierdza. Wysyła automatycznie żądanie z zapisem na usługę, konkretnego dnia i godziny wraz z wybranym pojazdem.

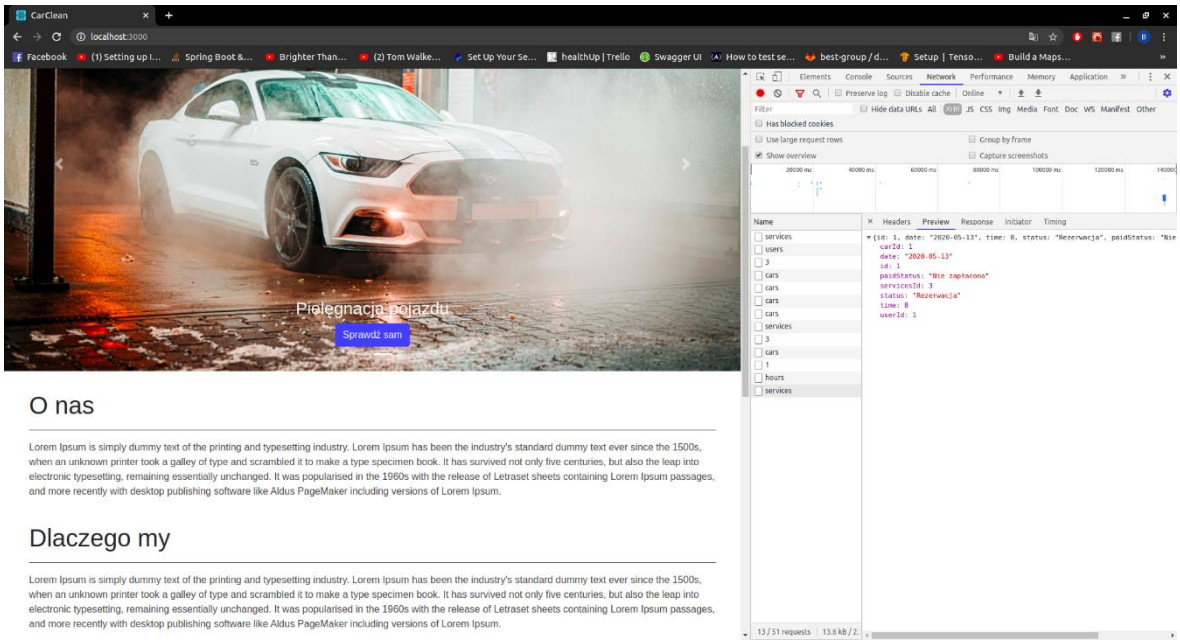


Rysunek 1.1.6 Wysłanie żądania o rezerwacji.

2) Serwer dostaje dane przetwarza je na swoje modele i wysyła żądanie do bazy danych, aby zapisać dane.
3) Baza danych zapisuje dane lub zawraca błąd.
4) Serwer dostaje informację o zapisie. Jeżeli zapis udał się pozytywnie zwraca status operacji wraz z danymi dotyczącymi zapisu.

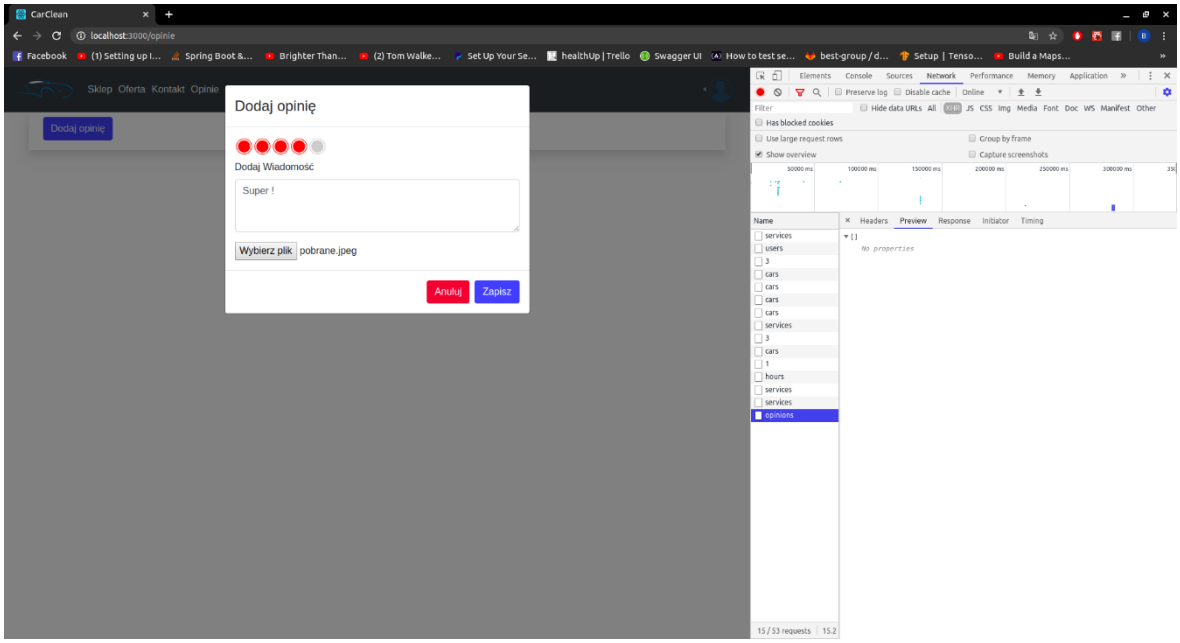
```
@Override //services/ OrderServiceServiceImpl.java
    @Transactional
    public ResponseEntity<? extends Object> addReservationService(String username, CreateOrderServi
ceDto createOrderServiceDto) {
        try {
            if (orderServiceRepository.existsByDateEqualsAndTimeEquals(LocalDate.parse(createOrderS
erviceDto.getDate()), createOrderServiceDto.getTime()))
                throw new RuntimeException("Wybrany termin jest niedostępny");
        } catch (DateTimeException e) {
            throw new RuntimeException("Błędny format daty, poprawny to YYYY-MM-DD");
        }

        Car car = carRepository.findByUserUsernameAndId(username, createOrderServiceDto.getCarId())
.orElseThrow(() -> new RuntimeException("Błędny pojazd"));
        User user = userRepository.findByUsername(username).orElseThrow(() -
> new UsernameNotFoundException("Brak osoby rozpoczynającej"));

        Services services = servicesRepository.findById(createOrderServiceDto.getServicesId()).orEl
seThrow(() -> new RuntimeException("Błędny wybrany serwis"));

        OrderService orderService = OrderService.builder()
                .date(LocalDate.parse(createOrderServiceDto.getDate()))
                .time(createOrderServiceDto.getTime())
                .status(OrderServiceStatus.WAITING)
                .paidStatus(PaidStatus.NOT_PAID)
                .car(car)
                .user(user)
                .serviceid(services)
                .build();

        orderServiceRepository.save(orderService);
        return ResponseEntity.ok(CreateOrderServiceDto.build(orderService));
    }
```

5) Warstwa prezentacji, jeżeli dostanie wiadomość i status 200 to przekierowuje użytkownika na stronę główną aplikacji.



