One Authentication/Authorization Framework

18-08-2022

Spring Security /OAUTH2 – Approach

Implementation Guide

Demo Application

INSIDE

—

Spring Security /OAuth2

Describes Spring Security construct and OAUTH2.0 Standards

—

Social Login /Email Authentication

Social Login and email ID based Authorization enablement.

Detailed implementation on Token-JWT Provision

—

Front End

Sample Front-End Application to show the Authorization Capabilities

Spring Boot OAuth2 Social Login with Google, Facebook, and Github etc as well as email and password-based login using Spring Security.

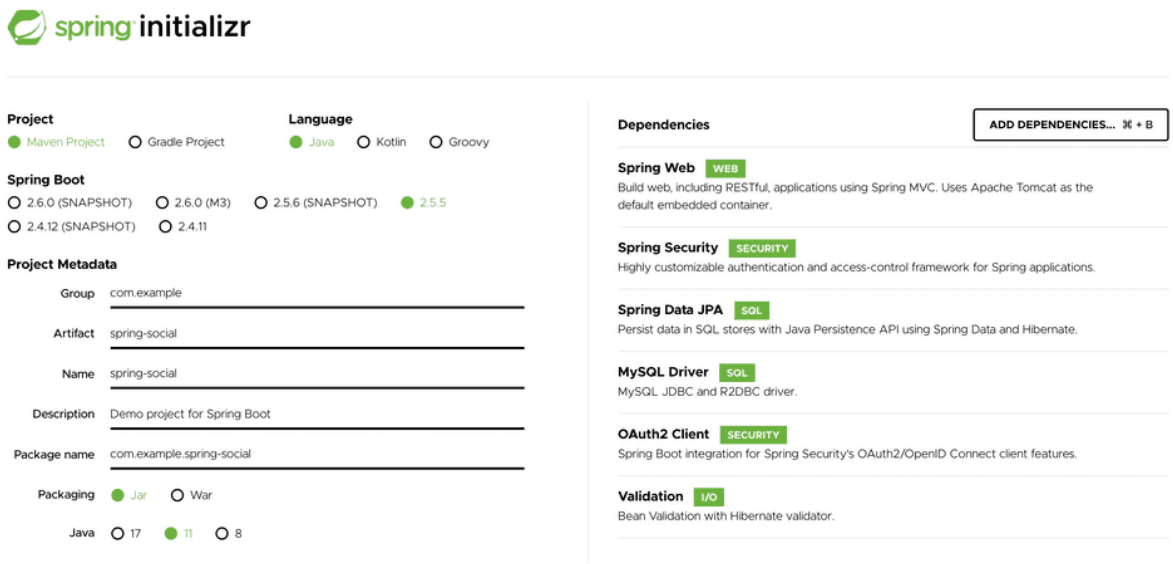
Add social as well as email and password-based login to your spring boot application using the new OAuth2 functionalities provided in Spring Security.

Creating the Project

Let's create our project using Spring Initializr web tool. Head over to http://start.spring.io, fill in the details as follows:

- **Artifact**: spring-social
- **Dependencies**: Spring Web, Spring Security, OAuth2 Client, Spring Data JPA, MySQL Driver, Validation

You can leave the rest of the fields to their default values and click **Generate** to generate and download the project -



Directory structure of the complete project for reference

Following is the directory structure of the complete project for your reference. We'll create all the classes and interfaces one by one and understand their details:


spring-social ~/spring-boot-react-oauth2-social-login-demo/spring-social
src
main
java
com.example.springsocial
config
AppProperties
SecurityConfig
WebMvcConfig
controller
AuthController
UserController
exception
BadRequestException
OAuth2AuthenticationProcessingException
ResourceNotFoundException
model
AuthProvider
User
payload
ApiResponse
AuthResponse
LoginRequest
SignUpRequest
repository
UserRepository
security
oauth2
user
FacebookOAuth2UserInfo
GithubOAuth2UserInfo
GoogleOAuth2UserInfo
OAuth2UserInfo
OAuth2UserInfoFactory
CustomOAuth2UserService
HttpCookieOAuth2AuthorizationRequestRepository
OAuth2AuthenticationFailureHandler
Oauth2AuthenticationSuccessHandler
CurrentUser
CustomUserDetailsService
RestAuthenticationEntryPoint
TokenAuthenticationFilter
TokenProvider
UserPrincipal
util
CookieUtils
SpringSocialApplication
resources
application.yml
test

Additional dependencies

We'll need to add few additional dependencies to our application that are not present in spring initializr web tool. Open the `pom.xml` file located in the root directory of the project and add the following dependencies -

```
<!-- JWT library -->
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.2</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.2</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.11.2</version>
    <scope>runtime</scope>
</dependency>
```

User email & password Storage

- The email and password will be stored in MySQL database relevant to user's information.
- Refer Database design document for detailed information.

Creating OAuth2 apps for social login

To enable social login with an OAuth2 provider, you'll need to create an app in the OAuth2 provider's console and get the ClientId and ClientSecret, sometimes also called an AppId and AppSecret.

OAuth2 providers use the ClientId and ClientSecret to identify your app. The providers also ask for many other settings that include -

- **Authorized redirect URIs**: These are the valid list of redirect URIs where a user can be redirected after they grant/reject permission to your app. This should point to your app endpoint that will handle the redirect.
- **Scope**: Scopes are used to ask users for permission to access their data.

Creating Facebook, Github, and Google Apps

- **Facebook App**: You can create a facebook app from the Facebook apps dashboard
  (Refer Appendix for procedure for adding Oauth2 for Facebook)

- **Github App**: Github apps can be created from https://github.com/settings/apps.
- **Google Project**: Head over to Google Developer Console to create a Google Project and the credentials for OAuth2.

  (Refer Appendix for Costing for using google identify platform)

Sample demo app for Facebook, Google, and Github has been created for reference. We'll use the demo apps to perform social login.

Configuring the Spring Boot application

Spring boot reads configurations from `src/main/resource/application.properties` file by default. It also supports `.yaml` configurations. In this project, we'll use yaml configurations because they represent hierarchical data more clearly.

