



Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Control Engineering and Information Technology

Ashwin Varma

DEVELOPING A PRICE TRACKER APPLICATION FOR ANDROID

SUPERVISOR

Norbert Somogyi, PhD student

BUDAPEST, 2022

Contents

Summary.....	4
1 Introduction.....	6
2 Applied Technologies.....	7
2.1 Android	7
2.2 Spring and Spring Boot.....	8
2.3 Java Persistence API	8
2.4 MySQL.....	9
2.5 REST	9
3 Design	10
3.1 Requirements	10
3.2 Data model	10
3.3 Architecture.....	11
4 Implementation	13
4.1 Backend – Java Spring Boot	13
4.1.1 DTO	14
4.2 Controllers – Service – Repository	15
4.2.1 Scheduler.....	19
4.3 Entities	20
4.4 pom.xml	21
4.5 Tests	22
4.6 Frontend – Android Application	23
4.6.1 Service.....	23
4.6.2 Activities	24
4.7 Utils.....	30
4.8 Model	31
4.9 Running the Application	32
5 Conclusion.....	38
6 References	39

STUDENT DECLARATION

I, **Ashwin Varma**, the undersigned, hereby declare that the present BSc thesis work has been prepared by myself and without any unauthorized help or assistance. Only the specified sources (references, tools, etc.) were used. All parts taken from other sources word by word, or after rephrasing but with identical meaning, were unambiguously identified with explicit reference to the sources utilized.

I authorize the Faculty of Electrical Engineering and Informatics of the Budapest University of Technology and Economics to publish the principal data of the thesis work (author's name, title, abstracts in English and in a second language, year of preparation, supervisor's name, etc.) in a searchable, public, electronic and online database and to publish the full text of the thesis work on the internal network of the university (this may include access by authenticated outside users). I declare that the submitted hardcopy of the thesis work and its electronic version are identical.

Full text of thesis works classified upon the decision of the Dean will be published after a period of three years.

Budapest, 9 December 2022

.....
Ashwin Varma

Summary

Nowadays, more and more people use online webshops. This thesis focuses on tracking prices of products for online shopping platforms. The application which is designed and introduced in this thesis is aimed at finding the notifications of a desired product's price changes. The application sends notifications when the prices of the products selected by the user changes and this gives the users an attainment of their needs as they are able to track the prices of their selected products.

The application is built on 3 main components. The android application (Kotlin) basically acts as a frontend which the user sees and interacts with. The server is created in SpringBoot which is a java framework to write APIs. Finally, the MySQL database stores the relevant information about users and products.

In conclusion, the application's goal is to create a price tracker service for the Android platform which will analyse the user trends and update the prices of the selected products. The changing trends of prices are also visualized, the basic information regarding the products is collected, and users are able to subscribe to a product and ask for a notification if its price drops below a certain threshold. This shall let the user have the privilege of getting the best price for their selected product.

सारांश

सारांश

एंड्रॉइड प्रोजेक्ट ऑनलाइन शॉपिंग प्लेटफॉर्म के लिए उत्पादों की कीमतों को ट्रैक करने पर केंद्रित है। इस थीसिस में डिज़ाइन और पेश किए गए एप्लिकेशन का उद्देश्य वांछित उत्पाद की कीमतों में बदलाव की अधिसूचनाएं ढूंढना है। उपयोगकर्ता द्वारा चुने गए उत्पादों की कीमतों में परिवर्तन होने पर एप्लिकेशन सूचनाएं भेजता है और इससे उपयोगकर्ताओं को उनकी आवश्यकताओं की प्राप्ति होती है क्योंकि वे अपने चयनित उत्पादों की कीमतों को ट्रैक करने में सक्षम होते हैं।

एप्लिकेशन मोटे तौर पर 3 घटकों पर बनाया गया है जो एंड्रॉइड एप्लिकेशन (कोटलिन), स्प्रिंगबूट (जावा) और माई एसक्यूएल (डेटाबेस) हैं। एंड्रॉइड एप्लिकेशन मूल रूप से एक दृश्यपटल के रूप में कार्य करता है जिसे उपयोगकर्ता देखता है और उसके साथ इंटरैक्ट करता है। उदाहरण के लिए इसे क्लाइंट भी कहा जा सकता है। उनमें से कई अपने व्यक्तिगत रूप से चयनित उत्पादों को ट्रैक करने के लिए सर्वर से जुड़े हो सकते हैं। सर्वर जो बैकएंड है, स्प्रिंगबूट में लिखा गया है जो एपीआई लिखने के लिए एक जावा फ्रेमवर्क है और एपीआई (एप्लीकेशन प्रोग्रामेबल इंटरफेस) को डेटाबेस और क्लाइंट ऐप (एंड्रॉइड ऐप) के बीच एक बिचौलिए के रूप में कहा जा सकता है।

अंत में, एप्लिकेशन का लक्ष्य Android प्लेटफॉर्म के लिए एक मूल्य ट्रैकर सेवा बनाना है जो उपयोगकर्ता के रुझान का विश्लेषण करेगी और चयनित उत्पादों की कीमतों को अपडेट करेगी। कीमतों के बदलते रुझानों की भी कल्पना की जाती है, उत्पादों के बारे में बुनियादी जानकारी एकत्र की जाती है, और उपयोगकर्ता किसी उत्पाद की सदस्यता लेने में सक्षम होते हैं और इसकी कीमत एक निश्चित सीमा से कम होने पर अधिसूचना मांगते हैं। इससे उपयोगकर्ता को अपने चुने हुए उत्पाद के लिए सर्वोत्तम मूल्य प्राप्त करने का विशेषाधिकार प्राप्त होगा।

1 Introduction

There exist several applications nowadays that facilitate users' buying behaviors. Applications that allow users to track the prices of various products are one example of this use-case. Since the advent of online shopping, there may be a large number of websites selling a specific product that a customer is interested in. In these situations, price tracking software frequently helps customers cut prices on their purchases significantly. In fact, features like showing historical trends in altering costs and keeping track of multiple webshops help consumers decide when and where to make purchases.

The goal of my thesis is to create one such application that will track the prices of the selected products of the users on multiple webshops. Originally, I intended to use the advertising API of *Amazon* [1] and *Ebay* [4]. However, because of strict user policies and regulations, these webshops have not granted access to their respective API-s. For this reason, two „fake API“-s were used [6]. These simulate online webshops with built-in products.

In my application, users should be able to look up, explore, and follow the prices of products on these websites using the application. Users should be able to subscribe to a product and request a notification if its price decreases below a specific threshold. Users should also be able to visualize the changing patterns in pricing and collect basic information about the products. It should be possible to browse, categorize, and filter items that have already been searched for by any user. It should also be possible to search for new products in the supported API-s.

The application is mainly built on 3 components:

1. Android application
2. SpringBoot backend
3. MySQL database

The android application acts as frontend which the user interacts with and can also be termed as the client. It communicates with the backend, and it shows the products to the users. It can also be termed as the client and we can have several clients connected to a server with their own preferences. The server communicates with the client, handles authentication and authorization, stores data in the MySQL database, and forwards user queries to the shopping API-s for data about the products. This thesis is structured as follows. Section 2 explains the ‘Applied Technologies’, in Section 3, I present the ‘Design phase of my work’, in Section 4, there is the ‘Implementation’, in Section 5, I conclude my work and lastly in Section 6, the ‘References’ are listed.

2 Applied Technologies

In this section, I present the most important technologies I used in my thesis.