Rysunek 1.1.7 Odpowiedź serwera na rezerwacje wykonaną przez użytkownika.

1.2. Dodanie opinii przez użytkownika



Rysunek 1.2.1 Zwrócona pusta lista opinii i okno modalne do dodania nowej.

1) Użytkownik po stronie prezentacji dodaje opinię. Wybiera ocenę, wiadomość tekstową i fotografię. Wybrane dane wysyła na serwer.

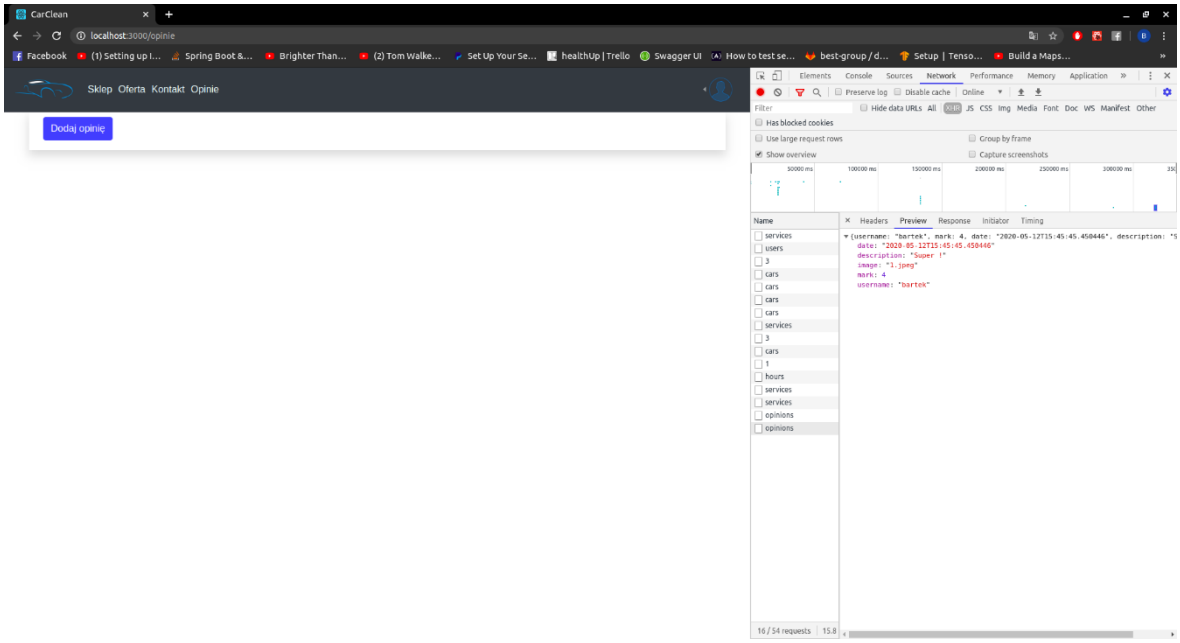
```
const safeOpinion = async () => {//pages/Opinions/index.js
   let formData = new FormData();
   formData.append('file', file);
   formData.append(
     'createOpinionDto',
     new Blob(
       [
         JSON.stringify({
           ...opinion,
         }),
       ],
       { type: 'application/json' }
     )
   );

   try {
     await addOpinionByUser(formData);
     hide();
   } catch (e) {
     console.log(e);
   }
 };
```

2) Serwer otrzymuje dane w JSON oraz dane binarne. Sprawdza rozszerzenie pliku, jeżeli jest dobre to konwertuje dane binarne na plik fizyczny i zapisuje go w udostępnianym katalogu media/opinions. Następnie wysyła dane ze ścieżka do pliku, oceną i wiadomością tekstową do bazy danych w celu zapisu.

```
//services/OpinionServiceImpl.java
private String saveFile(Long userId, MultipartFile file) throws IOException {

        String extension = FilenameUtils.getExtension(file.getOriginalFilename());
        if (!(extension.equals("png") || extension.equals("jpg") || extension.equals("jpeg")))
            throw new RuntimeException("Błędne rozszerzenie pliku");

        String imageName = new StringBuilder().append(userId).append(".").append(extension).toStrin
g();

        byte[] bytes = file.getBytes();
        Path path = Paths.get(folder + imageName);
        Files.write(path, bytes);
        return imageName;
    }


    @Override
    public ResponseEntity<CreateOpinionDto> createOpinion(String username, MultipartFile file, Crea
teOpinionDto createOpinionDto) throws IOException {

        User user = userRepository.findByUsername(username).orElseThrow(() -
> new UsernameNotFoundException("Użytkownik " + username + " nie został odnaleziony"));

        if (user.getOpinion() != null)
            this.deleteOpinion(user.getUsername());


        String image = saveFile(user.getId(), file);
        Opinion opinion = Opinion.builder()
                .user(user)
                .date(LocalDateTime.now())
                .description(createOpinionDto.getDescription())
                .mark(createOpinionDto.getMark())
                .image(image)
                .build();

        opinionRepository.save(opinion);
        return ResponseEntity.ok(CreateOpinionDto.build(opinion));
    }
```

3) Baza danych zapisuje dane.