Rename `application.properties` file to `application.yaml` and add the following configurations -

```yaml
spring:

    datasource:

        url:
jdbc:mysql://localhost:3306/spring_social?useSSL=false&serverTimezon
e=UTC&useLegacyDatetimeCode=false

        username: root

        password: root



    jpa:

        show-sql: true

        hibernate:

            ddl-auto: update

            naming-strategy:
org.hibernate.cfg.ImprovedNamingStrategy

        properties:

            hibernate:

                dialect: org.hibernate.dialect.MySQL5InnoDBDialect

    security:

      oauth2:

        client:

          registration:

            google:

              clientId: 5014057553-
8gm9um6vnli3cle5rgigcdjpdrid14m9.apps.googleusercontent.com

              clientSecret: tWZKVLxaD_ARWsriiiUFYoIk

              redirectUri:
"{baseUrl}/oauth2/callback/{registrationId}"

              scope:

                - email

                - profile
```

```
            facebook:

              clientId: 121189305185277

              clientSecret: 42ffe5aa7379e8326387e0fe16f34132

              redirectUri:
"{baseUrl}/oauth2/callback/{registrationId}" # Note that facebook
now mandates the use of https redirect URIs, so make sure your app
supports https in production

              scope:

                - email

                - public_profile

            github:

              clientId: d3e47fc2ddd966fa4352

              clientSecret: 3bc0f6b8332f93076354c2a5bada2f5a05aea60d

              redirectUri:
"{baseUrl}/oauth2/callback/{registrationId}"

              scope:

                - user:email

                - read:user

          provider:

            facebook:

              authorizationUri:
https://www.facebook.com/v3.0/dialog/oauth

              tokenUri:
https://graph.facebook.com/v3.0/oauth/access_token

              userInfoUri:
https://graph.facebook.com/v3.0/me?fields=id,first_name,middle_name,
last_name,name,email,verified,is_verified,picture.width(250).height(
250)

app:

  auth:

    tokenSecret:
04ca023b39512e46d0c2cf4b48d5aac61d34302994c87ed4eff225dcf3b0a218739f
3897051a057f9b846a69ea2927a587044164b7bae5e1306219d50b588cb1

    tokenExpirationMsec: 864000000
```

```
  cors:

    allowedOrigins: http://localhost:3000 # Comma separated list of
allowed origins

  oauth2:

    # After successfully authenticating with the OAuth2 Provider,

    # we'll be generating an auth token for the user and sending the
token to the

    # redirectUri mentioned by the client in the /oauth2/authorize
request.

    # We're not using cookies because they won't work well in mobile
clients.

    authorizedRedirectUris:

      - http://localhost:3000/oauth2/redirect

      - myandroidapp://oauth2/redirect

      - myiosapp://oauth2/redirect
```

The datasource configurations are used to connect to the MySQL database. Please create a database named `spring_social` and specify correct values for `spring.datasource.username` and `spring.datasource.password` as per your MySQL installation.

The `security.oauth2` configurations define all the oauth2 providers and their details. The `app.auth` configurations are used to generate a JWT authentication token once the user is successfully logged in.

Notice the use of `redirectUriTemplate` property in all the registered oauth2 providers. When you create an app in these OAuth2 providers websites, you must add an authorized redirect URI that matches this template.

For example, for your google app, you need to add the authorizedRedirectURI `http://localhost:8080/oauth2/callback/google`.

Binding AppProperties

Let's bind all the configurations prefixed with `app` to a POJO class using Spring Boot's `@ConfigurationProperties` feature-

```
package com.example.springsocial.config;
```

```java
import
org.springframework.boot.context.properties.ConfigurationProperties;


import java.util.ArrayList;

import java.util.List;


@ConfigurationProperties(prefix = "app")

public class AppProperties {

    private final Auth auth = new Auth();

    private final OAuth2 oauth2 = new OAuth2();


    public static class Auth {

        private String tokenSecret;

        private long tokenExpirationMsec;


        public String getTokenSecret() {

            return tokenSecret;

        }


        public void setTokenSecret(String tokenSecret) {

            this.tokenSecret = tokenSecret;

        }


        public long getTokenExpirationMsec() {

            return tokenExpirationMsec;

        }


        public void setTokenExpirationMsec(long tokenExpirationMsec)
{
```

```
            this.tokenExpirationMsec = tokenExpirationMsec;

        }

    }


    public static final class OAuth2 {

        private List<String> authorizedRedirectUris = new
ArrayList<>();


        public List<String> getAuthorizedRedirectUris() {

            return authorizedRedirectUris;

        }


        public OAuth2 authorizedRedirectUris(List<String>
authorizedRedirectUris) {

            this.authorizedRedirectUris = authorizedRedirectUris;

            return this;

        }

    }


    public Auth getAuth() {

        return auth;

    }


    public OAuth2 getOauth2() {

        return oauth2;

    }

}
```

Enabling AppProperties

We'll need to enable configuration properties by adding the `@EnableConfigurationProperties` annotation. Please open the main application class `SpringSocialApplication.java` and add the annotation like so-

```
package com.example.springsocial;


import com.example.springsocial.config.AppProperties;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import
org.springframework.boot.context.properties.EnableConfigurationProperties;


@SpringBootApplication

@EnableConfigurationProperties(AppProperties.class)

public class SpringSocialApplication {


	public static void main(String[] args) {

		SpringApplication.run(SpringSocialApplication.class,
args);

	}

}
```

Enabling CORS

Let's enable CORS so that our frontend client can access the APIs from a different origin. I've enabled the origin `http://localhost:3000` since that is where our frontend application will be running.

```
package com.example.springsocial.config;


import org.springframework.beans.factory.annotation.Value;
```

```
import org.springframework.context.annotation.Configuration;

import
org.springframework.web.servlet.config.annotation.CorsRegistry;

import
org.springframework.web.servlet.config.annotation.WebMvcConfigurer;



@Configuration

public class WebMvcConfig implements WebMvcConfigurer {


    private final long MAX_AGE_SECS = 3600;


    @Value("${app.cors.allowedOrigins}")

    private String[] allowedOrigins;


    @Override

    public void addCorsMappings(CorsRegistry registry) {

        registry.addMapping("/**")

        .allowedOrigins(allowedOrigins)

        .allowedMethods("GET", "POST", "PUT", "PATCH", "DELETE",
"OPTIONS")

        .allowedHeaders("*")

        .allowCredentials(true)

        .maxAge(MAX_AGE_SECS);

    }

}
```