2.1 Android

Android [9] is an operating system designed for creating applications on mobiles. The main building blocks of an android application are *activities*. An activity is a screen that holds local data and communicates with other components of the application. In android, activities follow a lifecycle presented in Figure 1.[7]

- onCreate() : You must execute this callback, which fires when the framework is to be created. On activity creation, the activity enters the Started state.
- onStart() : When the activity enters the Started state, the framework invokes this callback. The onStart() call makes the activity visible to the user, as the app plans for the activity to enter the foreground.
- onResume() : When the activity enters the Resumed state, it comes to the foreground, and after that the system invokes the onResume() callback. This is often the state in which the app interacts with the user. The app remains in this state until something happens to take focus away from the app.
- onPause() : The framework calls this state as the primary sign that the user is taking off the activity (in spite of the fact that it does not continuously mean the movement is being destroyed) it shows that the activity is now not within the foreground (in spite of the fact that it may still be visible in case the client is in multi-window mode).
- onStop() : When your activity is now not visible to the client, it has entered the Stopped state, and the framework conjures the onStop() callback. This may happen, for case, when a newly launched movement covers the whole screen.
- onDestroy() : When the movement moves to the destroyed state, any lifecycle-aware component tied to the activity's lifecycle will get the ON_DESTROY event.

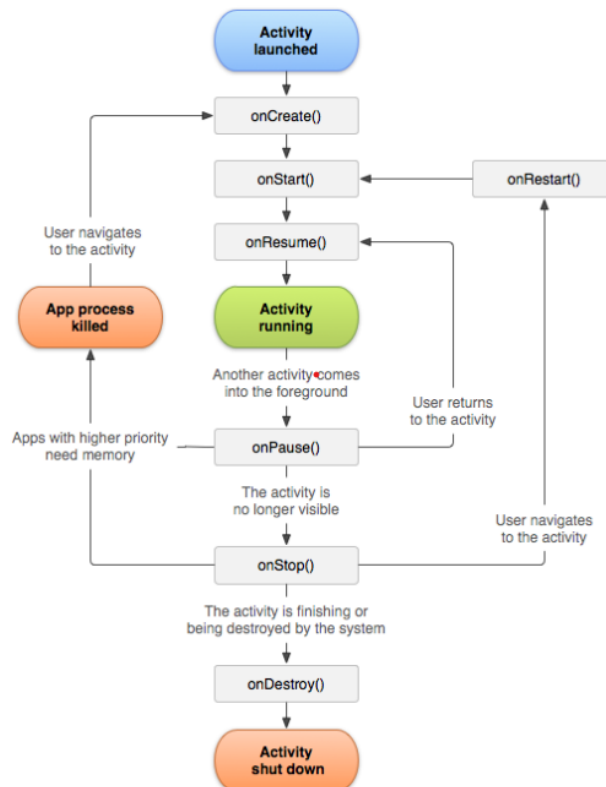


Figure 1 Android activity lifecycle (Kumar, 2022).

2.2 Spring and Spring Boot

Spring Boot [11] is a Java framework, built on top of the Spring framework [10], utilized for developing web applications. It permits the user to make REST APIs with negligible setups. In the event that the dependencies are there in your classpath, Spring Boot will auto-create the beans for it. Beans in Spring are objects that are instantiated and overseen by Spring through Spring IoC containers. The user does not have to make and design the beans as Spring Boot will do it for the user. The dependencies that were used :

- Spring Web: It is required for building Restful web applications.
- Spring Data JPA: It is required to get to the information from the database. JPA (Java Persistence API) is a Java Specification that maps Java objects to database entities, moreover known as ORM (Object Relational Mapping).
- MySQL Driver: required to associate with MySQL database.

2.3 Java Persistence API

Mapping Java objects to database tables is termed as Object-relational mapping (ORM) and the Java Persistence API (JPA) is one of the most conceivable approach to ORM. By means of JPA, the

developer can outline, store, upgrade and retrieve information from relational databases to Java objects.

2.4 MySQL

MySQL [12] is a relational database management system which can be used for storage of data for all computer program applications. For instance, at whatever point somebody conducts a web search, logs in to an account, or completes an exchange, a database framework is putting away the data so it can be accessed in the future.

The “SQL” term of “MySQL” stands for “Structured Inquiry Language.” SQL is the foremost common standardized language utilized to get to databases.

2.5 REST

REST stands for Representational State Transfer and it is a set of protocols/standards that portray how communication ought to take place between the computers and other applications over the network. For example, to have a better understanding let's assume the user needs to drive a car and he simply needs to utilize the accelerator, clutch etc. to drive it. The same way assume a Web App needs to communicate to a Web Server therefore, a Rest API uses GET, POST, PUT, Delete methods to communicate.

Subsequently, we can conclude that REST is an architectural pattern for planning network applications and it utilizes simple HTTP methods to communicate between clients and servers although it should be understood that HTTP and REST are not same.

3 Design

In this section, I present the design phase of the application, including the requirements, data model, UI wireframes, and the overall architecture.

3.1 Requirements

After analyzing the task description, I have identified the following requirements.

- Users should be able to register themselves so as to track their price trends individually.
- Users should be able to search for their desired products, browse all the available products, get the notifications for the price change alerts.
- The application asks for the threshold price and if the product's price drops below the threshold price then the user receives a notification.

3.2 Data model

The data model of the application is presented in Figure 2. The tables that we have are Users (basically API users), Saved Products (the products that the user selects) and Prices (the prices of the products that go below the threshold price). The Users table has the UserID as the primary key which acts as a foreign key in the Prices table. The motivation behind this is to track the prices for different users respectively. The Saved Products table has the primary key ProductID and also acts as a foreign key in Prices table.

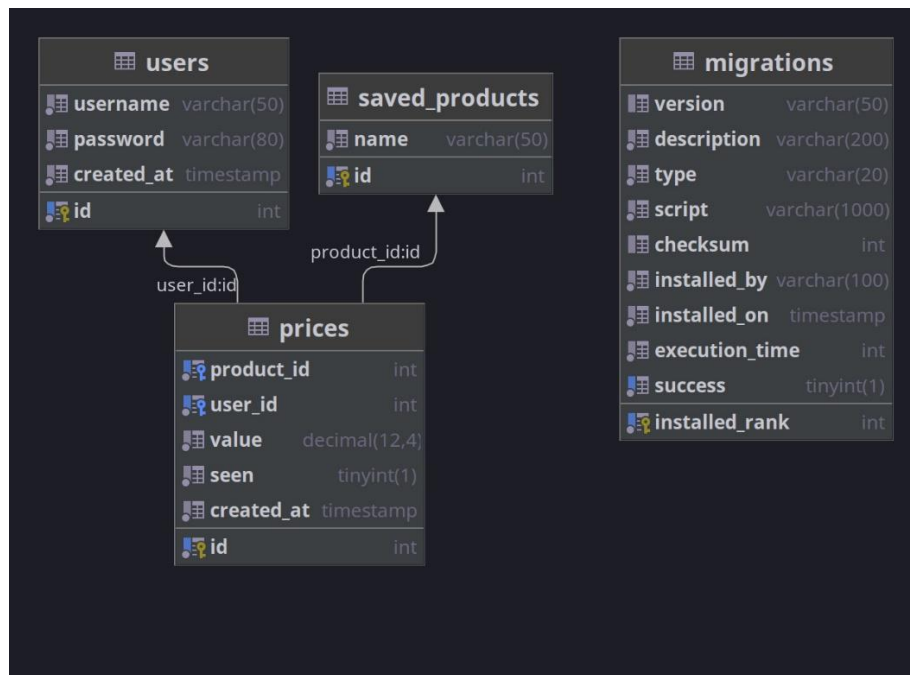


Figure 2 Entity Diagram (data model)

The ‘migrations’ can be basically termed as when we change something on the database so the database can document the changes, for example addition or removal. Migrations can be understood as we move information from source databases to target databases.