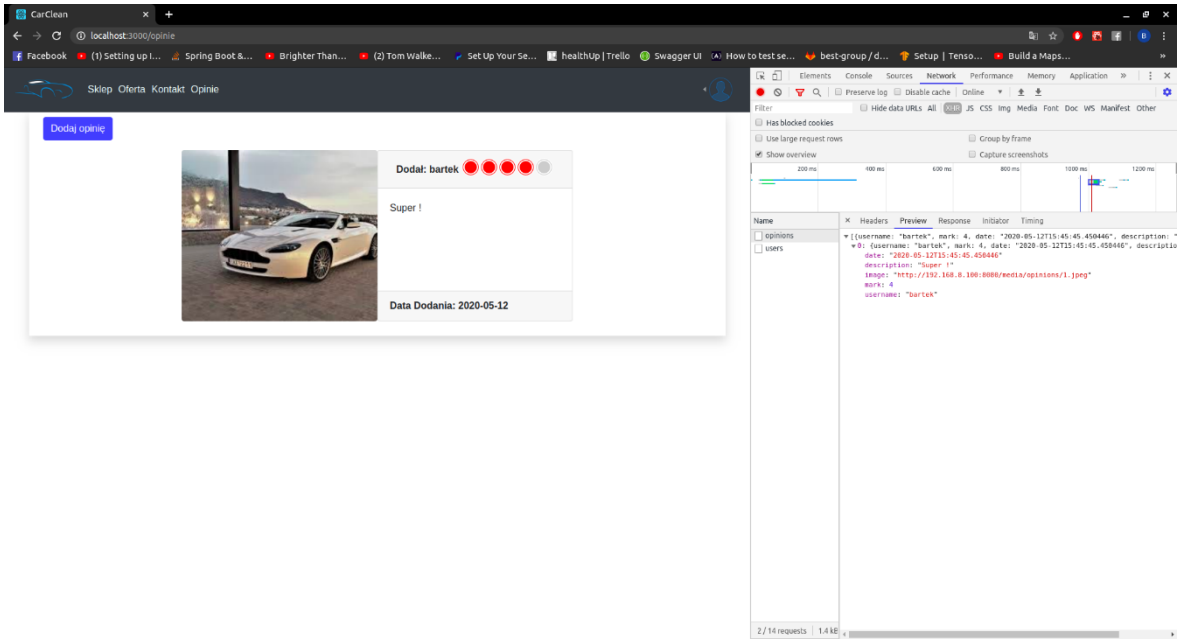
4) Jeżeli serwer nie dostanie błędu oznacza, że dane zostały zapisane, więc zwraca status operacji na warstwę prezentacji.
5) Warstwa prezentacji dostaje status pozytywny, zamyka dzięki temu okno modalne dotyczące dodawania opinii.



Rysunek 1.2.2 Odpowiedź zamykająca okno modalne, informująca o pozytywnym dodaniu opinii.

W celu wyświetlenia dodanej opinii, użytkownik musi wysłać zapytanie GET poprzez odświeżenie strony. Serwer w tym wypadku konwertuje ścieżkę zdjęć dla aktualnej domeny.

```
@Override//services/OpinionServiceImpl.java
    public ResponseEntity<List<CreateOpinionDto>> getAllOpinions() {
        List<Opinion> opinions = opinionRepository.findAll();
        List<CreateOpinionDto> createOpinionDto = opinions.stream().map(opinion -
> CreateOpinionDto.build(opinion)).collect(Collectors.toList());
        createOpinionDto.forEach(opinion -> opinion.setImage(
                new StringBuilder()
                        .append(this.domainName)
                        .append("/")
                        .append(this.folder)
                        .append(opinion.getImage())
                        .toString()
        ));
        return ResponseEntity.ok(createOpinionDto);
    }
```

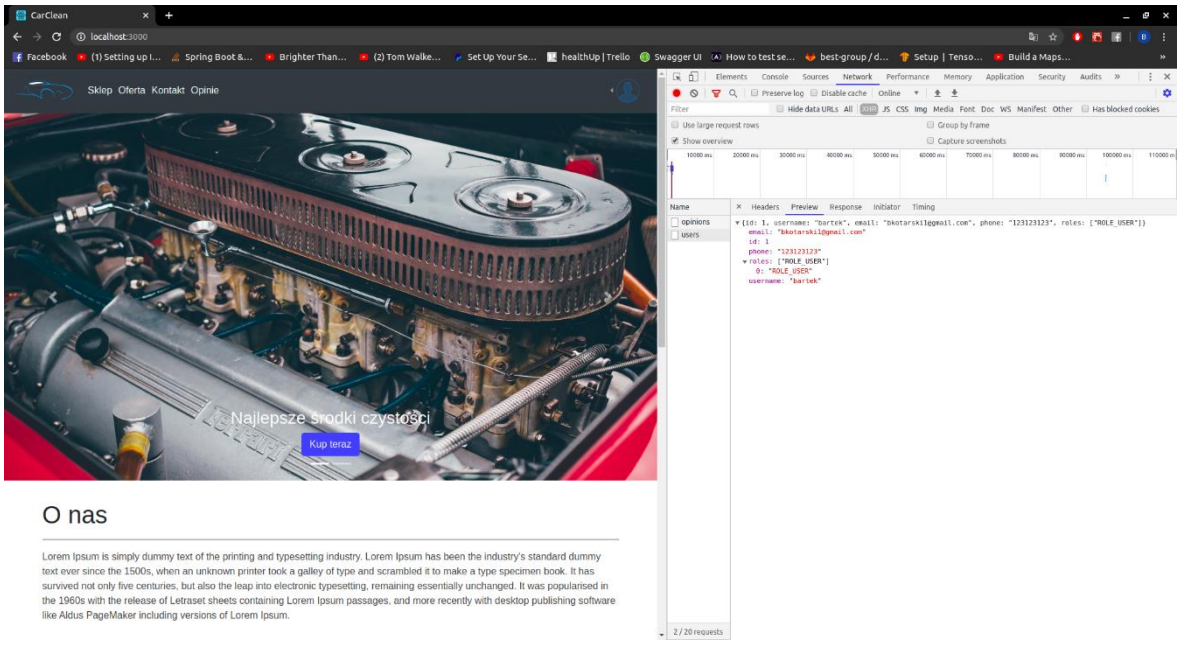


Rysunek 1.2.3 Odpowiedź zawierają listę wszystkich opinii.