Creating the database entities

Let's now create the Entity classes of our application. Following is the definition of the `User` class -

```java
package com.example.springsocial.model;

import com.fasterxml.jackson.annotation.JsonIgnore;


import javax.persistence.*;

import javax.validation.constraints.Email;

import javax.validation.constraints.NotNull;


@Entity

@Table(name = "users", uniqueConstraints = {

        @UniqueConstraint(columnNames = "email")

})

public class User {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;


    @Column(nullable = false)

    private String name;


    @Email

    @Column(nullable = false)

    private String email;


    private String imageUrl;


    @Column(nullable = false)

    private Boolean emailVerified = false;
```

```
    @JsonIgnore
    private String password;

    @NotNull
    @Enumerated(EnumType.STRING)
    private AuthProvider provider;

    private String providerId;

    // Getters and Setters (Omitted for brevity)
}
```

The `User` class contains information about the authentication provider. Following is the definition of the `AuthProvider` enum -

```
package com.example.springsocial.model;

public enum  AuthProvider {
    local,
    facebook,
    google,
    github
}
```

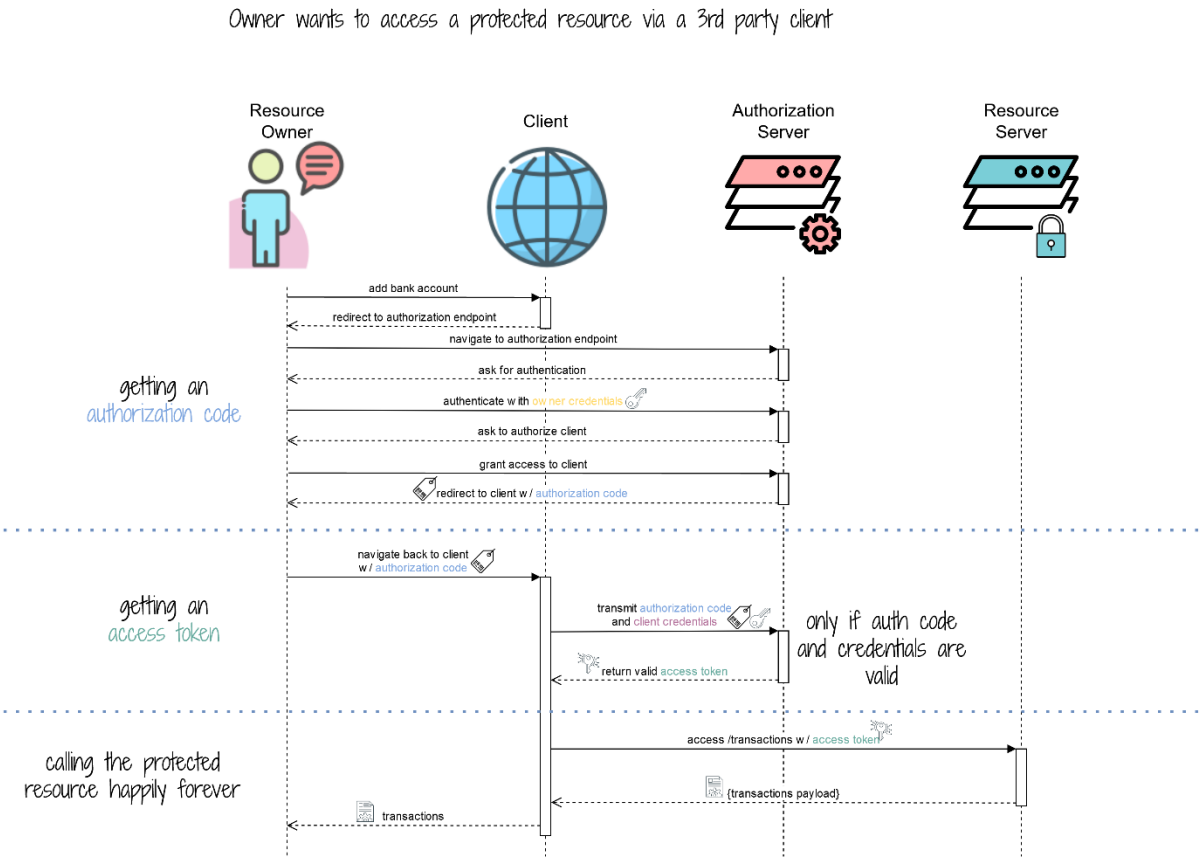
Creating the repositories for accessing data from the database

Repository layer for accessing data from the database. The following `UserRepository` interface provides database functionalities for the `User` entity. Spring-Data-JPA will be used.

```java
package com.example.springsocial.repository;

import com.example.springsocial.model.User;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.Optional;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    Optional<User> findByEmail(String email);

    Boolean existsByEmail(String email);

}
```

Spring Security OAuth2 - Social login & User implementation Approach

The process steps to implement the Spring Security OAUTH login flow has been explained below



Reference Diagram for app & OAUTH2 sequence diagram.

SecurityConfig

The following SecurityConfig class is the crux of our security implementation. It contains configurations for both OAuth2 social login as well as email and password based login.