3.3 Architecture

The backend of the application follows the Controller – Service – Repository architecture/pattern [3] This architecture is really useful as it separates the concerns of processes for the backend. Figure 3 illustrates the architecture.

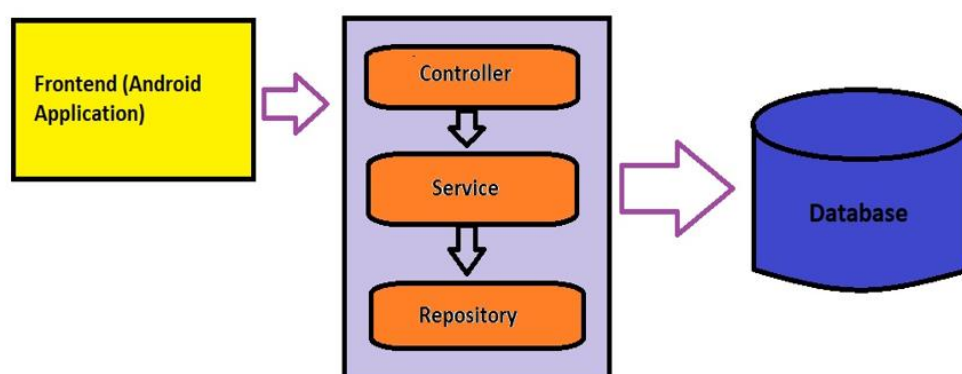


Figure 3 Architecture of Application

The controller (also called the top-layer) basically is used for getting the requests and providing the REST interface. It sends as well as fetches the response to and from the service respectively. The service layer can also be termed as the most crucial layer as this is where the business logic resides. It runs the functions that are supposed to be done, how to be done and when to be done. It can also be termed as the implementation layer. The repository communicates with the database, fetching and writing data when needed.

4 Implementation

As mentioned earlier, the project mainly constitutes of 3 parts (Frontend – Android application, Backend – SpringBoot, Database – MySQL). Let's understand each part starting with the backend (SpringBoot) which makes all our required heavy lifting and which helps the user/client to execute their processes.

4.1 Backend – Java Spring Boot

In the most basic description, we have the server which is written in SpringBoot which is a Java framework to write APIs.

The application.properties file contains the configurations used in the project.

```
spring.datasource.url=jdbc:mysql://localhost:3306/price_tracker
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.show-sql=true

server.port=5000
server.error.whitelabel.enabled=false

spring.task.scheduling.pool.size=2

spring.jackson.property-naming-strategy=SNAKE_CASE

spring.flyway.enabled=true
spring.flyway.baseline-on-migrate=true
spring.flyway.locations=classpath:migrations
spring.flyway.table=migrations

def.JWTSecret=SOME_SECRET
```

The configurations mean the following :

The access to the datasource which stores/fetches the data has been given the initial username and password as 'root' which has to be entered when trying to access the database in MySQL as shown below in image Figure 4 :-

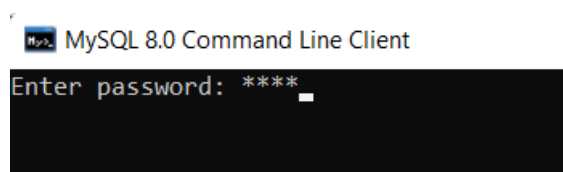


Figure 4 MySQL Command line client

The JDBC-Driver is the program that interacts with the database. There are 4 types of drivers, JDBC-ODBC bridge driver, Native-API driver (partially java driver), Network Protocol driver (fully java driver) and Thin driver (fully java driver). It is convenient to use and can be connected to any database.

The sql line basically means that the below condition is set to true then you show the database, this can also be used to hide the data from the external users with unauthorized access.

This app is hosted on a port which can be specified, that can be termed as some kind of a door inside the computer which listens to requests (Transfer of data) in the most basic explanation. In the event that we have an application deployed in totally different environments, we may need it to run on distinctive ports on each system. We can effortlessly accomplish this by combining the property files approach with Spring profiles. Particularly, able to make a property record for each environment. The data will be fetched with the snake case naming convention which is one of the 3 naming conventions (camelCase, PascalCase, snake_case).

The flyway is what basically builds the database. It overhauls a database from one form to the another utilizing migrations which are the documented changes in the database.

The class 'FlywayConfiguration' utilizes its functionalities and sets the migration strategies including repairing a failed migration. The code can be seen below.

```
@Configuration
public class FlywayConfiguration {

    @Bean
    public FlywayMigrationStrategy repairMigrationStrategy() {
        return flyway -> {
            flyway.repair();
            flyway.migrate();
        };
    }
}
```

All of these configurations can be seen in the 'application.properties' file in 'resources' tab.

4.1.1 DTO

A Data Transfer Object is an object that's utilized to typify information so to say, and send it from one subsystem of an application to another or basically we can call it a class that represents data which can be transferred. DTOs are generally utilized by the Services layer in an N-[5] (GeeksForGeeks, 2017) Tier application to exchange information between itself and the UI layer. The most important advantage here is that it reduces the sum of information that needs to be sent over the wire in distributed applications. The project uses a number of DTO-s, including the user

credentials (AuthRequest), JWT token (TokenDTO) and the id of the price we want to monitor (IdDTO).

4.2 Controllers – Service – Repository

Let's understand more about the Controller – Service – Repository pattern/architecture in detail and how they use the DTOs. In the project, the Controller-Service-Repository pattern works with the use of a strategy/design pattern called 'Dependency Injection'. Dependency Injection (DI) is of many types but the DI we are using is called Constructor DI. Simply put, Dependency Injection is a principal angle of the Spring framework, through which the Spring container "injects" objects into other objects or "dependencies". In the project, the service is getting injected in the controller and the repository is getting injected in the service.

```
public AuthController(AuthenticationManager authenticationManager,
    AuthService authService) {
    this.authenticationManager = authenticationManager;
    this.authService = authService;
}
```

The above code is for the class 'AuthController' which is the controller responsible for getting the request and calling the service. All the controller classes in SpringBoot are annotated by the keyword @RestController or @Controller because these stamp controller classes as a request handler to permit Spring to recognize it as a Restful service amid runtime. The @Controller annotation can be recognized as a superset of the @Component annotation which tells Spring that it needs to be managed. The @Controller annotation expands the use-case of @Component and marks the annotated class that has been marked as a business or introduction layer, so when a request is made, this will illuminate the DispatcherServlet to incorporate the controller class in checking for strategies mapped by the @RequestMapping annotation.

Now, we'll announce the real controller to characterize the business logic and handle all the demands related to the model. First, stamp the course with the @Controller comment along side @RequestMapping and indicate the path as /api/auth in our case. The @Autowired annotation is utilized to consequently inject conditions of the desired type into the current bean. In this case, the AuthRepository bean is infused as a dependency of AuthController. The code above can justify the use of the same :

```
@PostMapping("/register")
public ResponseEntity<TokenDTO> register(@RequestBody AuthRequest request) {
    return ResponseEntity.ok(authService.register(request));
}

@PostMapping("/login")
```

```

public ResponseEntity<TokenDTO> login(@RequestBody AuthRequest request) {
    return ResponseEntity.ok(authService.getToken(request));
}

@PostMapping("/logout")
public ResponseEntity<String>logout(HttpServletRequest request, HttpServletResponse
response) {
    SecurityContextLogoutHandler securityContextLogoutHandler = new
    SecurityContextLogoutHandler();
    securityContextLogoutHandler.logout(request, response, null);
    return ResponseEntity.ok().build();
}

```

@PostMapping acts as an alternate route for @RequestMapping(method = RequestMethod.POST). Getting requests have a status code which represents the manifestation or successful completion of the desired request. Some of these are: 200 – OK, 400 – Bad Request, 401 – Unauthorized.

