Politechnika Świętokrzyska w Kielcach
Podsumowanie Projektu

Kierunek: Informatyka
Rok: 3
Semestr: 5
Skład zespołu: Patryk Grzywacz i Dominik Grudzień
Nazwa przedmiotu: Podstawy Inżynierii Programowania (Projekt)
Temat projektu: Aplikacja bankowa

Cel projektu

Celem projektu było opracowanie i zaimplementowanie aplikacji na temat wybranego systemu informacyjnego. Projekt musiał być zrealizowany za pomocą relacyjnej bazy danych (PostgresSQL) oraz obiektowego języka programowania (Java). Aplikacja musiała posiadać interfejs użytkownika, przetwarzanie danych oraz składowanie danych. Dodatkowo projekt musiał być podzielony na klika rodzajów użytkowników (Administrator, Pracownik, Klient), której liczba jest większa bądź równa liczbie osób w zespole.

Użyte biblioteki i narzędzia w projekcie

Podczas pracy przy projekcie pomogły nam następujące biblioteki i narzędzia:

Backend (Język programowania Java 16)

Biblioteki:

- Spring Boot 2.5.5
    - Data JPA
    - Spring Boot Mail
    - Spring Security
    - Spring Boot Test
    - Spring Doc (Swagger 2.0)
- Maven 4.0
- JUnit 5
- JSON Web Token
- Vladmihalcea (Hibernate-Types)
- Mockito
- Project Lombok
- PostgresSQL
- Net.kaczmarzyk Specifications
- Eclipse Yasson
- Javax JSON
- Awaitility
- Jasypt Encryptor
- BCryptPasswordEncryptor

Narzędzia:

- Postman
- Heroku
- IntelliJ IDEA 2021.2, Github
- Discord
- Pg Admin

Frontend (Języki programowania: TypeScript 4.4.4, SCSS. Język znaczników: HTML)

Biblioteki:

- React 17.0.2
- Axios
- Bootstrap 5
- React-bootstrap

- React-bootstrap icons
- React-bootstrap-table-next
- React-datepicker
- React-number-format
- React-toastify
- React-router
- Sass
- Yup
- DayJs
- Formik
- Jwt-decode
- Lodash.isequal
- Moment

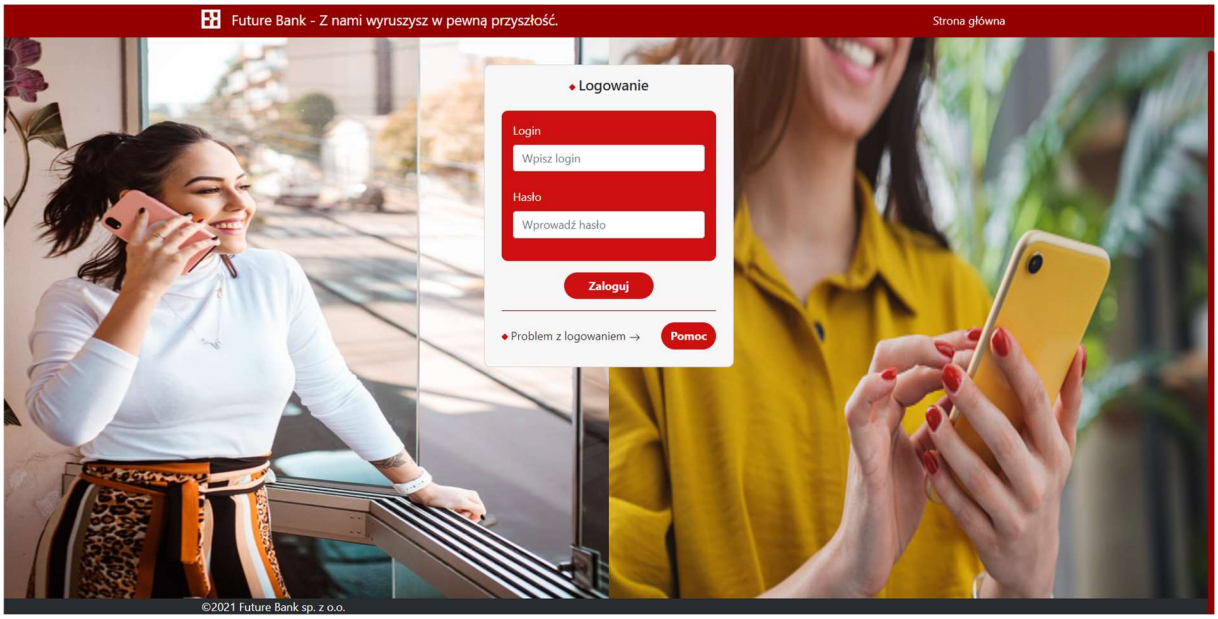
Narzędzia:

- WebStorm 2021.2.1
- Konsola Google Chrome

Sposób wykonania projektu

Frontend

- W interfejsie użytkownika został zachowany duży poziom szczegółowości. Każde detale zostały dokładnie wystylizowane za pomocą naszych selektorów SCSS oraz predefiniowanych klas z biblioteki Bootstrap 5.



Przykład strony logowania do aplikacji

- Staraliśmy się unikać powtórzeń w kodzie za pomocą tworzenia osobnych zbiorów wyrażeń, które wykonują jakieś zadanie oraz stosowaniem typów generycznych.

```
export const useFetchRawData = <T extends unknown>(endpoint: string, params?: any) => {
  const [rawData, setRawData] = useState<T>();
  const [isPending, setIsPending] = useState(false);

  const fetchData = useCallback(async (callbackParams?: any) => {
```

```
    setIsPending(true);
    try {
      const { data } = await axios.get<T>(endpoint, { params: callbackParams || params });
      setRawData(data);
    } catch (e) {
      toast.error(`Coś poszło nie tak przy pobieraniu danych: ${e}`);
    } finally {
      setIsPending(false);
    }
  }, [setIsPending, setRawData, endpoint, params]);

  useEffect(() => {
    fetchData().catch();
  }, [fetchData]);

  return { rawData, fetchData, isPending };
};
```

Przykład funkcji pobierającej dane z aplikacji backendowej

- Podczas pisania komponentów, które korzystają ze wspólnych stanów stosowaliśmy tzw. konteksty. Dzięki nim w łatwy sposób mogliśmy uzyskać dostęp do danych z różnych poziomów zagnieżdżeń bez konieczności przesyłania ich między komponentami.

```
Const Loan = () => {
  return (
    <LoanProvider>
      <Row>
        <Col xs={8}>
          <ClientCardLayout location={LocationHeaders.Loan} style={{ height: '45.2rem' }}>
            <NewLoan/>
          </ClientCardLayout>
        </Col>

        <Col xs={4} className="mt-2" style={{ height: '45.2rem' }}>
          <ActiveLoan/>
        </Col>
      </Row>
    </LoanProvider>
  );
};
```