1.3. Logowanie zapisanego użytkownika w przeglądarce

Jeżeli użytkownik jest już zalogowany to jego dane przechowywane są w 2 miejscach. LocalStoarge zawiera zakodowany token, a reduxowy wzorzec projektowy pozwala na przechowywanie stanu, który posiada dane o roli, tokenie i czy użytkownik został zautoryzowany. W momencie odświeżania karty lub zamknięcia przeglądarki, stan reduxowy jest czyszczony i zostaje jedynie token w przeglądarce, wraz z zakodowaną datą ważności. Poniższy przypadek ukazuje „nieświadome" logowanie użytkownika, który odświeżył przeglądarkę lub ją zamknął.

```
export const authReducer = (state, action) => {//context.reducer.js
  switch (action.type) {
    case LOGIN:
      localStorage.setItem('@token', action.user.token);
      const temp = {
        isAuthenticated: true,
        token: action.user.token,
        roles: action.user.roles,
      };
      return {
        ...state,
        ...temp,
      };
    case LOGOUT:
      localStorage.clear();
      const temp2 = {
        isAuthenticated: false,
        token: '',
        roles: [],
      };
      return {
        ...state,
        ...temp2,
      };
    default:
      return state;
  }
};
```



Rysunek 1.3.1 Odpowiedź uzyskana poprzez wysłanie tokenu na serwer. Odpowiedź zawiera dane o użytkowniku.

1) Warstwa prezentacji wysyła żądanie na serwer wraz z tokenem zapisanym w localstorage.

```
useEffect(() => {//context/index.js
  const searchUser = async () => {
    const token = localStorage.getItem('@token');
    if (token) {
      try {
        const user = (await getUserByToken()).data;
        user.token = token;
        dispatch({ type: LOGIN, user: { ...user } });
      } catch (e) {
        console.log(e);
      }
    }
  };
  searchUser();
}, []);
```

2) Serwer dostaje token. Dekoduje go, sprawdzając jego ważność oraz przynależność. Przynależność jest sprawdzana za pomocą sekretnego klucza, który zakodował podpis tokenu jwt. Jeżeli ważność i przynależność są prawidłowe to system wyciąga z tokenu nazwę użytkownika, którą

wysyła do bazy danych w celu pobrania większej ilości danych (rola, id, email, telefon).

```
public boolean validateToken(String token) { //JwtProvider.java
        try {
            Jwts.parser().setSigningKey(secret).parseClaimsJws(token);
            return true;
        } catch (SignatureException e) {
            logger.error("Invalid signature: {}", e.getMessage());
        } catch (MalformedJwtException e) {
            logger.error("Invalid token: {}", e.getMessage());
        } catch (ExpiredJwtException e) {
            logger.error("Expired token: {}", e.getMessage());
        } catch (UnsupportedJwtException e) {
            logger.error("Unsupported token: {}", e.getMessage());
        } catch (IllegalArgumentException e) {
            logger.error("Claims string is empty: {}", e.getMessage());
        }
        return false;
    }
```

```
@Override //AuthorizationServiceImpl.java
    public ResponseEntity loginUser(SignInDto signInDto) {
        Authentication authentication = manager.authenticate(
                new UsernamePasswordAuthenticationToken(
                        signInDto.getUsername(),
                        signInDto.getPassword()
                )
        );

        SecurityContextHolder.getContext().setAuthentication(authentication);
        String token = provider.generateToken(authentication);

        return ResponseEntity.ok(new JwtTokenDto(token));
    }
```

3) Baza danych zwraca dane według nazwy użytkownika.
4) Serwer formatuje te dane według określonego schematu dto, żeby zwrócić na warstwę prezentacji w bezpieczny sposób, odpowiednie informacje.

```
 @Override //AuthorizationServiceImpl.java
    public UserDto getUser(Authentication authentication) {
        return UserDto.build((UserPrincipal) authentication.getPrincipal());
    }
```

5) Warstwa prezentacji dostaje dane uwierzytelniające i zapisuje je w stanie kontekstu. Umożliwia to dostęp do części aplikacji, które są niewidoczne dla użytkowników np. o innej roli.

1.4. Zmiana statusu usługi przez pracownika

W celu zmiany statusu najpierw musi zostać podjęta akcja wyboru serwisu do edycji.

1) Pracownik wysyła zapytanie (jako wybraną datę) na serwer.

```
getAllEmployeeServicesByDay(new Date(formatDate(chosenDate)))

export const getAllEmployeeServicesByDay = (body) =>
postSafe(servicesEmployeeApiUrl(), body);
```

2) Serwer przetwarza datę i tworzy odpowiednie zapytanie do bazy danych.

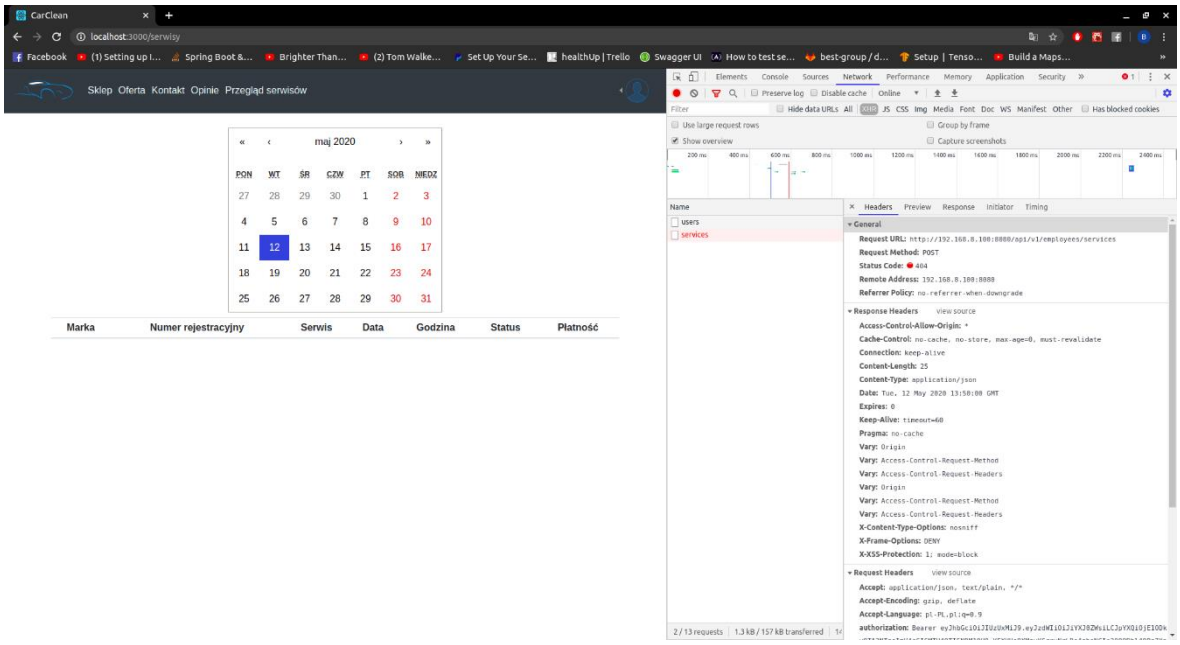
```
List<OrderService>orders=
orderServiceRepository.findAllByDate(localDate).orElseThrow(()->new
RuntimeException("Brak rezerwacji tego dnia"));
```

3) Baza danych zawraca listę serwisów tego dnia.
4) Serwer sprawdza tą listę, jeżeli jest pusta to zwraca błąd 404. W przeciwnym wypadku zwraca listę serwisów tego dnia.