The ‘AuthController’ is used for the users registration, login and logout. It is basically used to track different users registration, activities etc. It calls the ‘AuthService’ which uses JWT (JSON Web Token) to register the users and track them. The JWT or JSON Web Token is a 2-way encryption standard. We can think of it as stateless which means that we are getting the info from the client and need not be saved on the server. It is provided by the class ‘JwtTokenProvider’, it has method ‘createToken()’ which creates a jwtToken with the user credentials and sets the validity of the token which is the expiration of the token.

```

public String createToken(Long id, String username) {
    Date now = new Date();
    Date validity = new Date(now.getTime() + 15 * 24 * 60 * 60 * 1000);

    return Jwts.builder()
        .setIssuedAt(now)
        .setSubject(username)
        .setExpiration(validity)
        .signWith(SignatureAlgorithm.HS512, definedProperties.getJWTSecret())
        .setId(String.valueOf(id))
        .compact();
}

```

It has the method ‘validateToken()’ which checks the validity of the token.

```

public boolean validateToken(String token) {
    try {
        return !jwtParser.parseClaimsJws(token).getBody().getExpiration().before(new
Date());
    } catch (JwtException | IllegalArgumentException exception) {
        throw new BadCredentialsException("Invalid token");
    }
}

```

The method ‘getAuthentication()’ is used to see which user is using it so that we can track the activities of different users.


```

public Authentication getAuthentication(String token) {
    APIUser user = APIUserDetails.loadUserById(getUserId(token));
    return new UsernamePasswordAuthenticationToken(user, null, null);
}

```

‘loadUserById()’ method loads the user by ID and ‘getUserId()’ method gets the ID of the user. Finally, the ‘resolveToken()’ method generates the Authorization header in the format “Authorization: Bearer <token>”.

```

public String resolveToken(HttpServletRequest request) {
    final String authHeader = request.getHeader("Authorization");

    if (authHeader == null || !authHeader.startsWith("Bearer")) {
        return null;
    }

    return authHeader.replace("Bearer", "").trim();
}

```

‘JwtTokenFilter’ is basically a processing unit that puts tokens in a stack. Its main function is to check if there is a token. It uses the ‘doFilter()’ method to do so. The code below explains it in detail :

```

public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,
    FilterChain filterChain) throws IOException, ServletException {
    String token = jwtTokenProvider.resolveToken((HttpServletRequest)
    servletRequest);

    try {
        if (token != null && jwtTokenProvider.validateToken(token)) {
            Authentication authentication = jwtTokenProvider.getAuthentication(token);

            if (authentication != null) {
                SecurityContextHolder.getContext().setAuthentication(authentication);
            }
        }
    } catch (AuthenticationException e) {
        SecurityContextHolder.clearContext();
        throw new BadCredentialsException("Invalid token");
    }
}

```

The ‘AuthService’ does the main processing and uses the method ‘getToken()’ to create the token and this is used by the ‘register()’ method to register the user in the database using the entity ‘APIUser’.

```

public TokenDTO getToken(AuthRequest request) {
    APIUser user = authUserRepository.findByUsername(request.username())
        .orElseThrow(() -> new APIException(HttpStatus.UNAUTHORIZED));

    return new TokenDTO(jwtTokenProvider.createToken(user.getId(),
    user.getPassword()));
}

```

```

public TokenDTO register(AuthRequest request) {
    if (authUserRepository.findByUsername(request.username()).isPresent()) {
        throw new APIException(HttpStatus.BAD_REQUEST);
    }

    APIUser user = new APIUser();
    user.setUsername(request.username());
    user.setPassword(passwordEncoder.encode(request.password()));
    user.setCreatedAt(LocalDateTime.now());
    authUserRepository.save(user);

    return getToken(request);
}

```

‘ConnectionController’ basically checks if there is a valid connection.

```

@RestController
@RequestMapping(value = "/api/connection")
public class ConnectionController {

    @GetMapping("/check")
    public ResponseEntity<String> checkConnection() {
        return ResponseEntity.ok().build();
    }
}

```

Now, let’s move onto ‘PricesController’ which gets the changed prices of the products if the price changes and checks if the product’s price changing notification is seen by the user or not. You can see in the below code, the requests are mapped with their respective functions.

```

@GetMapping("/changes")
public ResponseEntity<PriceDTO> getPrices() {
    return ResponseEntity.ok(priceService.getPrices());
}
@PostMapping("/seen")
public ResponseEntity<String> readPrice(@RequestBody IdDTO request) {
    priceService.seePrice(request);
    return ResponseEntity.ok().build();
}

```

The ‘PriceService’ has the ‘getPrices()’ method that maps all the columns of the table ‘price’ for all the respective users so that we can track each users activities.

```

public PriceDTO getPrices() {
    Object principal =
    SecurityContextHolder.getContext().getAuthentication().getPrincipal();

    if (principal instanceof APIUser user) {
        List<PriceDTO.PriceEntityDTO> priceList =
        priceRepository.findAllBySeenFalseAndProduct_User(user)
            .stream()
            .map(price -> new PriceDTO.PriceEntityDTO(
                price.getId(),
                price.getProduct().getName(),
                price.getValue(),
                Utils.getStringFromLocalDT(price.getCreatedAt())
            )
        )
    }
}

```

```

        ))
        .toList();

        return new PriceDTO(pricelist);
    } else {
        throw new APIException(HttpStatus.INTERNAL_SERVER_ERROR);
    }
}

```

When we swipe the price change alert notification on the android application, the status of the price change notification changes to 'seen' which is executed by the 'seePrice()' method, which sets the price change notification to 'seen'. The 'ProductsController' is used for saving the chosen products and getting the products from the API (will be discussed later).

The 'ProductsService' gets the products from the APIs and stores them in the database basically in the 'products' table. The code is similar to the 'PriceController' and 'PriceService'. When we try to save the product we need to monitor the price of, we need to enter the threshold price and if the price of the product goes below that then we will be notified.

4.2.1 Scheduler

One of the most crucial parts of the project is the scheduler. It basically fetches the prices of the products at a fixed time to check if there is any change in the prices of the saved product and if there is then it notifies the user about the same. The 'PriceScheduler' class is used to set the fixed time for the scheduler by using the updatedPrices() method [`@Scheduled(fixedRate = 60 * 1000)`] which means that in every 1 minute, the scheduler will run and check the prices of the saved products.