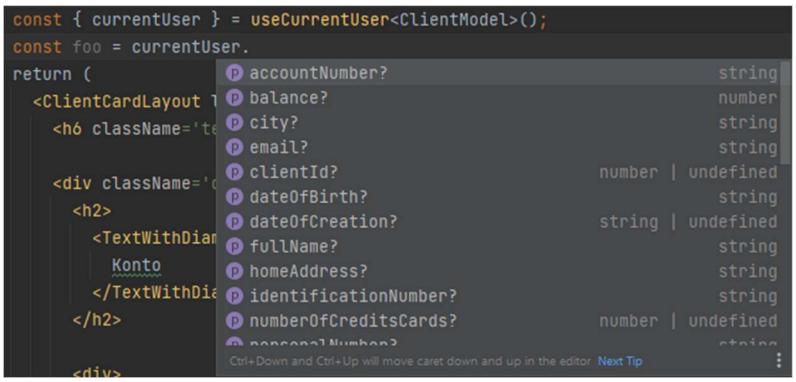
Przykład inicjacji konteksu za pomocą dostawcy LoanProvider

```
const NewLoanForm = () => {
  const { currentLoan } = useLoan();
  return (
    <Form className='position-relative mt-4 h-100' noValidate>
      <FormBlob style={{ zIndex: 1 }}/>

      <Row>
        <Col {...colProps}>
          <TextInput
            name='initialRatesNumber'
            label='Liczba rat'
            type='number'
            className='rounded-0 rounded-start float-start'
            labelClassName='fw-bold'
            placeholder='Wpisz liczbę rat'
            disabled={currentLoan.isActive}
          />
        </Col>
      </Row>
```

Przykład dostępu do danych z formularza zagnieżdżonego 3 komponenty dalej od inicjacji kontekstu

- Dzięki językowi TypeScript oraz interfejsom mogliśmy w łatwy sposób otypować dane. Usprawniło to pracę oraz czytelność kodu.



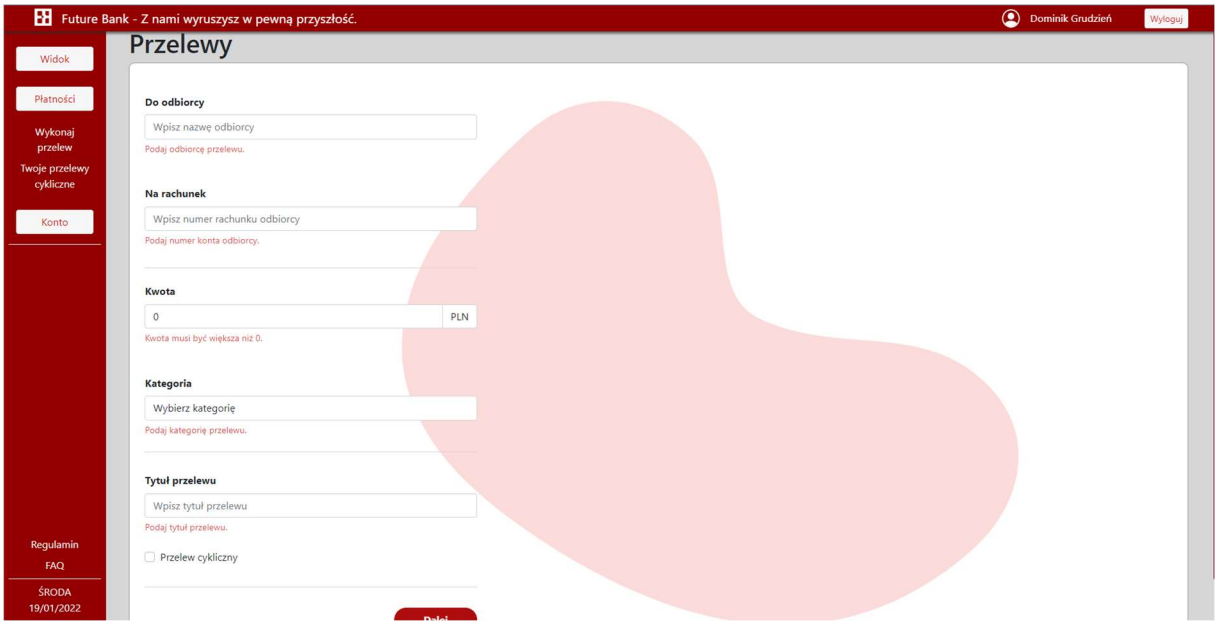
Przykład zastosowania interfejsu ClientModel

- Stosowanie zewnętrznych bibliotek do React'a ułatwiło rozwiązywanie problemów związanych z walidacją formularzy, czy budowaniem tabel za pomocą kolumn wierszy oraz danych

```
import * as Yup from 'yup';

export const TransferValidationSchema = Yup.object().shape({
  amount: Yup.number().required('Podaj kwotę.')
    .min(1, 'Kwota musi być większa niż 0.')
    .max(5000, 'Kwota musi być mniejsza od 5000'),
  transferDate: Yup.string()
    .when('isCyclicalTransfer', {
      is: true,
      then: Yup.string().required('Podaj datę przelewu cyklicznego.'),
    }),
  category: Yup.string().required('Podaj kategorię przelewu.'),
  receiver_sender: Yup.string().required('Podaj odbiorcę przelewu.'),
  title: Yup.string().required('Podaj tytuł przelewu.'),
  toAccountNumber: Yup.string().required('Podaj numer konta odbiorcy.')
    .min(26, 'Niepoprawny numer konta odbiorcy'),
});
```

Przykład schematu walidującego (formularz nowego przelewu)



Efekt zastosowania walidacji w formularzu

```
const columns = [
  {
    ...defaultColumnStyle,
    dataField: 'transferDate',
```

```
    text: 'Data',
    sort: true,
  },
  {
    ...defaultColumnStyle,
    dataField: 'category',
    text: 'Kategoria',
    classes: getCategoryCellStyle,
  },
  {
    ...defaultColumnStyle,
    dataField: 'receiver_sender',
    text: 'Odbiorca / Nadawca',
  },
  {
    ...defaultColumnStyle,
    dataField: 'displayAmount',
    text: 'Kwota',
    classes: getAmountCellStyle,
  },
];

const HistoryTable = () => {
  const { transfers } = useHistory();
  const { data, isPending } = transfers;

  const {
    toggleVisibility,
    showModal,
    entity,
  } = useModalState<TransferDisplayModel>();

  const tableProps = useTableProps<TransferDisplayModel>(
    { data, isPending },
    'transferId',
    { initialSortBy: 'transferDate' },
    (e: any, row: TransferDisplayModel) => toggleVisibility(row),
  );

  return (
    <>
      <BootstrapTable
        {...tableProps}
        columns={columns}
      />

      <TransferDetailsModal
        showModal={showModal}
        toggleVisibility={toggleVisibility}
        data={entity || {} as TransferDisplayModel}
      />
    </>
  );
};
```