Let’s first look at all the configurations and then we’ll dive into the details of each configuration one-by-one in this article.

```
package com.example.springsocial.config;


import com.example.springsocial.security.*;

import
com.example.springsocial.security.oauth2.CustomOAuth2UserService;

import
com.example.springsocial.security.oauth2.HttpCookieOAuth2Authorizati
onRequestRepository;

import
com.example.springsocial.security.oauth2.OAuth2AuthenticationFailure
Handler;

import
com.example.springsocial.security.oauth2.OAuth2AuthenticationSuccess
Handler;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import
org.springframework.security.authentication.AuthenticationManager;

import org.springframework.security.config.BeanIds;

import
org.springframework.security.config.annotation.authentication.builde
rs.AuthenticationManagerBuilder;

import
org.springframework.security.config.annotation.method.configuration.
EnableGlobalMethodSecurity;

import
org.springframework.security.config.annotation.web.builders.HttpSecu
rity;

import
org.springframework.security.config.annotation.web.configuration.Ena
bleWebSecurity;

import
org.springframework.security.config.annotation.web.configuration.Web
SecurityConfigurerAdapter;

import
org.springframework.security.config.http.SessionCreationPolicy;
```

```
import
org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

import org.springframework.security.crypto.password.PasswordEncoder;

import
org.springframework.security.oauth2.client.web.AuthorizationRequestR
epository;

import
org.springframework.security.oauth2.core.endpoint.OAuth2Authorizatio
nRequest;

import
org.springframework.security.web.authentication.UsernamePasswordAuth
enticationFilter;



@Configuration

@EnableWebSecurity

@EnableGlobalMethodSecurity(

        securedEnabled = true,

        jsr250Enabled = true,

        prePostEnabled = true

)

public class SecurityConfig extends WebSecurityConfigurerAdapter {


    @Autowired

    private CustomUserDetailsService customUserDetailsService;


    @Autowired

    private CustomOAuth2UserService customOAuth2UserService;


    @Autowired

    private OAuth2AuthenticationSuccessHandler
oAuth2AuthenticationSuccessHandler;
```

```
    @Autowired

    private OAuth2AuthenticationFailureHandler
oAuth2AuthenticationFailureHandler;



    @Autowired

    private HttpCookieOAuth2AuthorizationRequestRepository
httpCookieOAuth2AuthorizationRequestRepository;



    @Bean

    public TokenAuthenticationFilter tokenAuthenticationFilter() {

        return new TokenAuthenticationFilter();

    }



    /*

      By default, Spring OAuth2 uses
HttpSessionOAuth2AuthorizationRequestRepository to save

      the authorization request. But, since our service is
stateless, we can't save it in

      the session. We'll save the request in a Base64 encoded cookie
instead.

    */

    @Bean

    public HttpCookieOAuth2AuthorizationRequestRepository
cookieAuthorizationRequestRepository() {

        return new HttpCookieOAuth2AuthorizationRequestRepository();

    }



    @Override

    public void configure(AuthenticationManagerBuilder
authenticationManagerBuilder) throws Exception {

        authenticationManagerBuilder

                .userDetailsService(customUserDetailsService)
```

```
                .passwordEncoder(passwordEncoder());
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }



    @Bean(BeanIds.AUTHENTICATION_MANAGER)
    @Override
    public AuthenticationManager authenticationManagerBean() throws
Exception {
        return super.authenticationManagerBean();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
                .cors()
                    .and()
                .sessionManagement()
.sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                    .and()
                .csrf()
                    .disable()
                .formLogin()
                    .disable()
                .httpBasic()
```

```
                    .disable()
                .exceptionHandling()
                    .authenticationEntryPoint(new
RestAuthenticationEntryPoint())
                    .and()
                .authorizeRequests()
                    .antMatchers("/",
                        "/error",
                        "/favicon.ico",
                        "/**/*.png",
                        "/**/*.gif",
                        "/**/*.svg",
                        "/**/*.jpg",
                        "/**/*.html",
                        "/**/*.css",
                        "/**/*.js")
                        .permitAll()
                    .antMatchers("/auth/**", "/oauth2/**")
                        .permitAll()
                    .anyRequest()
                        .authenticated()
                    .and()
                .oauth2Login()
                    .authorizationEndpoint()
                        .baseUri("/oauth2/authorize")

.authorizationRequestRepository(cookieAuthorizationRequestRepository
())
                        .and()
                    .redirectionEndpoint()
```

```
                            .baseUri("/oauth2/callback/*")

                            .and()

                        .userInfoEndpoint()

                            .userService(customOAuth2UserService)

                            .and()


.successHandler(oAuth2AuthenticationSuccessHandler)


.failureHandler(oAuth2AuthenticationFailureHandler);



        // Add our custom Token based authentication filter

        http.addFilterBefore(tokenAuthenticationFilter(),
UsernamePasswordAuthenticationFilter.class);

    }

}
```

The above class basically ties up different components together to create an application-wide security policy. If you've read my Spring Security React full stack article, many of the components except the ones tied up with `oauth2Login()` will be familiar to you.

OAuth2 Login Flow

- The OAuth2 login flow will be initiated by the frontend client by sending the user to the endpoint `http://localhost:8080/oauth2/authorize/{provider}?redirect_uri=<redirect_uri_after_login>`.
  The `provider` path parameter is one of `google`, `facebook`, or `github`.
  The `redirect_uri` is the URI to which the user will be redirected once the authentication with the OAuth2 provider is successful. This is different from the OAuth2 redirectUri.
- On receiving the authorization request, Spring Security's OAuth2 client will redirect the user to the AuthorizationUrl of the supplied `provider`.
  All the state related to the authorization request is saved using the `authorizationRequestRepository` specified in the SecurityConfig.
  The user now allows/denies permission to your app on the provider's page. If the user allows permission to the app, the provider will redirect the user to the callback url `http://localhost:8080/oauth2/callback/{provider}` with an

authorization code. If the user denies the permission, he will be redirected to the same callbackUrl but with an `error`.

- If the OAuth2 callback results in an error, Spring security will invoke the `oAuth2AuthenticationFailureHandler` specified in the above `SecurityConfig`.
- If the OAuth2 callback is successful and it contains the authorization code, Spring Security will exchange the `authorization_code` for an `access_token` and invoke the `customOAuth2UserService` specified in the above SecurityConfig.
- The `customOAuth2UserService` retrieves the details of the authenticated user and creates a new entry in the database or updates the existing entry with the same email.
- Finally, the `oAuth2AuthenticationSuccessHandler` is invoked. It creates a JWT authentication token for the user and sends the user to the `redirect_uri` along with the JWT token in a query string.

Custom classes for OAuth2 Authentication

1. HttpCookieOAuth2AuthorizationRequestRepository

The OAuth2 protocol recommends using a `state` parameter to prevent CSRF attacks. During authentication, the application sends this parameter in the authorization request, and the OAuth2 provider returns this parameter unchanged in the OAuth2 callback.

The application compares the value of the `state` parameter returned from the OAuth2 provider with the value that it had sent initially. If they don't match then it denies the authentication request.

To achieve this flow, the application needs to store the `state` parameter somewhere so that it can later compare it with the `state` returned from the OAuth2 provider.