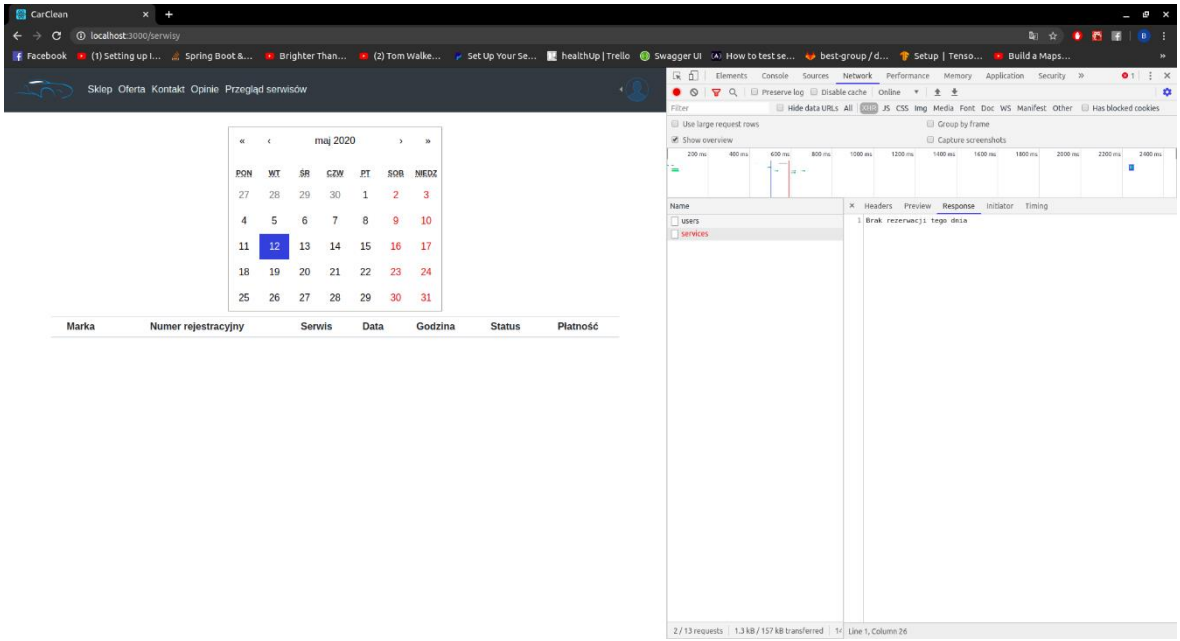
```
List<OrderService>orders=
orderServiceRepository.findAllByDate(localDate).orElseThrow(() -> new
RuntimeException("Brak rezerwacji tego dnia"));
return ResponseEntity.ok(orderServiceDtoList);
…
publicResponseEntity<List<GetOrderServiceDto>> getAllServiceByDay(@RequestBody
LocalDate localDate) {
return orderServiceService.getAllServiceByDay(localDate); }
```

5) Warstwa prezentacji dostaje dane i wyświetla je w formacie tabelki.

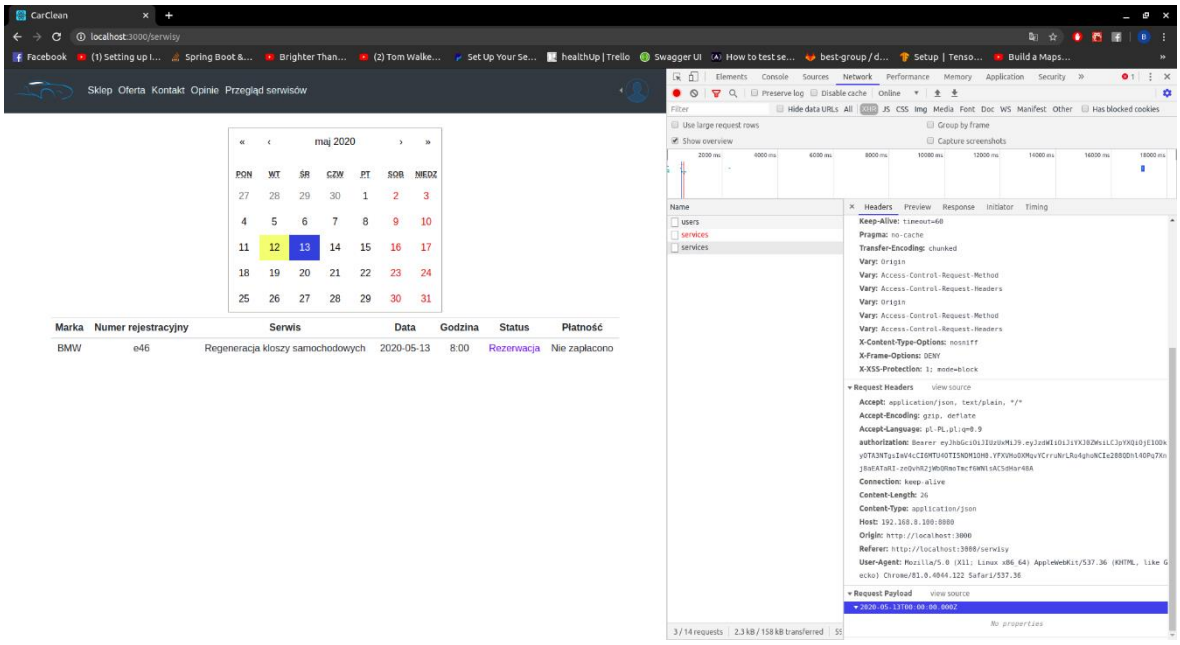
```
useEffect(() => {
 getAllEmployeeServicesByDay(new Date(formatDate(chosenDate)))
   .then((res) => {
     if (res.status === 200) setServiceData(res.data);
   })
   .catch((err) => console.log(err));
}, [chosenDate, isModalVisible]);
```