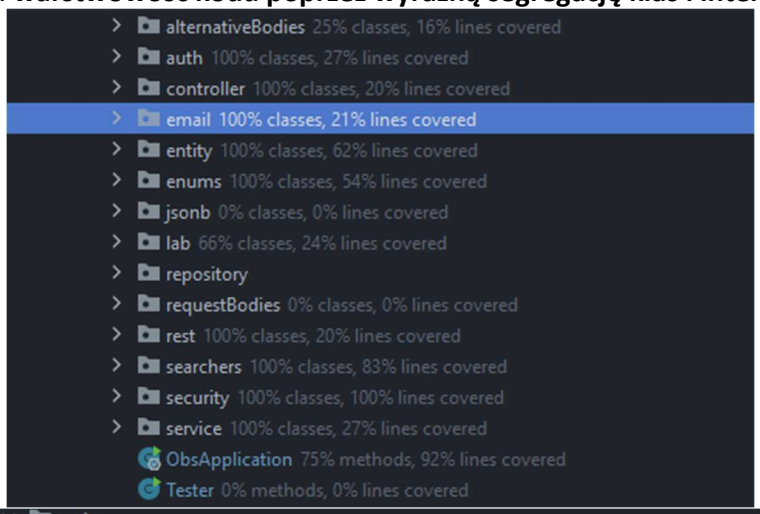
Przykład zaimplementowania tabelki przelewów

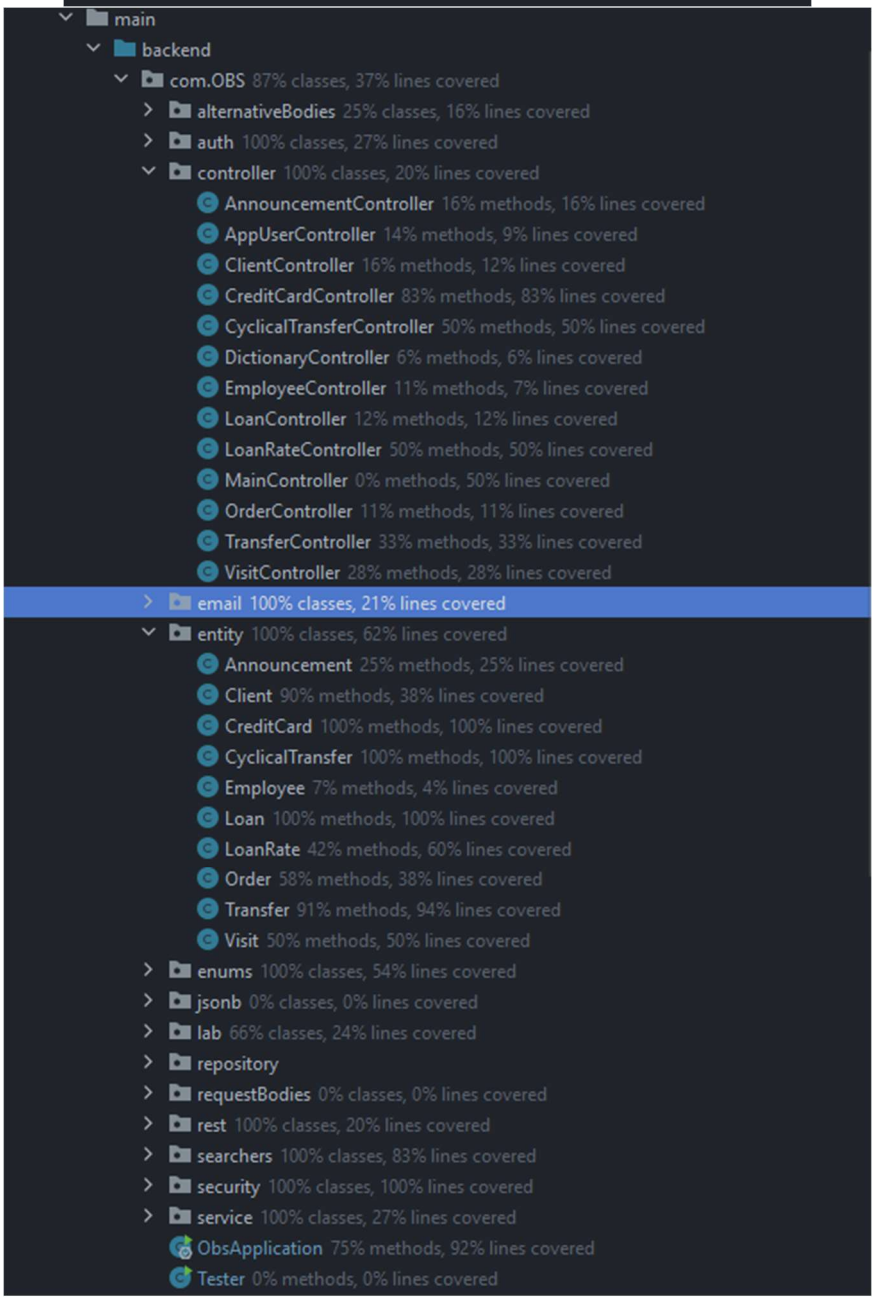
Tabela ostatnich przelewów

Data	Kategoria	Odbiorca / Nadawca	Kwota
30.12.2021	Wydatki Bieżące	Michał Tester	-1.00 PLN
28.12.2021	Wynagrodzenie	Michał Tester	-3.00 PLN
26.12.2021	Wynagrodzenie	s	-1.00 PLN
26.12.2021	Wynagrodzenie	s	-2.00 PLN
26.12.2021	Wynagrodzenie	Micas	-3.00 PLN
26.12.2021	Wydatki Bieżące	tatst	-4.00 PLN
26.12.2021	Rachunki	s	-1.00 PLN
26.12.2021	Wydatki Bieżące	s	-3.00 PLN
26.12.2021	Rachunki	s	-2.00 PLN
26.12.2021	Wynagrodzenie	s	-12.00 PLN
26.12.2021	Wydatki Bieżące	Mchał Tester	-5.00 PLN
26.12.2021	Wynagrodzenie	s	-3.00 PLN
26.12.2021	Rachunki	asdad	-3.00 PLN
26.12.2021	Rachunki	lhgfd	-1.00 PLN

Backend

- **Pisząc naszą aplikację serwerową postaraliśmy się stawiać na jak najlepsze praktyki programowania, bezpieczeństwa i organizacji pracy.**
    - **Podział na warstwowość kodu poprzez wyraźną segregację klas i interfejsów w podgrupy**