We'll be storing the `state` as well as the `redirect_uri` in a short-lived cookie. The following class provides functionality for storing the authorization request in cookies and retrieving it.

```
package com.example.springsocial.security.oauth2;



import com.example.springsocial.util.CookieUtils;

import com.nimbusds.oauth2.sdk.util.StringUtils;

import
org.springframework.security.oauth2.client.web.AuthorizationRequestR
epository;
```

```
import
org.springframework.security.oauth2.core.endpoint.OAuth2Authorizatio
nRequest;

import org.springframework.stereotype.Component;



import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;



@Component

public class HttpCookieOAuth2AuthorizationRequestRepository
implements
AuthorizationRequestRepository<OAuth2AuthorizationRequest> {

    public static final String
OAUTH2_AUTHORIZATION_REQUEST_COOKIE_NAME = "oauth2_auth_request";

    public static final String REDIRECT_URI_PARAM_COOKIE_NAME =
"redirect_uri";

    private static final int cookieExpireSeconds = 180;



    @Override

    public OAuth2AuthorizationRequest
loadAuthorizationRequest(HttpServletRequest request) {

        return CookieUtils.getCookie(request,
OAUTH2_AUTHORIZATION_REQUEST_COOKIE_NAME)

                .map(cookie -> CookieUtils.deserialize(cookie,
OAuth2AuthorizationRequest.class))

                .orElse(null);

    }



    @Override

    public void saveAuthorizationRequest(OAuth2AuthorizationRequest
authorizationRequest, HttpServletRequest request,
HttpServletResponse response) {

        if (authorizationRequest == null) {
```

```
            CookieUtils.deleteCookie(request, response,
OAUTH2_AUTHORIZATION_REQUEST_COOKIE_NAME);

            CookieUtils.deleteCookie(request, response,
REDIRECT_URI_PARAM_COOKIE_NAME);

            return;

        }



        CookieUtils.addCookie(response,
OAUTH2_AUTHORIZATION_REQUEST_COOKIE_NAME,
CookieUtils.serialize(authorizationRequest), cookieExpireSeconds);

        String redirectUriAfterLogin =
request.getParameter(REDIRECT_URI_PARAM_COOKIE_NAME);

        if (StringUtils.isNotBlank(redirectUriAfterLogin)) {

            CookieUtils.addCookie(response,
REDIRECT_URI_PARAM_COOKIE_NAME, redirectUriAfterLogin,
cookieExpireSeconds);

        }

    }



    @Override

    public OAuth2AuthorizationRequest
removeAuthorizationRequest(HttpServletRequest request) {

        return this.loadAuthorizationRequest(request);

    }



    public void removeAuthorizationRequestCookies(HttpServletRequest
request, HttpServletResponse response) {

        CookieUtils.deleteCookie(request, response,
OAUTH2_AUTHORIZATION_REQUEST_COOKIE_NAME);

        CookieUtils.deleteCookie(request, response,
REDIRECT_URI_PARAM_COOKIE_NAME);

    }

}
```

2. CustomOAuth2UserService

The `CustomOAuth2UserService` extends Spring Security's `DefaultOAuth2UserService` and implements its `loadUser()` method. This method is called after an access token is obtained from the OAuth2 provider.

In this method, we first fetch the user's details from the OAuth2 provider. If a user with the same email already exists in our database then we update his details, otherwise, we register a new user.

```
package com.example.springsocial.security.oauth2;



import
com.example.springsocial.exception.OAuth2AuthenticationProcessingExc
eption;

import com.example.springsocial.model.AuthProvider;

import com.example.springsocial.model.User;

import com.example.springsocial.repository.UserRepository;

import com.example.springsocial.security.UserPrincipal;

import com.example.springsocial.security.oauth2.user.OAuth2UserInfo;

import
com.example.springsocial.security.oauth2.user.OAuth2UserInfoFactory;

import org.springframework.beans.factory.annotation.Autowired;

import
org.springframework.security.authentication.InternalAuthenticationSe
rviceException;

import org.springframework.security.core.AuthenticationException;

import
org.springframework.security.oauth2.client.userinfo.DefaultOAuth2Use
rService;

import
org.springframework.security.oauth2.client.userinfo.OAuth2UserReques
t;

import
org.springframework.security.oauth2.core.OAuth2AuthenticationExcepti
on;

import org.springframework.security.oauth2.core.user.OAuth2User;
```

```java
import org.springframework.stereotype.Service;

import org.springframework.util.StringUtils;


import java.util.Optional;


@Service

public class CustomOAuth2UserService extends
DefaultOAuth2UserService {


    @Autowired

    private UserRepository userRepository;


    @Override

    public OAuth2User loadUser(OAuth2UserRequest oAuth2UserRequest)
throws OAuth2AuthenticationException {

        OAuth2User oAuth2User = super.loadUser(oAuth2UserRequest);


        try {

            return processOAuth2User(oAuth2UserRequest, oAuth2User);

        } catch (AuthenticationException ex) {

            throw ex;

        } catch (Exception ex) {

            // Throwing an instance of AuthenticationException will
trigger the OAuth2AuthenticationFailureHandler

            throw new
InternalAuthenticationServiceException(ex.getMessage(),
ex.getCause());

        }

    }
```

```
    private OAuth2User processOAuth2User(OAuth2UserRequest
oAuth2UserRequest, OAuth2User oAuth2User) {

        OAuth2UserInfo oAuth2UserInfo =
OAuth2UserInfoFactory.getOAuth2UserInfo(oAuth2UserRequest.getClientR
egistration().getRegistrationId(), oAuth2User.getAttributes());

        if(StringUtils.isEmpty(oAuth2UserInfo.getEmail())) {

            throw new OAuth2AuthenticationProcessingException("Email
not found from OAuth2 provider");

        }



        Optional<User> userOptional =
userRepository.findByEmail(oAuth2UserInfo.getEmail());

        User user;

        if(userOptional.isPresent()) {

            user = userOptional.get();


if(!user.getProvider().equals(AuthProvider.valueOf(oAuth2UserRequest
.getClientRegistration().getRegistrationId()))) {

                throw new
OAuth2AuthenticationProcessingException("Looks like you're signed up
with " +

                        user.getProvider() + " account. Please use
your " + user.getProvider() +

                        " account to login.");

            }

            user = updateExistingUser(user, oAuth2UserInfo);

        } else {

            user = registerNewUser(oAuth2UserRequest,
oAuth2UserInfo);

        }



        return UserPrincipal.create(user,
oAuth2User.getAttributes());

    }
```

```
    private User registerNewUser(OAuth2UserRequest
oAuth2UserRequest, OAuth2UserInfo oAuth2UserInfo) {

        User user = new User();



user.setProvider(AuthProvider.valueOf(oAuth2UserRequest.getClientReg
istration().getRegistrationId()));

        user.setProviderId(oAuth2UserInfo.getId());

        user.setName(oAuth2UserInfo.getName());

        user.setEmail(oAuth2UserInfo.getEmail());

        user.setImageUrl(oAuth2UserInfo.getImageUrl());

        return userRepository.save(user);

    }



    private User updateExistingUser(User existingUser,
OAuth2UserInfo oAuth2UserInfo) {

        existingUser.setName(oAuth2UserInfo.getName());

        existingUser.setImageUrl(oAuth2UserInfo.getImageUrl());

        return userRepository.save(existingUser);

    }



}
```

3. OAuth2UserInfo mapping

Every OAuth2 provider returns a different JSON response when we fetch the authenticated user's details. Spring security parses the response in the form of a generic `map` of key-value pairs.