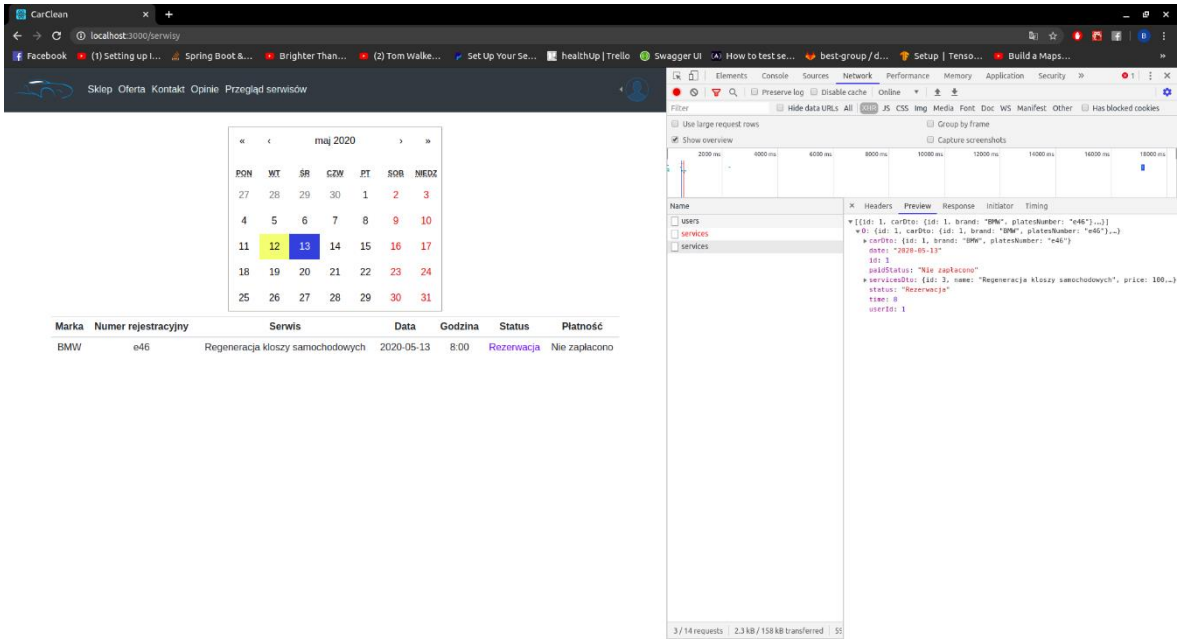
Rysunek 1.4.1 Odpowiedź z błędem 404 na żądanie z datą serwisów.



Rysunek 1.4.2 Treść błędu 404 w przypadku pobrania serwisów danego dnia.



Rysunek 1.4.3 Przykład żądania z wybraną przez użytkownika datą.

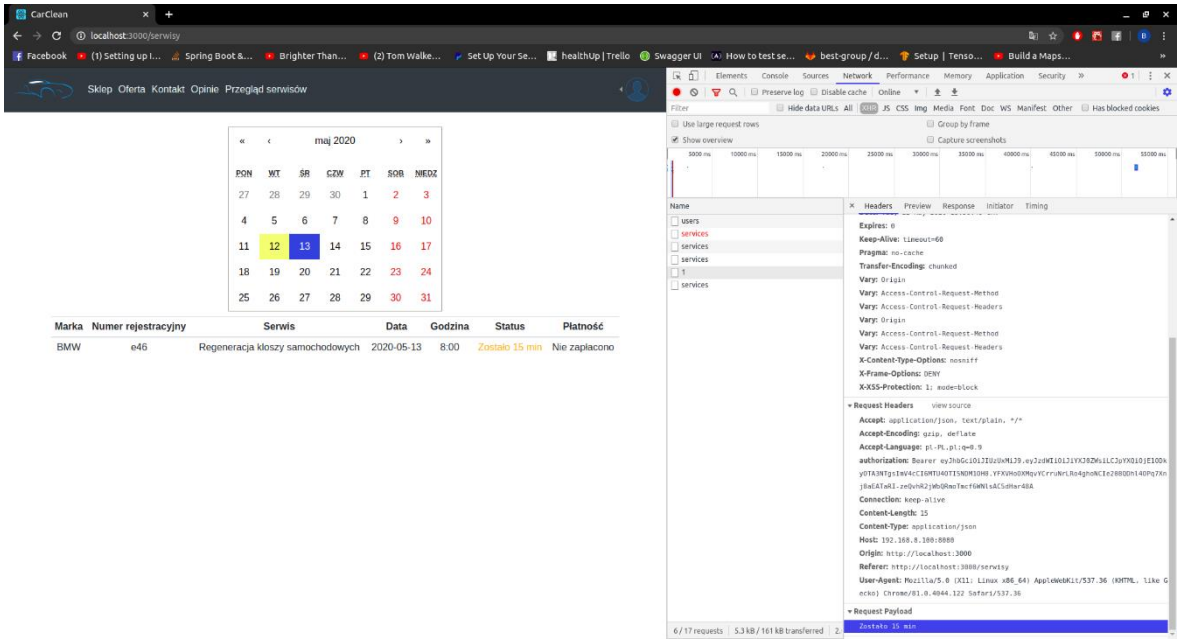


Rysunek 1.4.4 Odpowiedź serwera wraz z wizualizacją w warstwie prezentacji.

W momencie, kiedy lista serwisów jest dostępna pracownik klika w wybrane zadanie następnie otwiera się specjalne okno modalne, aby zmienić status.

1) Pracownik po stronie warstwy prezentacji klika przycisk zmiany statusu o nazwie „Zostało 15 min”.

```
    <Button
              onClick={setStatusByMethod}
              name='Zostało 15 min'
              variant='outline-warning'
            >
  const saveStatusChanges = async () => {
    if (status) {
      try {
        await putServiceStatusById(serviceId, status);
…
```



Rysunek 1.4.5 Wysłanie żądania wraz z wybranym nowym statusem.

2) Serwer dostaje wiadomość o nowym statusie. Pobiera z bazy ten serwis sprawdza jego aktywny status. Jeżeli aktywny status to „Zakończono” lub „Anulowano” to nie można zmienić statusu i zwraca błąd na warstwę prezentacji. W innym przypadku po pobraniu obiektu, aktualizuje go i zapisuje w bazie danych.

```
OrderService order = orderServiceRepository.findById(idService).orElseThrow(() -> new
RuntimeException("Brak serwisu"));
        if (order.getStatus() == OrderServiceStatus.DONE || order.getStatus() ==
OrderServiceStatus.CANCEL)
            return ResponseEntity.badRequest().body(GetOrderServiceDto.build(order));
```

3) Baza danych aktualizuje wybraną pozycje.