- **Zastosowanie Waterfall Spring Boot typu : Controller->Service->Repository przyniosło korzyści rozgałęzienia kodu dostępowego do serwera, kodu wykonującego logikę i operacje oraz metod dostępowych do bazy danych . np.:**

```
@RestController
@RequestMapping(path = "/dictionary")
@AllArgsConstructor
public class DictionaryController {
…

    @GetMapping(path = "/orders")
    public List<Order> getOrders() {
        return orderService.getOrders(ADMIN.name());
    }

    @GetMapping(path = "/orders/for-employees")
    public List<Order> getOrdersForEmployees(){ return
orderService.getOrders(EMPLOYEE.name());}

    …
}
```

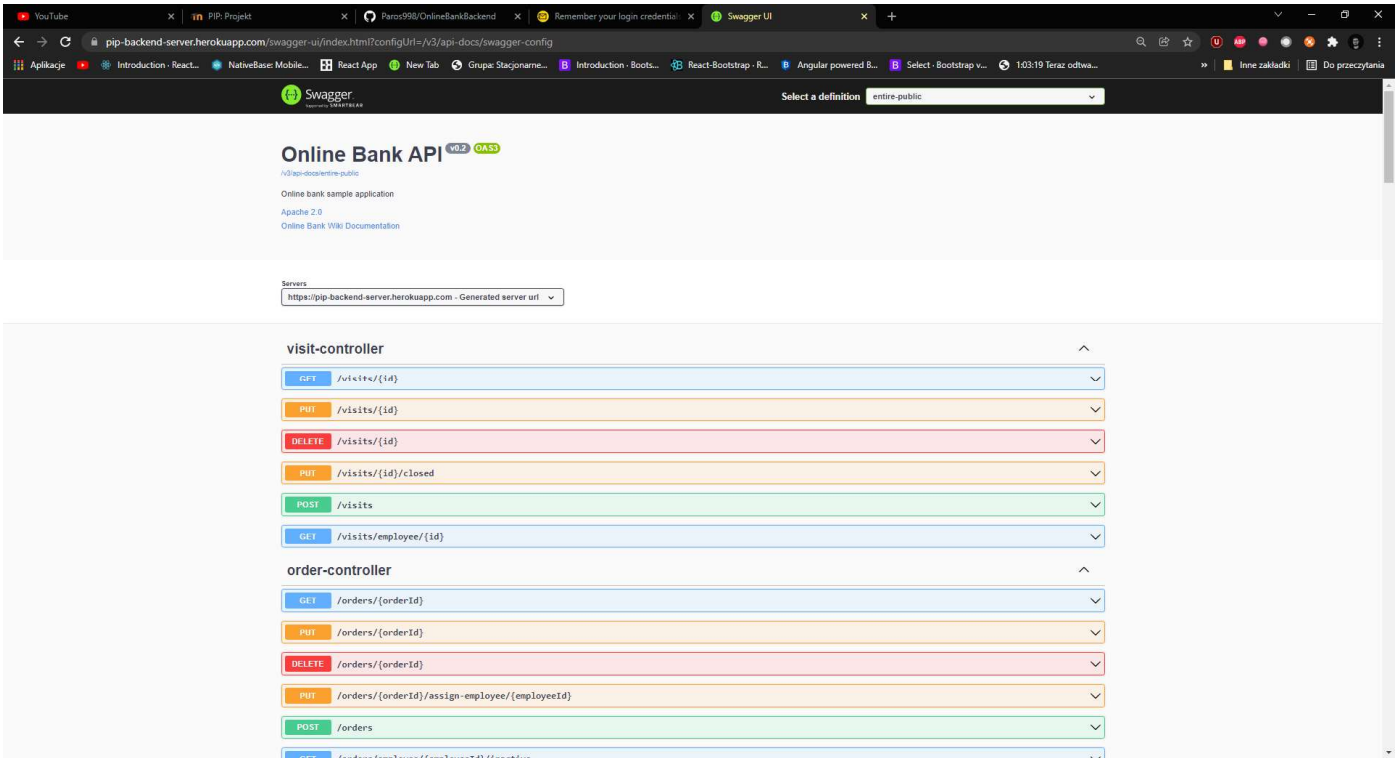
```
@Service
@RequiredArgsConstructor
public class OrderService {
        …
        public List<Order> getOrders(String role) {
             specification = new BuilderSpecification<Order>()
             .add(new SearchCriteria("createDate", LocalDateTime.now().minusDays(1),
                 SearchOperation.GREATER_THAN_DATE))
             .add(new SearchCriteria("employee", null, SearchOperation.EQUAL_NULL));
             if (!Objects.equals(role, ADMIN.name())) {
             ((BuilderSpecification<Order>) specification)
                .add(new SearchCriteria("orderType", changeEmployee.toString(),
                        SearchOperation.NOT_EQUAL))
                 .add(new SearchCriteria("orderType", changeUser.toString(),
                        SearchOperation.NOT_EQUAL))
                 .add(new SearchCriteria("orderType", createUser.toString(),
                        SearchOperation.NOT_EQUAL));
             }
             return orderRepository.findAll(specification,
                Sort.by(Sort.Direction.ASC,"createDate"));
         }
        …
}
```