The following classes are used to get the required details of the user from the generic `map` of key-value pairs -

OAuth2UserInfo

```java
package com.example.springsocial.security.oauth2.user;

import java.util.Map;

public abstract class OAuth2UserInfo {
    protected Map<String, Object> attributes;

    public OAuth2UserInfo(Map<String, Object> attributes) {
        this.attributes = attributes;
    }

    public Map<String, Object> getAttributes() {
        return attributes;
    }

    public abstract String getId();

    public abstract String getName();

    public abstract String getEmail();

    public abstract String getImageUrl();
}
```

FacebookOAuth2UserInfo

```java
package com.example.springsocial.security.oauth2.user;
```

```java
import java.util.Map;


public class FacebookOAuth2UserInfo extends OAuth2UserInfo {

    public FacebookOAuth2UserInfo(Map<String, Object> attributes) {

        super(attributes);

    }


    @Override

    public String getId() {

        return (String) attributes.get("id");

    }


    @Override

    public String getName() {

        return (String) attributes.get("name");

    }


    @Override

    public String getEmail() {

        return (String) attributes.get("email");

    }


    @Override

    public String getImageUrl() {

        if(attributes.containsKey("picture")) {

            Map<String, Object> pictureObj = (Map<String, Object>)
attributes.get("picture");

            if(pictureObj.containsKey("data")) {

                Map<String, Object>  dataObj = (Map<String, Object>)
pictureObj.get("data");
```

```
                if(dataObj.containsKey("url")) {
                    return (String) dataObj.get("url");
                }
            }
        }
        return null;
    }
}
```

GoogleOAuth2UserInfo

```
package com.example.springsocial.security.oauth2.user;

import java.util.Map;

public class GoogleOAuth2UserInfo extends OAuth2UserInfo {

    public GoogleOAuth2UserInfo(Map<String, Object> attributes) {
        super(attributes);
    }

    @Override
    public String getId() {
        return (String) attributes.get("sub");
    }

    @Override
    public String getName() {
        return (String) attributes.get("name");
```

```
    }

    @Override
    public String getEmail() {
        return (String) attributes.get("email");
    }

    @Override
    public String getImageUrl() {
        return (String) attributes.get("picture");
    }
}
```

GithubOAuth2UserInfo

```
package com.example.springsocial.security.oauth2.user;

import java.util.Map;

public class GithubOAuth2UserInfo extends OAuth2UserInfo {

    public GithubOAuth2UserInfo(Map<String, Object> attributes) {
        super(attributes);
    }

    @Override
    public String getId() {
        return ((Integer) attributes.get("id")).toString();
    }
```

```
    @Override
    public String getName() {
        return (String) attributes.get("name");
    }

    @Override
    public String getEmail() {
        return (String) attributes.get("email");
    }

    @Override
    public String getImageUrl() {
        return (String) attributes.get("avatar_url");
    }
}
```

OAuth2UserInfoFactory

```
package com.example.springsocial.security.oauth2.user;


import
com.example.springsocial.exception.OAuth2AuthenticationProcessingExc
eption;
import com.example.springsocial.model.AuthProvider;


import java.util.Map;


public class OAuth2UserInfoFactory {
```

```
    public static OAuth2UserInfo getOAuth2UserInfo(String
registrationId, Map<String, Object> attributes) {


if(registrationId.equalsIgnoreCase(AuthProvider.google.toString()))
{

            return new GoogleOAuth2UserInfo(attributes);

        } else if
(registrationId.equalsIgnoreCase(AuthProvider.facebook.toString()))
{

            return new FacebookOAuth2UserInfo(attributes);

        } else if
(registrationId.equalsIgnoreCase(AuthProvider.github.toString())) {

            return new GithubOAuth2UserInfo(attributes);

        } else {

            throw new
OAuth2AuthenticationProcessingException("Sorry! Login with " +
registrationId + " is not supported yet.");

        }

    }

}
```

4. OAuth2AuthenticationSuccessHandler

On successful authentication, Spring security invokes the `onAuthenticationSuccess()` method of the `OAuth2AuthenticationSuccessHandler` configured in `SecurityConfig`.

