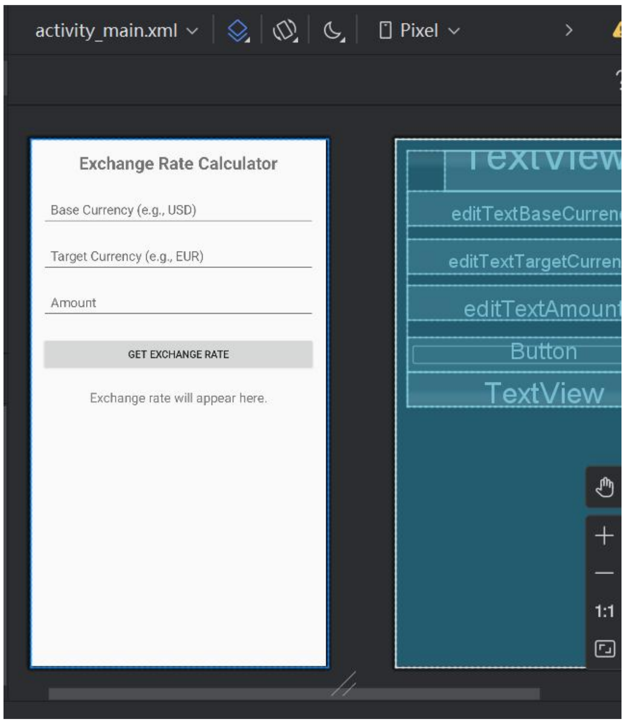
The ExchangeRateApp is an application that allow users to get the exchange rates up to date.



**1. Implement a native Android application**

**a. Has at least three different kinds of Views in your Layout (TextView, EditText, ImageView, or anything that extends android.view.View). In order to figure out if something is a View, find its API. If it extends android.view.View then it is a View.**

In activity_main.xml, there are TextView, EditText, Button, and ProgressBar.

**b. Requires input from the user**

In activity_main.xml, there is EditText that user can input base currency, target currency and amount.

**c. Makes an HTTP request (using an appropriate HTTP method) to your web service**

ExchangeRateService.kt:

```
interface ExchangeRateService {
    @GET("/getExchangeRate")
    fun getExchangeRate(
        @Query("baseCurrency") baseCurrency: String,
        @Query("targetCurrency") targetCurrency: String
    ): Call<ExchangeRateResponse>
}
```

MainActivity.kt's HTTP request:

```
private fun fetchExchangeRate() {
    val baseCurrency = editTextBaseCurrency.text.toString().trim().toUpperCase()
    val targetCurrency = editTextTargetCurrency.text.toString().trim().toUpperCase()
    val amountStr = editTextAmount.text.toString().trim()
    if (baseCurrency.isEmpty() || targetCurrency.isEmpty() || amountStr.isEmpty()) {
        Toast.makeText(this, "Please enter base currency, target currency, and amount.", Toast.LENGTH_SHORT).show()
        return
    }

    val amount: Double = try {
        amountStr.toDouble()
    } catch (e: NumberFormatException) {
        Toast.makeText(this, "Please enter a valid amount.", Toast.LENGTH_SHORT).show()
        return
    }

    progressBar.visibility = View.VISIBLE
    textViewResult.text = ""

    val call = exchangeRateService.getExchangeRate(baseCurrency, targetCurrency)
    call.enqueue(object : Callback<ExchangeRateResponse> {
        override fun onResponse(call: Call<ExchangeRateResponse>, response: Response<ExchangeRateResponse>) {
            progressBar.visibility = View.GONE

            if (!response.isSuccessful) {
                textViewResult.text = "Error: ${response.code()}"
                return
            }

            val exchangeRate = response.body()
            if (exchangeRate != null) {
                if (exchangeRate.result.equals("success", ignoreCase = true)) {
                    val convertedAmount = amount * exchangeRate.rate
                    val displayText = "1 ${exchangeRate.baseCurrency} = ${exchangeRate.rate} ${exchangeRate.targetCurrency}\n" +
                            "$amount ${exchangeRate.baseCurrency} = ${"%.2f".format(convertedAmount)} ${exchangeRate.targetCurrency}"
                    textViewResult.text = displayText
                } else {
                    textViewResult.text = "Error: ${exchangeRate.error}"
                }
            } else {
                textViewResult.text = "Error: Response body is null."
            }
        }
```

```
            override fun onFailure(call: Call<ExchangeRateResponse>, t: Throwable) {
                progressBar.visibility = View.GONE

                textViewResult.text = "Error: ${t.message}"
            }
        })
    }
```

**d. Receives and parses an XML or JSON formatted reply from your web service**

In ExchangeRateResponse.kt and MainActivity.kt:

Using the Gson library, the application can automatically parse the JSON format response into the ExchangeRateResponse data class, meeting the requirements of receiving and parsing JSON format replies.