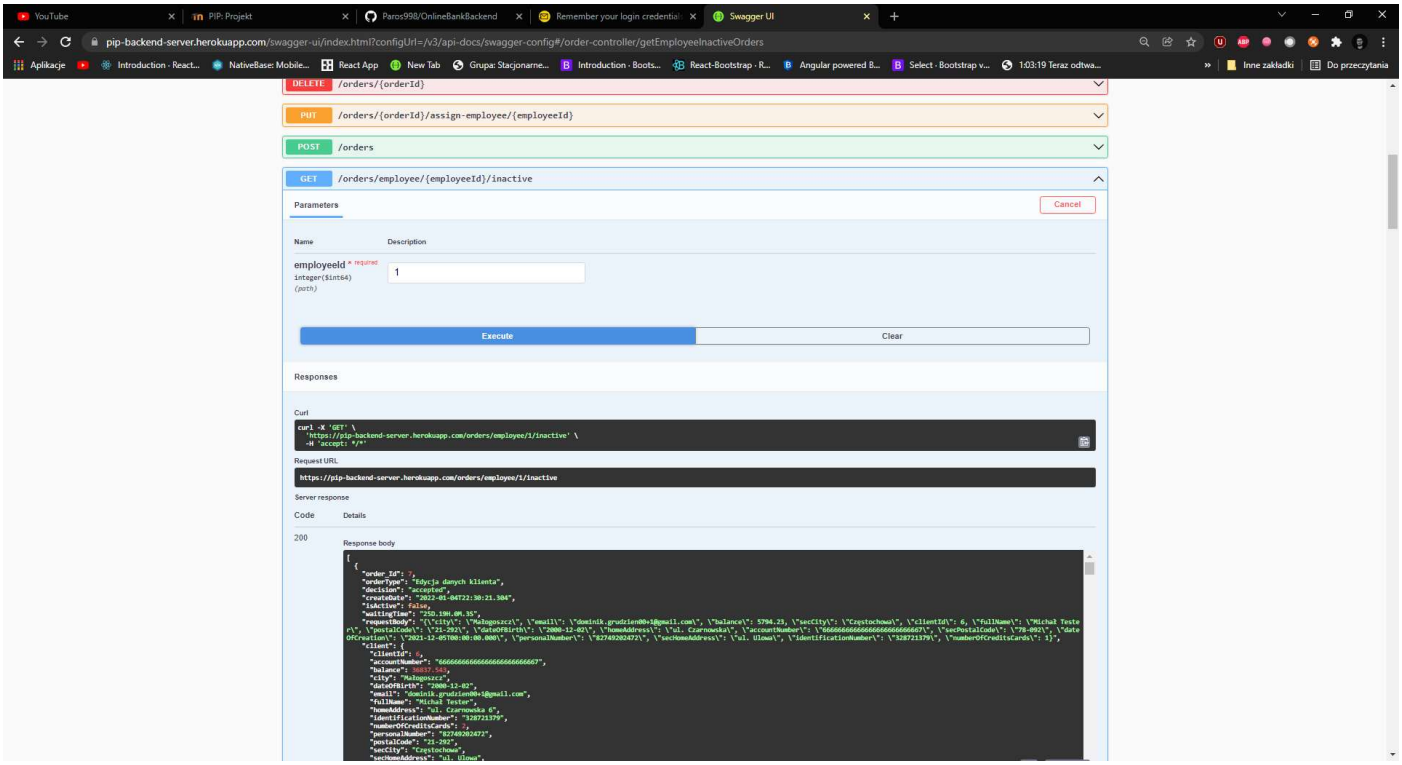
```
@Repository
public interface OrderRepository extends JpaRepository<Order, Long>,
                                         JpaSpecificationExecutor<Order> {
        …
}
```

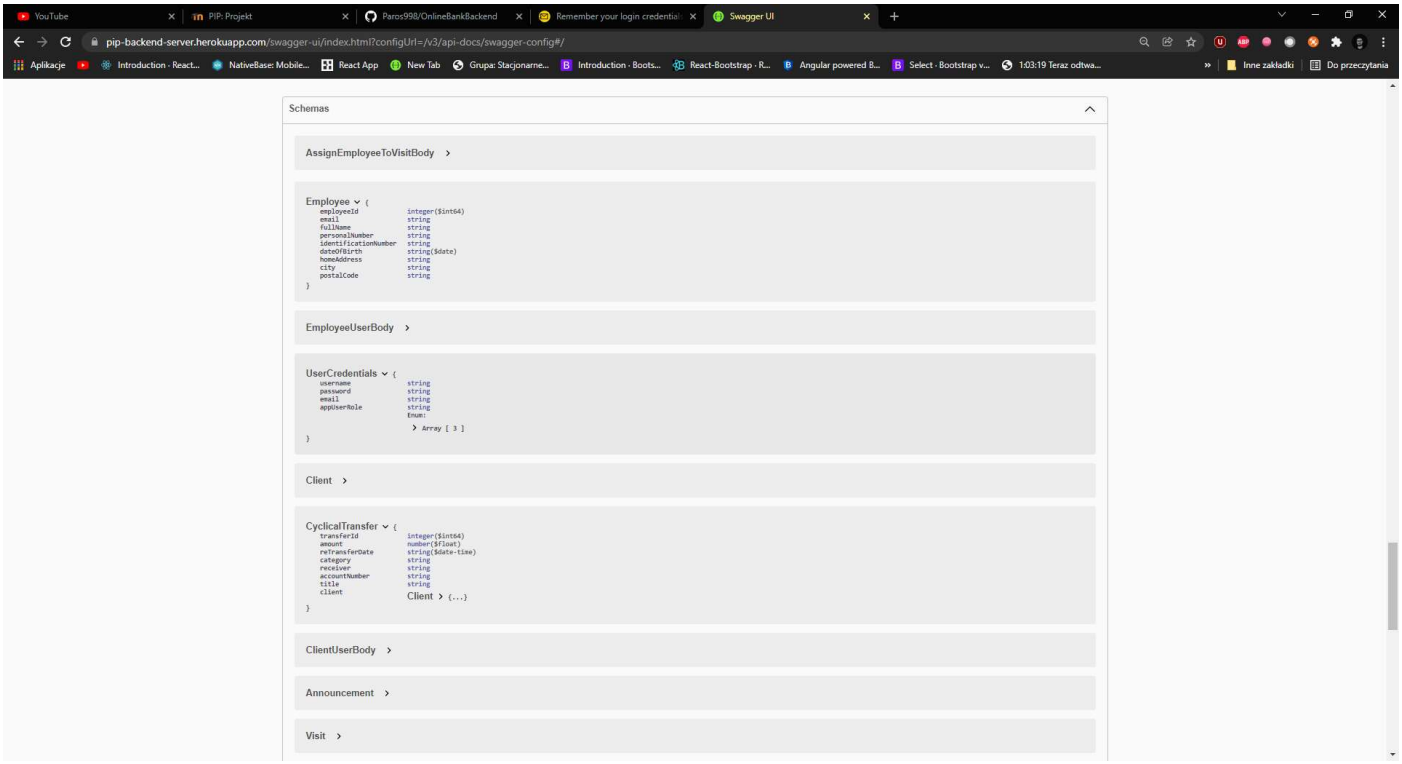
- **Zastosowanie JpaSpecificationExecutor w interfejsach dostępowych do bazy danych, pozwala nam na sprawne pisanie kryteriów mniej złożonych zapytań, które następnie mapowane są do zapytań bazodanowych. Np: przypadek wyżej lub przykład z parametrami dynamicznymi (czasami nawet pustymi) z aplikacji frontendowej.**

```
@RestController
@AllArgsConstructor
@RequestMapping(path = "/clients")
public class ClientController {
        …
        @GetMapping(path = "/filtered")
        public List<Client> getClientsSorted(@Conjunction(value = {
                @Or({
                   @Spec(path = "fullName", params = "personalNumber_personName",
                                spec = StartingWith.class),
                   @Spec(path = "personalNumber", params = "personalNumber_personName",
                                spec = StartingWith.class)
                }),
        }, and = @Spec(path = "dateOfBirth",
                        params = "birthDate",
                        spec = Equal.class)
                       ) Specification<Client>filterClientSpec) {
                return clientService.getClients(filterClientSpec);
        }
        …
}
```