In this method, we perform some validations, create a JWT authentication token, and redirect the user to the `redirect_uri` specified by the client with the JWT token added in the query string -

```
package com.example.springsocial.security.oauth2;



import com.example.springsocial.config.AppProperties;

import com.example.springsocial.exception.BadRequestException;

import com.example.springsocial.security.TokenProvider;
```

```
import com.example.springsocial.util.CookieUtils;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.security.core.Authentication;

import
org.springframework.security.web.authentication.SimpleUrlAuthenticat
ionSuccessHandler;

import org.springframework.stereotype.Component;

import org.springframework.web.util.UriComponentsBuilder;

import javax.servlet.ServletException;

import javax.servlet.http.Cookie;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import java.io.IOException;

import java.net.URI;

import java.util.Optional;


import static
com.example.springsocial.security.oauth2.HttpCookieOAuth2Authorizati
onRequestRepository.REDIRECT_URI_PARAM_COOKIE_NAME;


@Component

public class OAuth2AuthenticationSuccessHandler extends
SimpleUrlAuthenticationSuccessHandler {


    private TokenProvider tokenProvider;


    private AppProperties appProperties;


    private HttpCookieOAuth2AuthorizationRequestRepository
httpCookieOAuth2AuthorizationRequestRepository;
```

```
    @Autowired

    OAuth2AuthenticationSuccessHandler(TokenProvider tokenProvider,
AppProperties appProperties,


HttpCookieOAuth2AuthorizationRequestRepository
httpCookieOAuth2AuthorizationRequestRepository) {

        this.tokenProvider = tokenProvider;

        this.appProperties = appProperties;

        this.httpCookieOAuth2AuthorizationRequestRepository =
httpCookieOAuth2AuthorizationRequestRepository;

    }



    @Override

    public void onAuthenticationSuccess(HttpServletRequest request,
HttpServletResponse response, Authentication authentication) throws
IOException, ServletException {

        String targetUrl = determineTargetUrl(request, response,
authentication);



        if (response.isCommitted()) {

            logger.debug("Response has already been committed.
Unable to redirect to " + targetUrl);

            return;

        }



        clearAuthenticationAttributes(request, response);

        getRedirectStrategy().sendRedirect(request, response,
targetUrl);

    }



    protected String determineTargetUrl(HttpServletRequest request,
HttpServletResponse response, Authentication authentication) {
```

```
        Optional<String> redirectUri =
CookieUtils.getCookie(request, REDIRECT_URI_PARAM_COOKIE_NAME)

                .map(Cookie::getValue);



        if(redirectUri.isPresent() &&
!isAuthorizedRedirectUri(redirectUri.get())) {

            throw new BadRequestException("Sorry! We've got an
Unauthorized Redirect URI and can't proceed with the
authentication");

        }



        String targetUrl =
redirectUri.orElse(getDefaultTargetUrl());



        String token = tokenProvider.createToken(authentication);



        return UriComponentsBuilder.fromUriString(targetUrl)

                .queryParam("token", token)

                .build().toUriString();

    }



    protected void clearAuthenticationAttributes(HttpServletRequest
request, HttpServletResponse response) {

        super.clearAuthenticationAttributes(request);


httpCookieOAuth2AuthorizationRequestRepository.removeAuthorizationRe
questCookies(request, response);

    }



    private boolean isAuthorizedRedirectUri(String uri) {

        URI clientRedirectUri = URI.create(uri);
```

```
        return appProperties.getOauth2().getAuthorizedRedirectUris()

                .stream()

                .anyMatch(authorizedRedirectUri -> {

                    // Only validate host and port. Let the clients
use different paths if they want to

                    URI authorizedURI =
URI.create(authorizedRedirectUri);


if(authorizedURI.getHost().equalsIgnoreCase(clientRedirectUri.getHos
t())

                            && authorizedURI.getPort() ==
clientRedirectUri.getPort()) {

                        return true;

                    }

                    return false;

                });

    }

}
```

5. OAuth2AuthenticationFailureHandler

In case of any error during OAuth2 authentication, Spring Security invokes the `onAuthenticationFailure()` method of the `OAuth2AuthenticationFailureHandler` that we have configured in `SecurityConfig`.

It sends the user to the frontend client with an error message added to the query string -

```
package com.example.springsocial.security.oauth2;


import com.example.springsocial.util.CookieUtils;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.security.core.AuthenticationException;
```

```
import
org.springframework.security.web.authentication.SimpleUrlAuthenticat
ionFailureHandler;

import org.springframework.stereotype.Component;

import org.springframework.web.util.UriComponentsBuilder;



import javax.servlet.ServletException;

import javax.servlet.http.Cookie;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import java.io.IOException;



import static
com.example.springsocial.security.oauth2.HttpCookieOAuth2Authorizati
onRequestRepository.REDIRECT_URI_PARAM_COOKIE_NAME;



@Component

public class OAuth2AuthenticationFailureHandler extends
SimpleUrlAuthenticationFailureHandler {



    @Autowired

    HttpCookieOAuth2AuthorizationRequestRepository
httpCookieOAuth2AuthorizationRequestRepository;



    @Override

    public void onAuthenticationFailure(HttpServletRequest request,
HttpServletResponse response, AuthenticationException exception)
throws IOException, ServletException {

        String targetUrl = CookieUtils.getCookie(request,
REDIRECT_URI_PARAM_COOKIE_NAME)

                .map(Cookie::getValue)

                .orElse(("/"));
```

```
        targetUrl = UriComponentsBuilder.fromUriString(targetUrl)

                .queryParam("error",
exception.getLocalizedMessage())

                .build().toUriString();



httpCookieOAuth2AuthorizationRequestRepository.removeAuthorizationRe
questCookies(request, response);



        getRedirectStrategy().sendRedirect(request, response,
targetUrl);

    }

}
```

Controllers and Services for Email based authentication

Let's now look at the controllers and services for handling email and password based login.

1. AuthController

```
package com.example.springsocial.controller;


import com.example.springsocial.exception.BadRequestException;

import com.example.springsocial.model.AuthProvider;

import com.example.springsocial.model.User;

import com.example.springsocial.payload.ApiResponse;

import com.example.springsocial.payload.AuthResponse;

import com.example.springsocial.payload.LoginRequest;

import com.example.springsocial.payload.SignUpRequest;

import com.example.springsocial.repository.UserRepository;

import com.example.springsocial.security.TokenProvider;

import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.http.ResponseEntity;

import
org.springframework.security.authentication.AuthenticationManager;

import
org.springframework.security.authentication.UsernamePasswordAuthenti
cationToken;

import org.springframework.security.core.Authentication;

import
org.springframework.security.core.context.SecurityContextHolder;

import org.springframework.security.crypto.password.PasswordEncoder;

import org.springframework.web.bind.annotation.*;

import
org.springframework.web.servlet.support.ServletUriComponentsBuilder;


import javax.validation.Valid;

import java.net.URI;


@RestController

@RequestMapping("/auth")

public class AuthController {


    @Autowired

    private AuthenticationManager authenticationManager;


    @Autowired

    private UserRepository userRepository;


    @Autowired

    private PasswordEncoder passwordEncoder;


    @Autowired
```

```
    private TokenProvider tokenProvider;


    @PostMapping("/login")

    public ResponseEntity<?> authenticateUser(@Valid @RequestBody
LoginRequest loginRequest) {


        Authentication authentication =
authenticationManager.authenticate(

                new UsernamePasswordAuthenticationToken(

                        loginRequest.getEmail(),

                        loginRequest.getPassword()

                )

        );



SecurityContextHolder.getContext().setAuthentication(authentication)
;


        String token = tokenProvider.createToken(authentication);

        return ResponseEntity.ok(new AuthResponse(token));

    }


    @PostMapping("/signup")

    public ResponseEntity<?> registerUser(@Valid @RequestBody
SignUpRequest signUpRequest) {

        if(userRepository.existsByEmail(signUpRequest.getEmail())) {

            throw new BadRequestException("Email address already in
use.");

        }


        // Creating user's account
```

```
        User user = new User();
        user.setName(signUpRequest.getName());
        user.setEmail(signUpRequest.getEmail());
        user.setPassword(signUpRequest.getPassword());
        user.setProvider(AuthProvider.local);

user.setPassword(passwordEncoder.encode(user.getPassword()));

        User result = userRepository.save(user);

        URI location = ServletUriComponentsBuilder
                .fromCurrentContextPath().path("/user/me")
                .buildAndExpand(result.getId()).toUri();

        return ResponseEntity.created(location)
                .body(new ApiResponse(true, "User registered
successfully@"));
    }

}
```