**e. Displays new information to the user**

By updating the text content of textViewResult, the application can display the query results or error information to the user.

**f. Is repeatable (I.e. the user can repeatedly reuse the application without restarting it.)**

In MainActivity.kt there is:

```
buttonGetRate.setOnClickListener {
    fetchExchangeRate()
}
private fun fetchExchangeRate() {
    //...
}
```

Each time the user clicks the "Get Exchange Rate" button, the fetchExchangeRate() method is called to perform a new query without restarting the application, thus achieving reusable functionality.

**2. Implement a web service**

**a. Implement a simple (can be a single path) API.**

ExchangeRateServlet.java:

```
@WebServlet("/getExchangeRate")
public class ExchangeRateServlet extends HttpServlet {
    // ...
}
```

Web.xml:

```
<servlet>
    <servlet-name>ExchangeRateServlet</servlet-name>
    <servlet-class>ds.project4.ExchangeRateServlet</servlet-class>
</servlet>
```

```
<servlet-mapping>
        <servlet-name>ExchangeRateServlet</servlet-name>
        <url-pattern>/getExchangeRate</url-pattern>
</servlet-mapping>
```

A simple, single-path RESTful API is implemented through ExchangeRateServlet and related web.xml configuration.

**b. Receives an HTTP request from the native Android application**

ExchangeRateServlet.java -- doGet:

```
@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        String baseCurrency = request.getParameter("baseCurrency");
        String targetCurrency = request.getParameter("targetCurrency");

        if (baseCurrency == null || targetCurrency == null) {
                response.setStatus(HttpServletResponse.SC_BAD_REQUEST);
                return;
        }
        // ...
}
```

**c. Executes business logic appropriate to your application. This includes fetching XML or JSON information from some 3rd party API and processing the response.**

ExchangeRateServlet.java:

```
// Build the request URL of ExchangeRate-API
String apiRequestUrl = API_URL + baseCurrency.toUpperCase();

// HTTP request:
Request apiRequest = new Request.Builder()
                .url(apiRequestUrl)
                .build();

try (Response apiResponse = httpClient.newCall(apiRequest).execute()) {
        if (!apiResponse.isSuccessful()) {
                return;
        }

        // Parse the JSON response of ExchangeRate-API
        String responseBody = apiResponse.body().string();
        JsonObject apiData = JsonParser.parseString(responseBody).getAsJsonObject();

        if (!"success".equalsIgnoreCase(apiData.get("result").getAsString())) {
            return;
        }

        // Extract exchange rate information
        JsonObject conversionRates = apiData.getAsJsonObject("conversion_rates");
        if (!conversionRates.has(targetCurrency.toUpperCase())) {
            return;
        }

        double rate = conversionRates.get(targetCurrency.toUpperCase()).getAsDouble();

        // Create response JSON
        JsonObject exchangeRateResponse = new JsonObject();
        exchangeRateResponse.addProperty("result", "success");
        exchangeRateResponse.addProperty("baseCurrency", baseCurrency.toUpperCase());
        exchangeRateResponse.addProperty("targetCurrency", targetCurrency.toUpperCase());
        exchangeRateResponse.addProperty("rate", rate);

        //Send response to Android application
        PrintWriter out = response.getWriter();
        out.print(gson.toJson(exchangeRateResponse));
        out.flush();

    } catch (Exception e) {
        e.printStackTrace();
    }
```

**d. Replies to the Android application with an XML or JSON formatted response. The schema of the response can be of your own design.**

In ExchangeRateServlet.java, set the Content-Type of the response to application/json to ensure that the client parses it in JSON format.

A JSON object containing the result, baseCurrency, targetCurrency, and rate fields is constructed and returned to the Android application as a response.