- **Włączenie do aplikacji Swagger Open Api, pozwala nam na przyjazne wyświetlanie endpointów, struktur obiektów zwracanych oraz testowanie aplikacji . np.:**



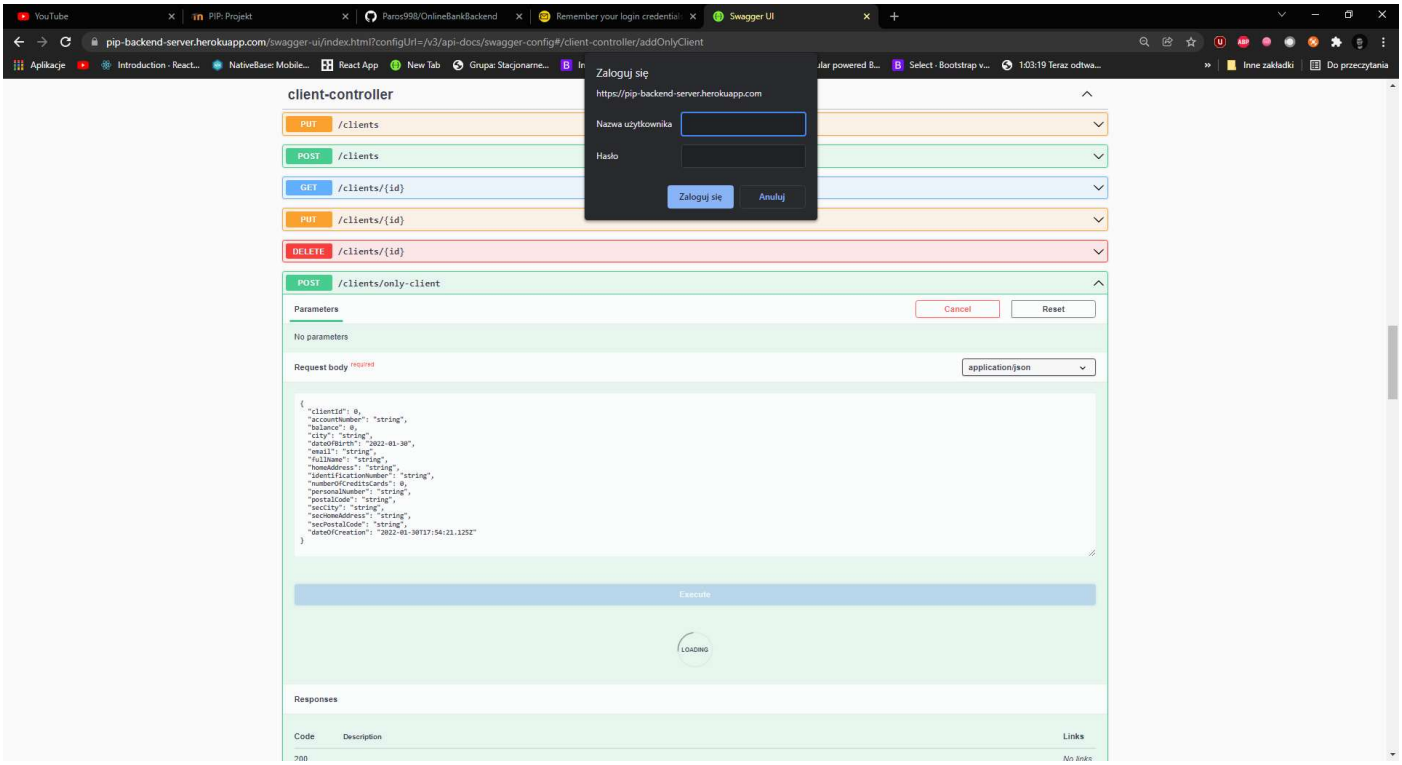

pip-backend-server.herokuapp.com/swagger-ui/index.html?configUrl=/v3/api-docs/swagger-config#/order-controller/getEmployeeInactiveOrders
DELETE /orders/{orderId}
PUT /orders/{orderId}/assign-employee/{employeeId}
POST /orders
GET /orders/employee/{employeeId}/inactive
Parameters
Cancel
Name
Description
employeeId * required
integer($int64)
(path)
1
Execute
Clear
Responses
Curl
curl -X 'GET' \
'https://pip-backend-server.herokuapp.com/orders/employee/1/inactive' \
-H 'accept: */*'
Request URL
https://pip-backend-server.herokuapp.com/orders/employee/1/inactive
Server response
Code
Details
200
Response body


pip-backend-server.herokuapp.com/swagger-ui/index.html?configUrl=/v3/api-docs/swagger-config#/
Schemas
AssignEmployeeToVisitBody
Employee
EmployeeUserBody
UserCredentials
Client
CyclicalTransfer
ClientUserBody
Announcement
Visit

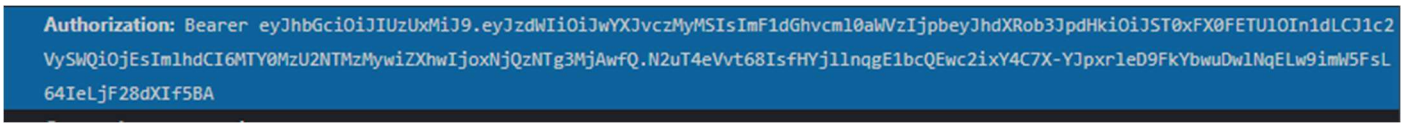
- **Bezpieczeństwo dostępu do serwera, baz danych, zasobów oraz przechowywania użytkowników**
  - **Dostęp do serwera jest możliwy tylko z wybranych lokalizacji w sieci**

```
corsConfiguration.setAllowedOrigins(Arrays.asList("http://localhost:3000",
        "http://localhost:5555",
        "http://localhost:19006" ,
        "https://pip-frontend-server.herokuapp.com"
));
```