The 'PriceSchedulerPriceRepository' and 'PriceSchedulerProductRepository' are interfaces which return the list of prices and products respectively. The 'PriceSchedulerService' is where the major processing and all the heavy-lifting happens. It implements the method 'fetchUpdatedPrices()' which gets the new prices of the products after we simulate the price change ourselves using the method 'simulatePriceChange'.

```

public void fetchUpdatedPrices() {
    final List<SenderProductDTO> updatedProductsList = Stream.of(
        dummySender.getProducts(null), fakeSender.getProducts(null)
    )
    .flatMap(Collection::stream)
    .toList();

    final List<String> titleList = updatedProductsList
        .stream()
        .map(SenderProductDTO::title)
        .toList();
}

```

```

        final List<Product> productList = productRepository.findAllByNameIn(titleList);
    }

```

The 'simulatePriceChange()' method takes the new product and changes its price with a random percent change (centered between -5 to 5) and uses that price as the new price of the product.

```

private SenderProductDTO simulatePriceChange(SenderProductDTO updatedProduct) {
    int changePercent = random.nextInt(11) - 5; // random number from 0 to 10, then
    we center to [-5,5]

    BigDecimal priceDiff = updatedProduct.price()
        .multiply(BigDecimal.valueOf(changePercent))
        .divide(BigDecimal.valueOf(100), RoundingMode.HALF_EVEN);

    BigDecimal newPrice = updatedProduct.price().add(priceDiff);
    return new SenderProductDTO(updatedProduct.title(), updatedProduct.description(),
    newPrice);
}
}

```

4.3 Entities

There are 3 entities which can also be thought of as 'tables' in the database which stores are data for the user called 'ApiUser' which saves the userID, Username, password and time of user creation. The second one is called 'Price' which saves the ID, Value, seen (which is 1 if its true and 0 if its false in the table) and finally the third one called 'Product' which saves the ID, Name of the product, description and threshold value (which will be entered by the user).

The code for the 'Product' entity is below :

```

@Entity
@Table(name = "products")
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private String description;

    @Column(nullable = false)
    private BigDecimal threshold;

    @OneToOne(optional = false)
    @JoinColumn(name = "user_id", nullable = false)
    private APIUser user;

    @OneToMany(mappedBy = "product", cascade = CascadeType.MERGE)
    private List<Price> priceList;

    public void setId(Long id) {
        this.id = id;
    }
}

```

```

    }

    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public BigDecimal getThreshold() {
        return threshold;
    }

    public void setThreshold(BigDecimal threshold) {
        this.threshold = threshold;
    }

    public APIUser getUser() {
        return user;
    }

    public void setUser(APIUser user) {
        this.user = user;
    }

    public List<Price> getPriceList() {
        return priceList;
    }

    public void setPriceList(List<Price> priceList) {
        this.priceList = priceList;
    }
}

```

4.4 pom.xml

The full form of pom is ‘Project Object Model’ which is used by maven to build the project. It is an xml file which contains information about the project and configuration details. The code can be seen below which shows some of the dependencies used by our project.

```

</dependency>
<dependency>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-core</artifactId>
    <version>8.5.13</version>
</dependency>

<dependency>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-mysql</artifactId>
    <version>8.5.13</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.30</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
</dependency>

```

4.5 Tests

The tests are made in the backend in Spring Boot. There are 3 unit tests that concern the JWT token and use HTTP post and get methods to get the products from our api.

```

@Test
public void productRequest_WithGivenQuery_ShouldIncludeMatches() throws Exception {
    final String query = "Watch";

    final String token = makeAuthRequest();
    final String headerValue = "Bearer " + token;

    mockMvc
        .perform(post("/api/products/get")
            .content("{ \"name\" : \"%s\" }".formatted(query))
            .header("Authorization", headerValue)
            .header("Content-Type", "application/json"))
        .andExpect(status().isOk())
        .andExpect(result -> {
            ProductResponseDTO response =
                jsonMapper.readValue(result.getResponse().getContentAsString(), new TypeReference<>()
                    {});

            assertNotNull(response, "Response not null");
            assertTrue("Result has no elements", !response.products().isEmpty());
        });
}

```

```

        for (ProductResponseDTO.ProductDTO product : response.products()) {
            final boolean containsQuery =
                product.title().toLowerCase().contains(query.toLowerCase()) ||
                product.description().toLowerCase().contains(query.toLowerCase());

            assertTrue("Does not contains query string", containsQuery);
        }
    });
}

```

There are 2 more products created that were used to test the change in price trends called ‘Tasty Product’ and ‘Fancy Product’.

4.6 Frontend – Android Application

Now, let’s move on to the frontend, which is the android application. We can start analysing the services layer which plays a major role initially.

4.6.1 Service

This folder has ‘AlarmSchedulerService’, ‘FetcherService’ and ‘FetcherReceiver’.

The ‘FetcherService’ has an intent service which runs in the background that runs an operation and return something back. Basically, it does the requests and calls the ‘FetcherReceiver’ which is basically a set of rules so to say by which any other class can use to interact with the ‘FetcherService’.

Let’s suppose we have any class that wants to send a request, what it does is that it implements this interface and gets that method and sends that receiver like itself to the ‘FetcherService’ that does the request and that in turn when it finishes it calls upon that method.

For instance, let’s imagine that we have a button then it needs to know when we click it and run the method what we provide to it. So, what we do is that we create a class that implements the interface that has this kind of onClick() method and then we pass this class with the onClick() method to that button which only cares about the listener and It doesn’t care where it came from, it just cares that there’s this method and its been passed from the outside. So, in this case it can be said that the button is the ‘FetcherService’ which does requests and acts as a button would do and our activity for example needs to react to the click of the button so it connects itself like it implements the interface and sends itself to the ‘FetcherService’ as the method. Therefore, the ‘FetcherService’ does not see that if it is an activity, it just checks if there is the method there. In this case, the initiator of the requests and the handler of the requests are the same object, its just that they are simply under different interfaces.

The code for the 'FetcherService' can be seen below :

```
class FetcherService : IntentService(FetcherService::class.java.simpleName) {

    override fun onHandleIntent(workIntent: Intent?) {
        Log.d(DEBUG_TAG, "AlarmService: Handling work intent")

        val method: Int = workIntent!!.getIntExtra("method", Request.Method.GET)
        val path: String = workIntent.getStringExtra("path")!!
        val resultReceiver: ResultReceiver =
workIntent.getParcelableExtra("receiver")!!
        val data: String? = workIntent.getStringExtra("data")

        val jsonObjectRequest = JsonRequest(
            method, path, data?.let{ JSONObject(it) },
            { response -> resultReceiver.send(200, getResponseBundle(workIntent,
response)) },
            { error -> resultReceiver.send(500, getResponseBundle(workIntent, error))
        }

    )

    Connector.getInstance(this).addToRequestQueue(jsonObjectRequest)
}

private fun getResponseBundle(workIntent: Intent, response: Any?): Bundle {
    val bundle = Bundle()
    bundle.putString("action", workIntent.action)
    bundle.putString("response", response?.toString())
    return bundle;
}
}
```

The code for the 'FetcherReceiver' can also be seen below :

```
class FetcherReceiver(handler: Handler?, private var responseHandler:
ResponseHandler) : ResultReceiver(handler) {

    override fun onReceiveResult(resultCode: Int, resultData: Bundle?) {
        Log.d(DEBUG_TAG, "FetcherReceiver: Received data passed to interface
implementation")
        responseHandler.handleResult(resultCode, resultData)
    }

}
```

4.6.2 Activities

An activity gives the window in which the app draws its User Interface. For the most part, one activity executes one screen in an app. For example, one of an app's exercises executes the details page whereas one of them executes the list of saved products.

4.6.2.1 LoginActivity :

This activity has the user's login page and register user button that has the intent to take us to the 'RegisterActivity' which is used for registering the user which saves the user credentials.