**3. Handle error conditions**

**Your application should test for and handle gracefully:**

**Invalid mobile app input**

**Invalid server-side input (regardless of mobile app input validation)**

**Mobile app network failure, unable to reach server**

**Third-party API unavailable**

**Third-party API invalid data**

**There are several checking for input in ExchangeRateServlet.java:**

```
String baseCurrency = request.getParameter("baseCurrency");
String targetCurrency = request.getParameter("targetCurrency");

if (baseCurrency == null || targetCurrency == null) {
        response.setStatus(HttpServletResponse.SC_BAD_REQUEST);
        PrintWriter out = response.getWriter();
        JsonObject errorResponse = new JsonObject();
        errorResponse.addProperty("error", "Missing parameters. Please provide baseCurrency and
targetCurrency.");
        out.print(gson.toJson(errorResponse));
        out.flush();
        return;
}
if (!conversionRates.has(targetCurrency.toUpperCase())) {
        response.setStatus(HttpServletResponse.SC_BAD_REQUEST);
        PrintWriter out = response.getWriter();
        JsonObject errorResponse = new JsonObject();
        errorResponse.addProperty("error", "Invalid targetCurrency provided.");
        out.print(gson.toJson(errorResponse));
        out.flush();
        return;
}
```

**In MainActivity.kt:**

```
override fun onFailure(call: Call<ExchangeRateResponse>, t: Throwable) {
        progressBar.visibility = View.GONE

        textViewResult.text = "Error: ${t.message}"
}
```

When a network request fails (e.g., cannot connect to the server), the onFailure callback is triggered.

Hide the ProgressBar and display an error message in the textViewResult to inform the user that a network failure has occurred.

**In ExchangeRateServlet.java:**

```
try (Response apiResponse = httpClient.newCall(apiRequest).execute()) {
        if (!apiResponse.isSuccessful()) {
                response.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
                PrintWriter out = response.getWriter();
                JsonObject errorResponse = new JsonObject();
                errorResponse.addProperty("error", "Failed to fetch data from ExchangeRate-API.");
                out.print(gson.toJson(errorResponse));
                out.flush();
```

```
            return;
        }
        // ...
} catch (Exception e) {
        e.printStackTrace();
        response.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
        PrintWriter out = response.getWriter();
        JsonObject errorResponse = new JsonObject();
        errorResponse.addProperty("error", "An error occurred while processing the request.");
        out.print(gson.toJson(errorResponse));
        out.flush();
}
```

If the third-party API returns a non-successful HTTP status code (such as 500), return the HTTP 500 (Internal Server Error) status code and send an error message.
Catch any exceptions (such as network problems, API unavailable, etc.) in the try-catch block and return the HTTP 500 status code and the corresponding error message.

ExchangeRateServlet.java also checks if the result field is success. If not, the third-party API returned an error result.
Check if conversion_rates contains the target currency. If not, the third-party API returned invalid data.
In both cases, the server will return the corresponding error response.

**4. Log useful information**
**5. Store the log information in a database**
**6. Display operations analytics and full logs on a web-based dashboard**
These requirements are just partially met yet.

## 7. Deploy the web service to GitHub Codespaces

**a. Accept the Github Classroom Assignment that you have been given the URL for. You will find a repository with:**

- **A .devcontainer.json and a Dockerfile which define how to create a Docker container, build a suitable software stack, and deploy the ROOT.war web application.**
- **A ROOT.war file which, like in Lab 3, contains a web application that will deployed in the container. This is a simple "Hello World!" application.**
- **An identical copy of this README.md**

**b. Click the green <> Code dropdown button, select the Codespaces tab, then click on "Create codespace on master".**

**c. Once the Codespace is running, the Terminal tab will show that Catalina (the Servlet container) is running. You should also see a "1" next to the Ports tab. Click on the Ports tab and you should see that port 8080 has been made available.**

**d. Mouse over the Local address item of the port 8080 line and you will find three icons. The leftmost is to copy the URL of your deployed application, the middle one (a globe) is to launch that URL in a browser. Clicking on the globe is a quick way to test your web service in a browswer. The copy is useful to use the URL in your Android App.**

**e. Click on the globe to confirm that the Hello World servlet is working.**

**f. By default, the URL in (d) requires you to be authenticated with Github. To test in a browser, that is fine, but when accessing your web service from your Android app, the Android app will not be authenticated. Therefore you must make the port visibility "Public". To do this, right or control click on the word "Private" in the Visibility column, and change Port Visibility to "Public". You will now be able to access the web service from your Android App or from an unauthenticated browser.**

**g. Copy the URL and paste into an Incognito Chrome window to confirm that the Hello World web app can be reached without authentication.**

**h. To deploy your own web service, create a ROOT.war like you did in Lab 3, upload or push the ROOT.war to your repository, and create a Codespace as has just been described.**

Done in codespace.