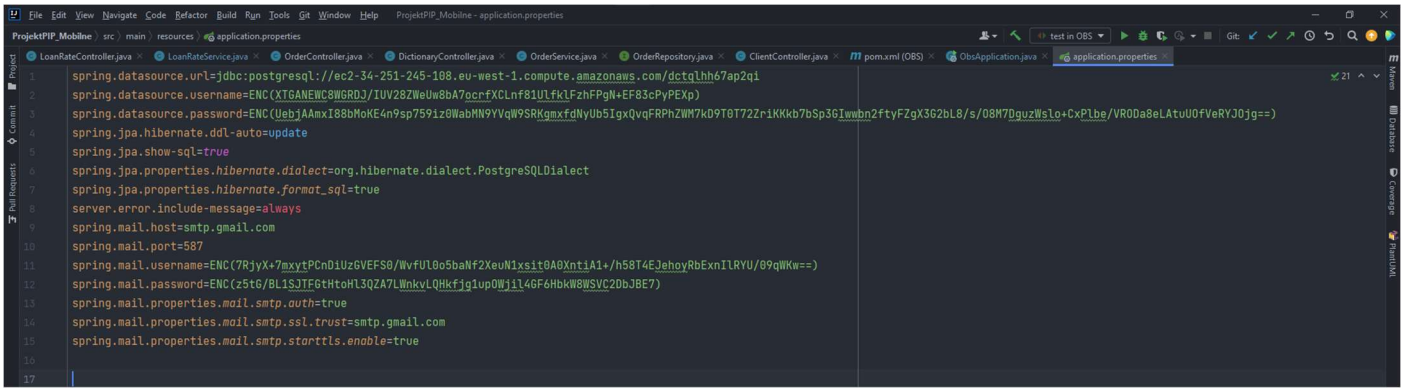
  - **Dwie metody uwierzytelniania użytkowników poprzez Basic oraz JWT**



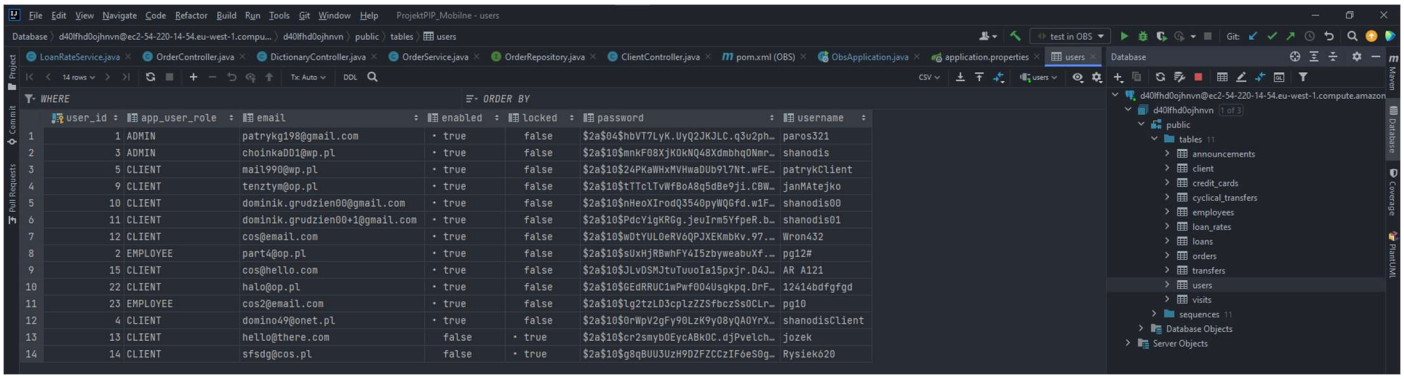
Token JWT jest tworzony po pomyślnym uwierzytelnieniu i jest wysyłany jako odpowiedź, dzięki temu przechowywując token, który jest ważny możemy uzyskiwać dostęp do zasobów bez konieczności kolejnego uwierzytelniania metodą Basic.



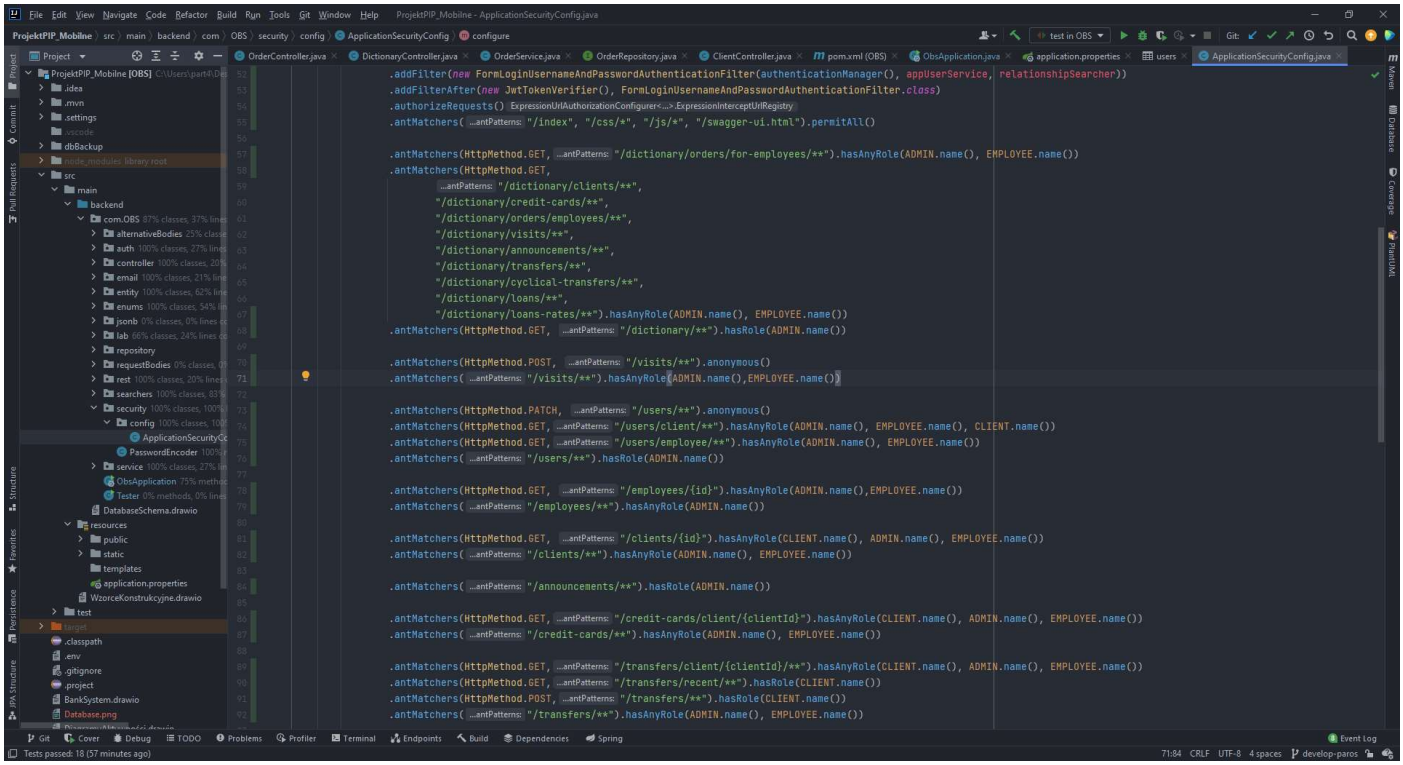
  - **Szyfrowanie parametrów połączeniowych do baz danych oraz innych usług wykorzystywanych przez aplikację za pomocą biblioteki Jasypt. Np.:**