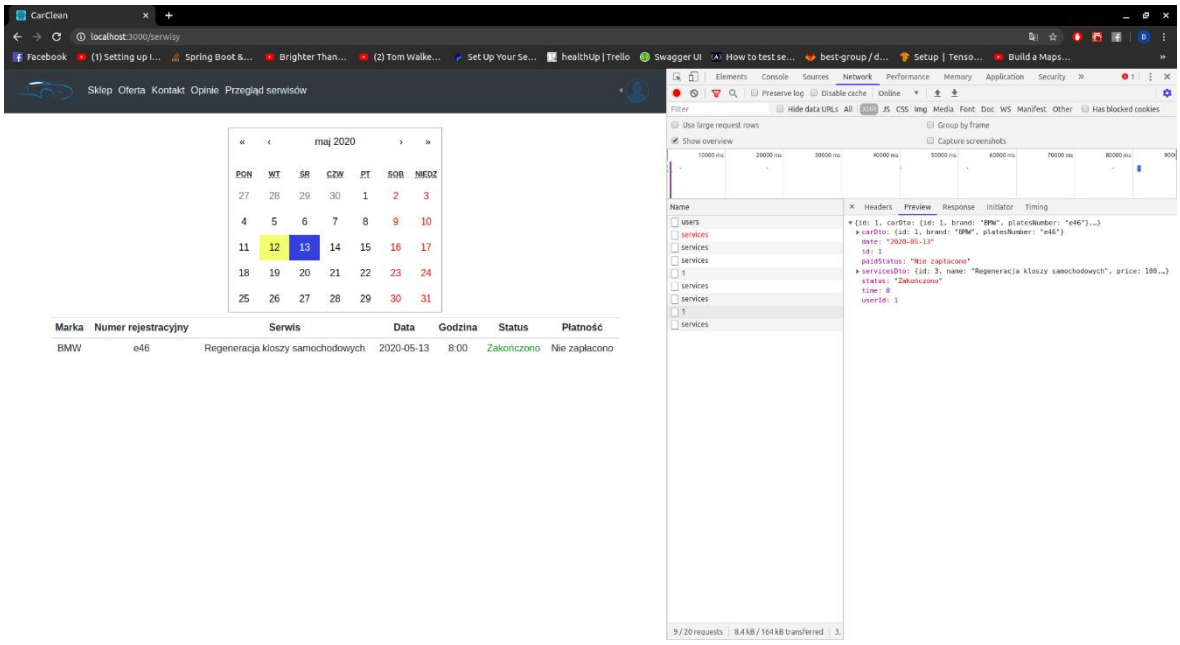
```
orderServiceRepository.save(order);
```

4) Serwer sprawdza, czy wystąpił jakiś błąd, jeżeli nie to zwraca status 200 i zaktualizowane dane.

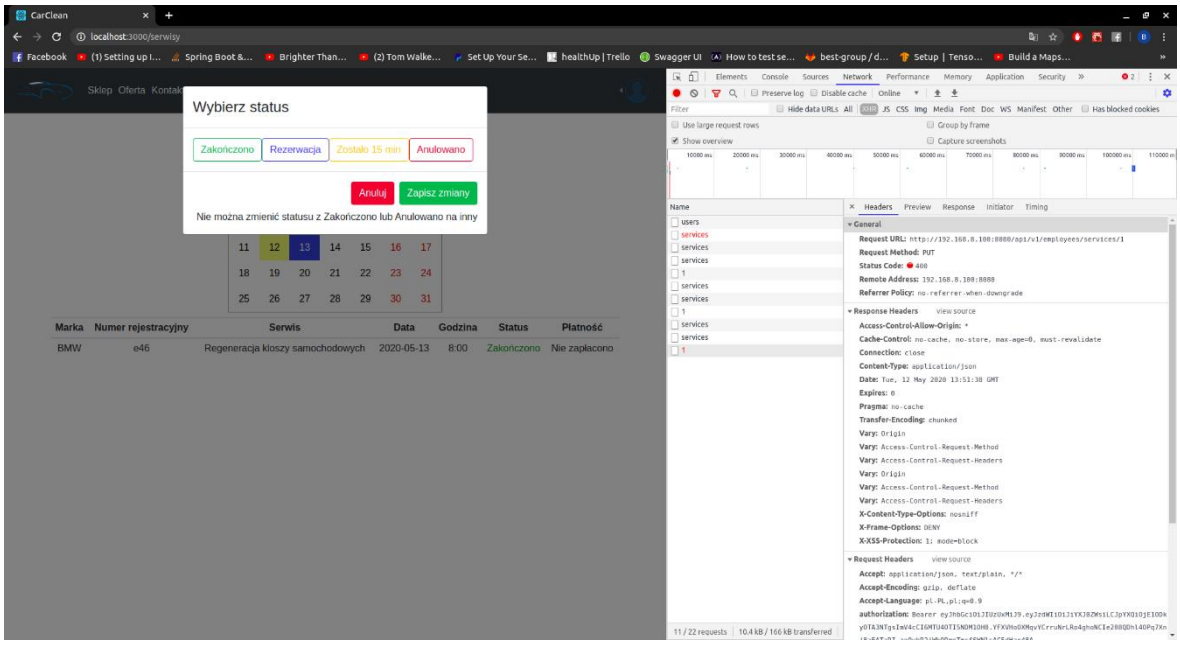
```
    @PutMapping("{idService}")
    @PreAuthorize("hasRole('EMPLOYEE')")
    public                                          ResponseEntity<GetOrderServiceDto>
changeServiceStatus(@PathVariable("idService") Long idService, @RequestBody String
status) {
        return orderServiceService.changeServiceStatus(idService, status);
    }
```

5) Warstwa prezentacji wyświetla zaktualizowane dane w formie tabelki.

```
                {serviceData.length > 0 &&
                  serviceData.map((service) => (
                    ...
```



Rysunek 1.4.6 Odpowiedź serwera na zmianę statusu.



Rysunek 1.4.7 Błąd 400 podczas niewłaściwej akcji pracownika.