It authenticates the user with the entered username and password and if the token matches then it takes the user to the 'NavActivity'.

```
private fun authenticate(username: String, password: String) {
    val body = JSONObject()
    body.put("username", username)
    body.put("password", password)

    val loginRequest = JsonRequest(
        Request.Method.POST, LOGIN_ENDPOINT, body,
        {
            Model.saveToken(it.getString("token"))

            val intent = Intent(this, NavActivity::class.java)
            startActivity(intent)
            finish()
        },
        {
            Toast.makeText(this, "An error occurred during login!",
                Toast.LENGTH_SHORT).show()
        }
    )
}
```

4.6.2.2 RegisterActivity

This activity is used to create a user with its username and password.

```
class RegisterActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_register)

        val username = findViewById<TextView>(R.id.registerUser)
        username.text = intent.getStringExtra("username").toString()

        findViewById<Button>(R.id.registerBtn).setOnClickListener {
            val password = findViewById<TextView>(R.id.registerPass).text.toString()
            val passwordConfirm = findViewById<TextView>(R.id.registerConfirm).text.toString()

            if (password != passwordConfirm) {
                Toast.makeText(this, "Passwords do not match!",
                    Toast.LENGTH_SHORT).show()
                return@setOnClickListener
            }

            register(username.text.toString(), password)
        }
    }

    private fun register(username: String, password: String) {
        val body = JSONObject()
        body.put("username", username)
        body.put("password", password)

        val registerRequest = JsonRequest(
            Request.Method.POST,
            REGISTER_ENDPOINT,
            body,
        )
    }
}
```

```

        {
            Model.saveToken(it.getString("token"))

            val intent = Intent(this, NavActivity::class.java)
            startActivity(intent)
            finish()
        },
        {
            Toast.makeText(this, "An error occurred during registration!",
                Toast.LENGTH_SHORT).show()
        }
    )

    Connector.getInstance(this).addToRequestQueue(registerRequest)
}

```

4.6.2.3 NavActivity :

It is used to show the menu and the navigation bar is used to switch to different functions of the application like browse products, save product, view product details and view alerts on price change notifications. It also implements the logout feature.

```

val drawerLayout: DrawerLayout = binding.drawerLayout
val navView: NavigationView = binding.navView
val navController = findNavController(R.id.nav_host_fragment_content_main)

appBarConfiguration = AppBarConfiguration(
    setOf(R.id.nav_alerts, R.id.nav_browse, R.id.nav_saved),
    drawerLayout
)

setupActionBarWithNavController(navController, appBarConfiguration)
navView.setupWithNavController(navController)

if (!isServiceRunning(AlarmSchedulerService::class.java)) {
    startService(Intent(this, AlarmSchedulerService::class.java))
}
}

override fun onCreateOptionsMenu(menu: Menu): Boolean {
    menuInflater.inflate(R.menu.main, menu)
    return true;
}

```

4.6.2.4 SavedDetailsActivity

This is used to show the trends of prices to the users of their respective ‘liked’ product. Every time, the price of the product changes, the time of its change is stored and the new price as well.

```

val checkIntent = Intent(this, FetcherService::class.java).apply {
    putExtra("path", Utils.HISTORY_ENDPOINT)
    putExtra("method", Request.Method.POST)
}

```

```

        putExtra("data", JsonObject(mapOf(Pair("title", title))).toJsonString())
        putExtra("receiver", FetcherReceiver(Handler(), object : ResponseHandler {
            override fun handleResult(resultCode: Int, resultData: Bundle?) {
                if (resultCode == 200) {
                    val parser: Parser = Parser.default()
                    val stringBuilder: StringBuilder =
StringBuilder(resultData!!.getString("response")!!)
                    val responseObject: JsonObject = parser.parse(stringBuilder) as
JsonObject
                    val productsArrayStr =
responseObject.lookup<JSONArray>("history").toJsonString()
                    val products = parsePriceHistory(productsArrayStr).map { it.toGraphDTO()
                }

                    val chartView = findViewById<AnyChartView>(R.id.detailsChart)
                    chartView.setProgressbar(findViewById(R.id.detailsChartProgressbar))

                    priceDiagram = PriceDiagram(chartView, products)
                    priceDiagram.init()
                } else {
                    Toast.makeText(context, "Could not get saved product details!",
Toast.LENGTH_SHORT)
                        .show()
                }
            }
        })
    }
}

```

4.6.2.5 SplashActivity

This activity is used to show the splash screen.

```

class SplashActivity : AppCompatActivity() {
    private var handler: Handler? = null

    private val splashPassRunnable = Runnable {
        if (!isFinishing) {
            val intent = Intent(applicationContext, LoginActivity::class.java)
            startActivity(intent)
            finish()
        }
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        window.setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
WindowManager.LayoutParams.FLAG_FULLSCREEN)
        setContentView(R.layout.activity_splash)
        Model.loadToken()
    }

    override fun onResume() {
        super.onResume()

        val checkIntent = Intent(this, FetcherService::class.java).apply {
            putExtra("path", CHECK_ENDPOINT)
            putExtra("receiver", FetcherReceiver(Handler(), object : ResponseHandler
{
                override fun handleResult(resultCode: Int, resultData: Bundle?) {
                    Log.d(DEBUG_TAG, "SplashActivity: Service result received in
SplashActivity")

                    handler = Handler()
                }
            })
        })
    }
}

```



```

val requestBody = JSONObject()
requestBody.put("id", Model.changesList[position].ident)

val jsonObjectRequest = JsonRequest(
    Request.Method.POST,
    CHANGES_READ_ENDPOINT,
    requestBody,
    {
        Model.changesList.removeAt(position)
        Model.saveChanges()
        notifyItemRemoved(position)
        notifyItemChanged(Model.changesList.size - 1)
        updateTotalPrices()
    },
    {
        Toast.makeText(context, "Cannot hide alarm, check your Internet
connection", Toast.LENGTH_SHORT).show()
    }
)

```

The ‘browse’ recycler view has an important method called ‘saveProduct()’ which uses the calls the ‘FetcherService’ which will be discussed later. The ‘PriceDiagram’ class uses the AnychartView to give the description, price and time of the price change of the product an intercative graphical diagram. The code of the ‘PriceDiagram’ can be seen below:

```

class PriceDiagram(
    private val chartView: AnyChartView,
    private val products: List<PriceGraphDTO>
) {
    fun init() {
        val cartesian: Cartesian = AnyChart.line()

        cartesian.animation(true)

        cartesian.padding(10.0, 20.0, 5.0, 20.0)

        cartesian.crosshair().enabled(true)
        cartesian.crosshair().yLabel(true)
        .yStroke(null as Stroke?, null, null, null as String?, null as String?)

        cartesian.tooltip().positionMode(TooltipPositionMode.POINT)

        cartesian.title("Price trend graph")

        cartesian.yAxis(0).title("Price in USD")
        cartesian.xAxis(0).title("Day")
        cartesian.xAxis(0).labels().padding(1.0, 1.0, 1.0, 1.0)
        cartesian.xAxis(0).labels().adjustFontSize(true)

        val set: Set = Set.instantiate()!!
        set.data(products)

        val series1: Line = cartesian.line(set.mapAs("{ x: 'date', value: 'price' }"))
        series1.name("Price in USD")
        series1.hovered().markers().enabled(true)
        series1.hovered().markers().type(MarkerType.CIRCLE).size(4.0)
    }
}

```

```

series1.tooltip().position("right").anchor(Anchor.LEFT_CENTER).offsetX(5.0).offsetY(5.0)

    cartesian.legend().enabled(true)
    cartesian.legend().fontSize(13.0)
    cartesian.legend().padding(0.0, 0.0, 10.0, 0.0)

    chartView.setChart(cartesian)
}
}