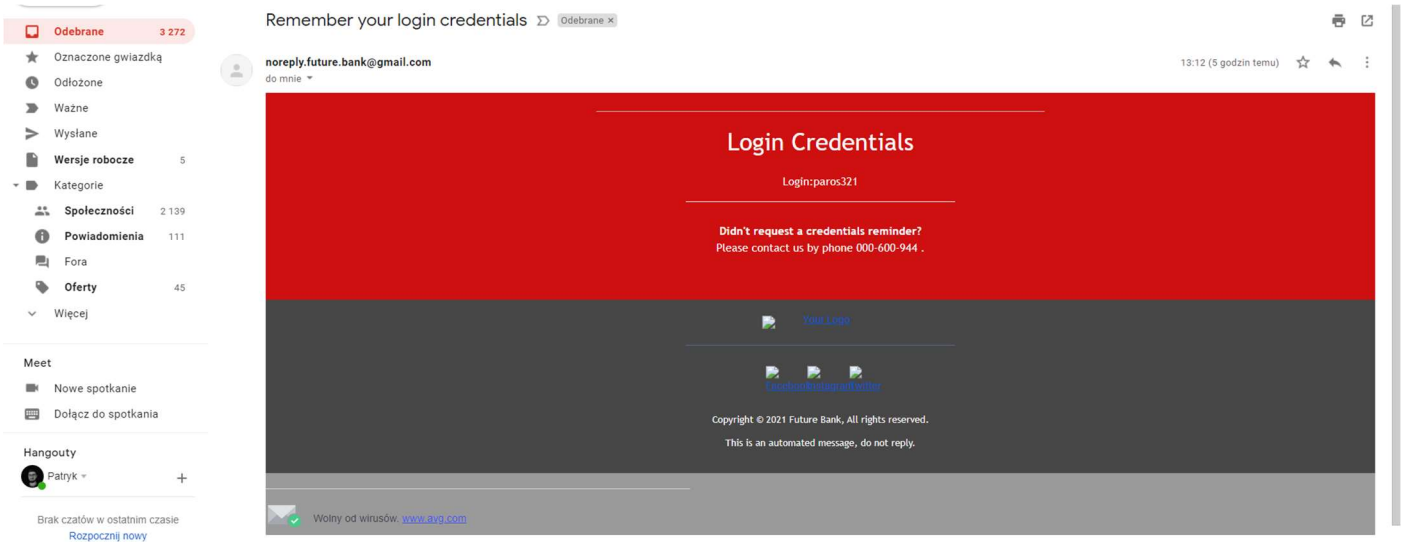
- **Szyfrowanie haseł użytkowników za pomocą BCryptPasswordEncoder kluczami RSA**



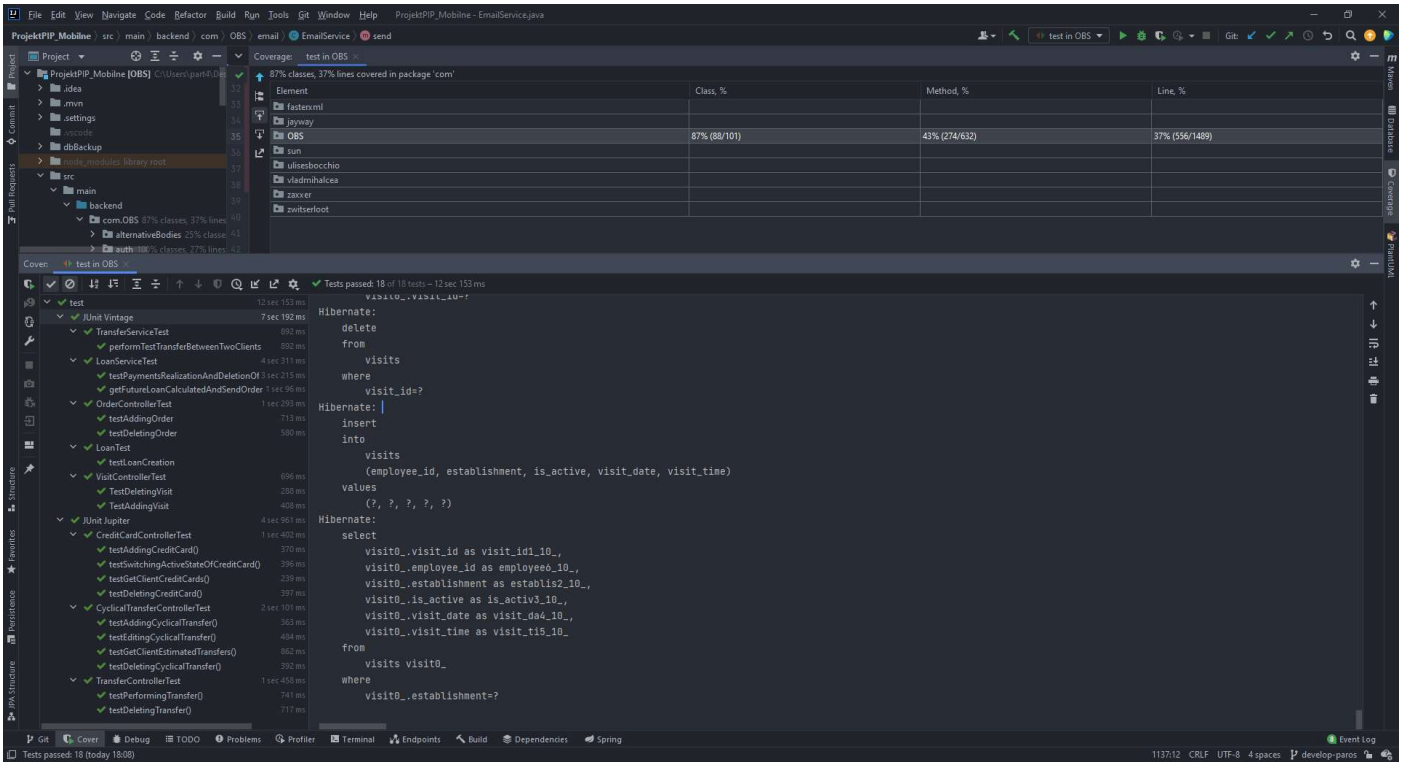
- **Role użytkowników usprawniają autoryzację dostępową do poszczególnych zasobów**



  - **Usługa wysyłki maili z informacjami do użytkowników poprzez SMTP Gmail**



- **Testy jednostkowe i integracyjne napisane w JUNIT 5**



- **Relacyjna baza danych PostgresSQL**