1.5. Dodawanie produktów do koszyka przez zalogowanego użytkownika

1) Klient wysyła żądanie o pobranie produktów z serwera.

```
  componentDidMount = async () => {
        try {
            const result = await getAllProducts();
        …
    };
…
export const getAllProducts = () => get(shopApiUrl());
```

2) Serwer otrzymuje i obsługuje żądanie, wysyła zapytanie do bazy danych celem pobrania produktów.

```
    @GetMapping
    public List<ProductDto> getAllProducts() {
        ...
    }
…
    @Override
    public List<ProductDto> getAllProducts() {
        return            productRepository.findAll().stream().map(product            ->
ProductDto.build(product)).collect(Collectors.toList());
    }
```

3) Baza danych zwraca na serwer produkty i ich własności.

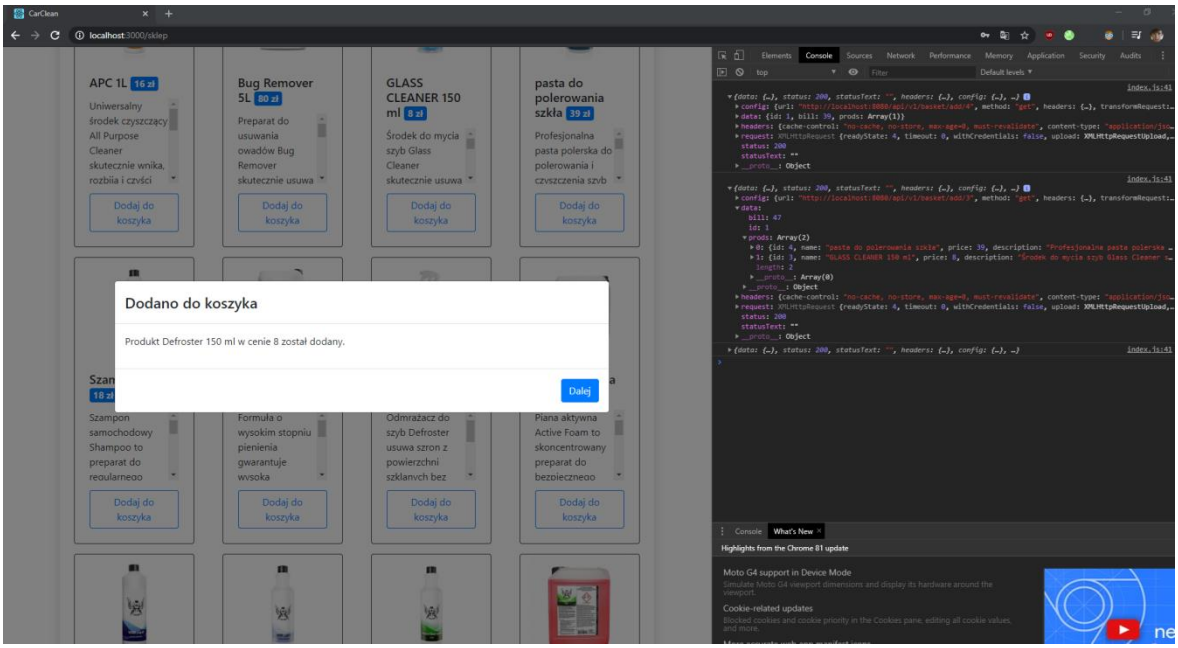
```
@Repository
public interface ProductRepository extends JpaRepository<Product, Integer> {
...
    List<Product> findAllByCategory(ProductCategory category);
...
}
```

4) Serwer zwraca do warstwy prezentacji dane (listę produktów).

```
    @GetMapping
    public List<ProductDto> getAllProducts() {
        return productService.getAllProducts();
    }
```

5) Warstwa prezentacji wyświetla dane.

```
  componentDidMount = async () => {
        …
            this.setState({ products: result });
        }
    };

        <Container className='shadow my-3'>
          <Row xs={1} sm={2} md={3} lg={4}>
            {products.map((product) => (
              <ProductCard
…
              />
            ))}
          </Row>
        </Container>
```



Odpowiedź serwera na dodanie produktów do koszyka

1) Użytkownik wybiera z listy żądany produkt. Klika przycisk "Dodaj do koszyka" co powoduje wysłanie żądania GET na serwer.

```
<Button onClick={addToBasket} variant='outline-primary'>
Dodaj do koszyka
</Button>

const addToBasket = async () => {
...
      await addToUserBasket(id)
...
};
```

2) W serwerze dla każdego nowo utworzonego użytkownika tworzony jest koszyk. Koszyk ten jest modyfikowany po kazdym dodaniu lub usunięciu produktu. Jest on też czyszczony po utworzeniu zamówienia. W tej sytuacji do listy z produktami dodawany jest nowy produkt i aktualizowana jest suma. Następnie serwer wysyła do bazy danych informacje o stanie koszyka.

```
@GetMapping("add/{productId}")
@PreAuthorize("hasRole('USER')")
public ResponseEntity<BasketDto> addProductToBasket(Authentication authentication,
@PathVariable("productId") int productId) {
...
```

```
}


int newPrice = basket.getBill() + product.getPrice();
basket.setBill(newPrice);
basketRepository.save(basket);
```

3) W bazie danych przypisywany jest ID produktu dla ID koszyka. Baza danych zwraca wyniki.

```
public interface BasketRepository extends JpaRepository<Basket, Integer> {

    Optional<Basket> findByUserUsername(String username);
}
```

4) Serwer sprawdza wynik. Przekazuje listę z zamówionymi dotychczas produktami.

```
public ResponseEntity<BasketDto> addProductToBasket(String username, int productId) {
...
return ResponseEntity.ok(BasketDto.build(basket));
}

@GetMapping("add/{productId}")
@PreAuthorize("hasRole('USER')")
public ResponseEntity<BasketDto> addProductToBasket(Authentication authentication,
@PathVariable("productId") int productId) {
return          basketService.addProductToBasket(AuthMiner.getUsername(authentication),
productId);
}
```

5) Warstwa prezentacji informuje o powodzeniu wyświetlając okno modalne.

```
const MyVerticallyCenteredModal = ({ name, price, onHide, show }) => (
…
      <Modal.Title>Dodano do koszyka</Modal.Title>
…
        Produkt {name} w cenie {price} został dodany.
…
      <Button onClick={onHide}>Dalej</Button>
…
);
```

WYKAZ RYSUNKÓW