```

4.7 Utils

The Utils has all the endpoints. The code below depicts it :

```

object Utils {
    private const val BASE_URL = "http://10.0.2.2:5000/api"
    const val CHECK_ENDPOINT = "$BASE_URL/connection/check"
    const val CHANGES_ENDPOINT = "$BASE_URL/prices/changes"
    const val PRODUCTS_ENDPOINT = "$BASE_URL/products/get"
    const val SAVED_PRODUCTS_ENDPOINT = "$BASE_URL/products/saved/get"
    const val HISTORY_ENDPOINT = "$BASE_URL/products/saved/history"
    const val SAVE_ENDPOINT = "$BASE_URL/products/save"
    const val CHANGES_READ_ENDPOINT = "$BASE_URL/prices/seen"
    const val LOGIN_ENDPOINT = "$BASE_URL/auth/login"
    const val LOGOUT_ENDPOINT = "$BASE_URL/auth/logout"
    const val REGISTER_ENDPOINT = "$BASE_URL/auth/register"

    const val DEFAULT_SYNC_INTERVAL = 10 * 1000.toLong()

    const val ACTION_CHECK = "CHECK"
    const val ACTION_FETCH = "FETCH"
    const val DEBUG_TAG = "PRICE_TRACKER_DEBUG"

    fun parseProducts(responseData: String): ArrayList<Product> {
        val klaxon = Klaxon()
        val products = arrayListOf<Product>()

        JsonReader(StringReader(responseData)).use { reader ->
            reader.beginArray {
                while (reader.hasNext()) {
                    val product = klaxon.parse<Product>(reader)!!
                    products.add(product)
                }
            }
        }

        return products
    }
}

```

As we can see above, there are endpoints which can be basically termed as communication channels for the API. It has the base url which is the consistent part of the URL and all the actions that will be taken after that will lead to endpoints that will extend this URL. For example, in the endpoint, the 10.0.2.2 is used rather than the computer's IP address because each instance of the emulator runs

behind a virtual router/firewall service that will move it away from the machine's network interfaces and settings and from the internet [13]. An emulator is unable to see our local development machine or other emulator instances on the network. Hence, the 10.0.2.2 ip address gives a special alias to the host loopback interface. After that is the port number (i.e. 5000 in this case) which is like a door in the computer which listens to requests and then as we move to different methods we have different endpoints but the same consistent Base URL.

4.8 Model

Model is an object that contains the lists of all products, saved products and products that need to be displayed in the notification recycler view. All the services, recycler views and alarms use these mutable arraylist to store the data. The code can be seen below :

```
object Model {
    var savedList: MutableList<Product> = arrayListOf()
    var productList: MutableList<Product> = arrayListOf()
    var changesList : MutableList<Price> = arrayListOf()
    var token: Token = Token(null)

    fun saveToken(tokenString: String) {
        token = Token(tokenString)
        token.save()
    }

    fun clearAll() {
        changesList.forEach { it.delete() }
        changesList = Collections.emptyList()
        token.delete()
        token = Token(null)
    }

    fun saveChanges() {
        changesList.forEach { it.save() }
    }

    fun loadToken() {
        token = try {
            val loadedToken : Token? = Select.from(Token::class.java).firstOrNull()
as Token?
            loadedToken ?: run { Token(null) }
        } catch (_: SQLiteException) {
            Token(null)
        }
    }

    fun saveProducts() {
        productList.forEach { it.save() }
    }
}
```

4.9 Running the Application

It is important to run the Spring Boot backend before running the android application because if its not running, then the application will throw an exception for not having the internet which is the server in our case as shown in Figure 5

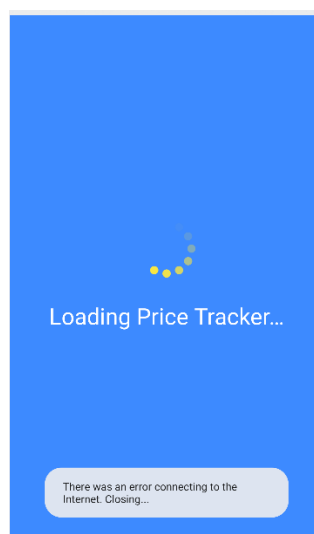


Figure 5 Running the app without the backend

The application starts with the splash screen shown in the image Figure 6

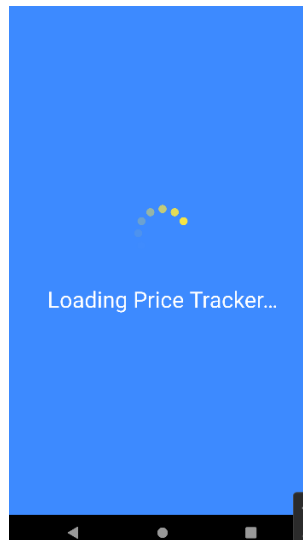


Figure 6 Splash Screen

Then, we move onto the login activity which is the login page shown in Figure 7 :

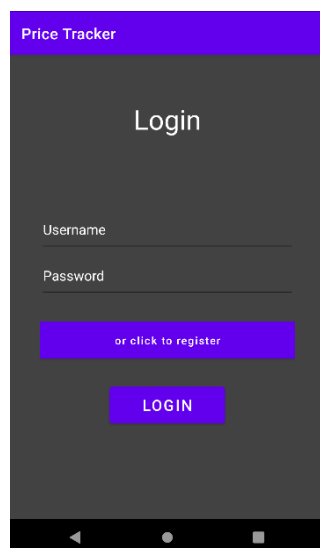
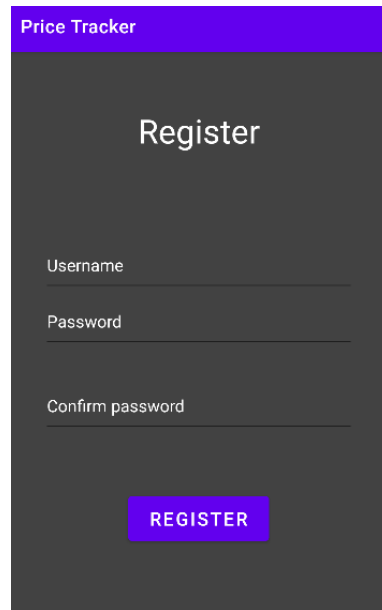


Figure 7 Login screen

If the user already has an account, then they will login otherwise they will click on the register button which leads them to the register activity shown in Figure 8.

A screenshot of a mobile application's 'Register' page. The page has a dark gray background. At the top, there is a purple header bar with the text 'Price Tracker' in white. Below the header, the word 'Register' is centered in a large, white, sans-serif font. Underneath, there are three input fields with white text labels: 'Username', 'Password', and 'Confirm password'. Each label is positioned to the left of a horizontal white line representing the input field. At the bottom of the form, there is a purple rectangular button with the word 'REGISTER' in white, uppercase letters.

Price Tracker

Register

Username

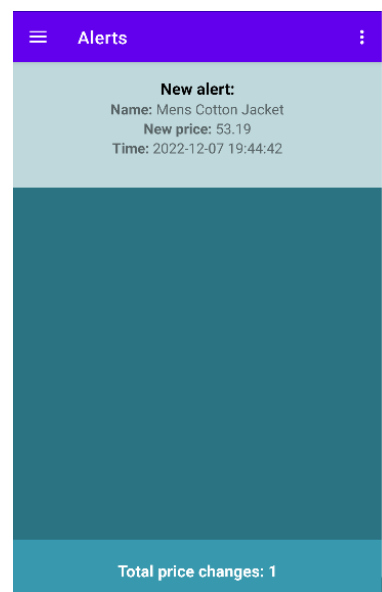
Password

Confirm password

REGISTER

Figure 8 Register Page

After registration or login, the user will be directed to the nav activity which will show him the menu as shown below in Figure 9:

A screenshot of a mobile application's 'Nav Activity' screen. The screen has a dark teal background. At the top, there is a purple header bar with a white hamburger menu icon on the left, the word 'Alerts' in white in the center, and a white vertical ellipsis icon on the right. Below the header, there is a light blue rectangular box containing the text 'New alert:' followed by three lines of smaller text: 'Name: Mens Cotton Jacket', 'New price: 53.19', and 'Time: 2022-12-07 19:44:42'. At the bottom of the screen, there is a teal rectangular bar with the text 'Total price changes: 1' in white.

Alerts

New alert:
Name: Mens Cotton Jacket
New price: 53.19
Time: 2022-12-07 19:44:42

Total price changes: 1

Figure 9 Nav Activity

Clicking on the menu shows us the functions of the application as shown in Figure 10 :

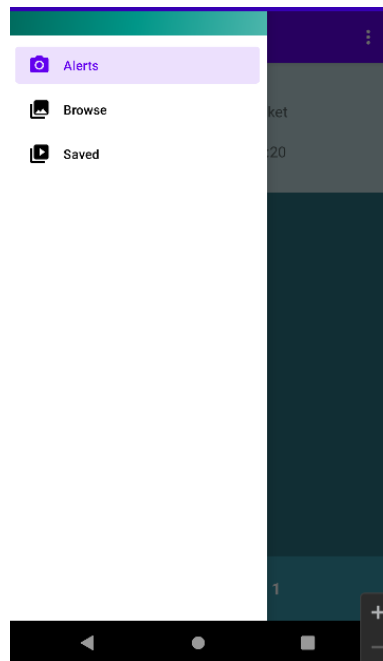


Figure 10 Menu

Clicking on the 'Heart' icon shown in Figure 11 has the intent to save the product and before doing so, it asks for the threshold price which can be seen in Figure 12.

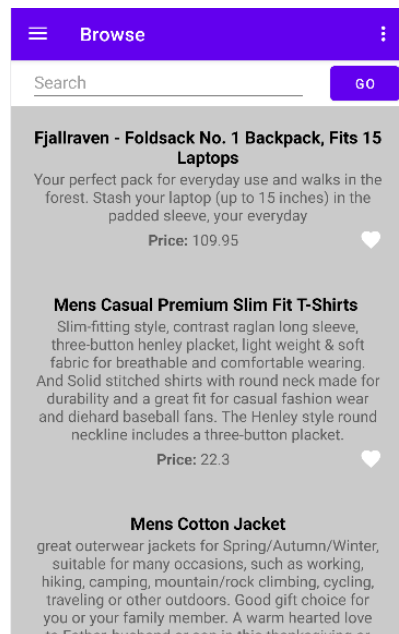


Figure 11 Browse and search

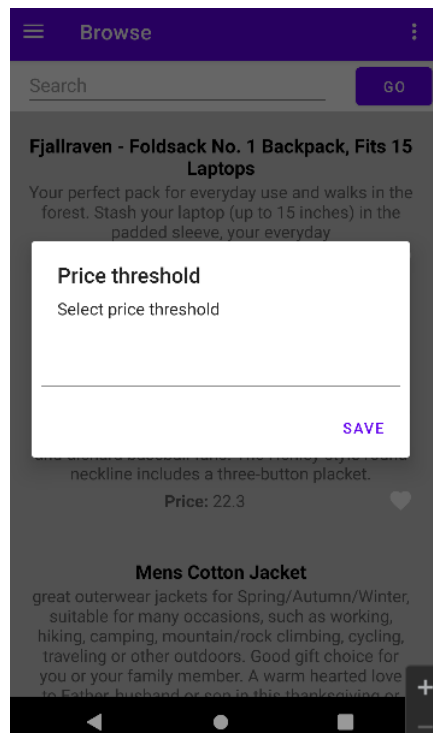


Figure 12 Price threshold dialog boks

When the user goes to the menu, he can select saved icon and see the list of saved products as shown in Figure 13.

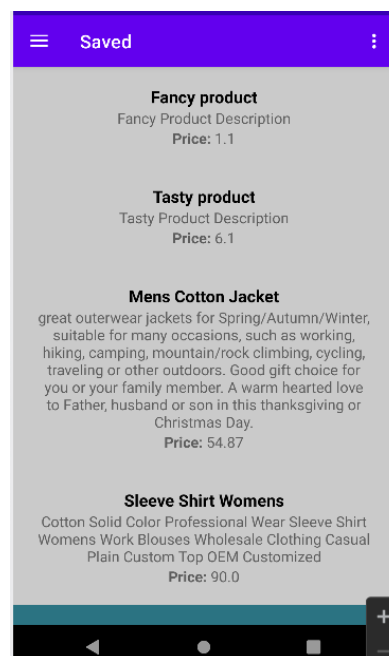


Figure 13 List of saved products

Clicking on the saved product opens the product’s description as shown in Figure 14.



Figure 14 Price trend Diagram

5 Conclusion

The project's goal was to create an application which is aimed at providing the users with notifications about the change in prices of their desired products if the price of the product goes below a certain threshold price then it shows them the trend of the price change. Users can register and login to the application, they can search for products in the API-s, filter and mark existing products searched for by any user as favorites and check the changing trends of prices on diagrams.

The project creates its own backend using Spring boot to write APIs. The interest in doing this project was ignited by the thought of contruscting the whole backend manually rather than using Firebase to understand the technology better.

In the future, this application can be extended to support actual existing websites and adding new features such as support for offline mode (by caching data from the server). To conclude, it can be said that the aim of the project was to learn more about new techniques and platforms and that has been achieved by also creating a sustainable application for the users at the same time.

6 References

- [1] Amazon. (2018, september 01). *Amazon Gateway APIs*. Retrieved from Aws: <https://aws.amazon.com/api-gateway/>
- [2] Baeldung. (2019, 01 02). *Baeldung-SpringBoot*. Retrieved from Baeldung: <http://www.baeldung.com>
- [3] Collings, T. (2021, August 20). *Medium-Controller-service-Repository*. Retrieved from Medium: <https://tom-collings.medium.com/controller-service-repository-16e29a4684e5>
- [4] EBay. (2022, jan 06). *Ebay Developers Program*. Retrieved from Ebay: <https://developer.ebay.com/develop/apis>
- [5] GeeksForGeeks. (2017, Jan 3). *GeeksForGeeks-DTO*. Retrieved from GeeksForGeeks: <https://www.geeksforgeeks.org/data-transfer-object-dto-in-spring-mvc-with-example/#:~:text=In%20Spring%20Framework%2C%20Data%20Transfer,the%20data%20for%20the%20call.>
- [6] Keikavousi, M. (2022, October 27). *Fake Store Api*. Retrieved from Fake Store api: <https://fakestoreapi.com/>
- [7] Kumar, A. (2022, 11 1). *TutorialsPoint - Activity Life =Cycle*. Retrieved from Tutorials Point: [\[www.tutorialspoint.com\]](http://www.tutorialspoint.com)
- [8] Mahse, S. (2022). *REST Tutorial*. Retrieved from REST: <https://restfulapi.net/>
- [9] Smith, J. (2018, 07 22). *Baeldung-OnKotlin*. Retrieved from Baeldung: <http://baeldung.com>
- [10] Tanzu, V. (2022). *Spring Boot*. Retrieved from spring.io: <https://spring.io/projects/spring-boot>
- [11] VMware. (2019, September 30). *Spring.io-db*. Retrieved from Spring.io: <https://spring.io/guides/gs/accessing-data-mysql/>
- [12] W3Schools. (2021). *W3Schools-mysql*. Retrieved from W3Schools-mysql-tutorial: <https://www.w3schools.com/MySQL/default.asp>
- [13] StackOverflow. (2020). *StackOverflow*: Retrived from <https://stackoverflow.com/questions/9808560/why-do-we-use-10-0-2-2-to-connect-to-local-web-server-instead-of-using-computer>