2. CustomUserDetailsService

```
package com.example.springsocial.security;

import com.example.springsocial.exception.ResourceNotFoundException;
import com.example.springsocial.model.User;
import com.example.springsocial.repository.UserRepository;
```

```
import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.security.core.userdetails.UserDetails;

import
org.springframework.security.core.userdetails.UserDetailsService;

import
org.springframework.security.core.userdetails.UsernameNotFoundExcept
ion;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;



@Service

public class CustomUserDetailsService implements UserDetailsService
{



    @Autowired

    UserRepository userRepository;



    @Override

    @Transactional

    public UserDetails loadUserByUsername(String email)

            throws UsernameNotFoundException {

        User user = userRepository.findByEmail(email)

                .orElseThrow(() ->

                        new UsernameNotFoundException("User not
found with email : " + email)

        );



        return UserPrincipal.create(user);

    }



    @Transactional
```

```
    public UserDetails loadUserById(Long id) {

        User user = userRepository.findById(id).orElseThrow(

            () -> new ResourceNotFoundException("User", "id", id)

        );


        return UserPrincipal.create(user);

    }

}
```

JWT Token provider, Authentication Filter, Authentication error handler, and UserPrincipal

TokenProvider

This class contains code to generate and verify Json Web Tokens -

```
package com.example.springsocial.security;


import com.example.springsocial.config.AppProperties;

import io.jsonwebtoken.*;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.security.core.Authentication;

import org.springframework.stereotype.Service;


import java.util.Date;


@Service

public class TokenProvider {
```

```
    private static final Logger logger =
LoggerFactory.getLogger(TokenProvider.class);


    private AppProperties appProperties;


    public TokenProvider(AppProperties appProperties) {

        this.appProperties = appProperties;

    }


    public String createToken(Authentication authentication) {

        UserPrincipal userPrincipal = (UserPrincipal)
authentication.getPrincipal();


        Date now = new Date();

        Date expiryDate = new Date(now.getTime() +
appProperties.getAuth().getTokenExpirationMsec());


        return Jwts.builder()

                .setSubject(Long.toString(userPrincipal.getId()))

                .setIssuedAt(new Date())

                .setExpiration(expiryDate)

                .signWith(SignatureAlgorithm.HS512,
appProperties.getAuth().getTokenSecret())

                .compact();

    }


    public Long getUserIdFromToken(String token) {

        Claims claims = Jwts.parser()

.setSigningKey(appProperties.getAuth().getTokenSecret())

                .parseClaimsJws(token)
```

```
                .getBody();


        return Long.parseLong(claims.getSubject());
    }


    public boolean validateToken(String authToken) {
        try {
Jwts.parser().setSigningKey(appProperties.getAuth().getTokenSecret()
).parseClaimsJws(authToken);
            return true;
        } catch (SignatureException ex) {
            logger.error("Invalid JWT signature");
        } catch (MalformedJwtException ex) {
            logger.error("Invalid JWT token");
        } catch (ExpiredJwtException ex) {
            logger.error("Expired JWT token");
        } catch (UnsupportedJwtException ex) {
            logger.error("Unsupported JWT token");
        } catch (IllegalArgumentException ex) {
            logger.error("JWT claims string is empty.");
        }
        return false;
    }
}
```

TokenAuthenticationFilter

This class is used to read JWT authentication token from the request, verify it, and set Spring Security's `SecurityContext` if the token is valid -

```
package com.example.springsocial.security;


import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Autowired;

import
org.springframework.security.authentication.UsernamePasswordAuthenti
cationToken;

import
org.springframework.security.core.context.SecurityContextHolder;

import org.springframework.security.core.userdetails.UserDetails;

import
org.springframework.security.web.authentication.WebAuthenticationDet
ailsSource;

import org.springframework.util.StringUtils;

import org.springframework.web.filter.OncePerRequestFilter;


import javax.servlet.FilterChain;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import java.io.IOException;


public class TokenAuthenticationFilter extends OncePerRequestFilter
{


    @Autowired

    private TokenProvider tokenProvider;


    @Autowired

    private CustomUserDetailsService customUserDetailsService;
```

```
    private static final Logger logger =
LoggerFactory.getLogger(TokenAuthenticationFilter.class);



    @Override

    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain filterChain) throws
ServletException, IOException {

        try {

            String jwt = getJwtFromRequest(request);



            if (StringUtils.hasText(jwt) &&
tokenProvider.validateToken(jwt)) {

                Long userId = tokenProvider.getUserIdFromToken(jwt);



                UserDetails userDetails =
customUserDetailsService.loadUserById(userId);

                UsernamePasswordAuthenticationToken authentication =
new UsernamePasswordAuthenticationToken(userDetails, null,
userDetails.getAuthorities());

                authentication.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));



SecurityContextHolder.getContext().setAuthentication(authentication)
;

            }

        } catch (Exception ex) {

            logger.error("Could not set user authentication in
security context", ex);

        }



        filterChain.doFilter(request, response);

    }
```

```
    private String getJwtFromRequest(HttpServletRequest request) {

        String bearerToken = request.getHeader("Authorization");

        if (StringUtils.hasText(bearerToken) &&
bearerToken.startsWith("Bearer ")) {

            return bearerToken.substring(7, bearerToken.length());

        }

        return null;

    }

}
```

RestAuthenticationEntryPoint

This class is invoked when a user tries to access a protected resource without authentication. In this case, we simply return a 401 Unauthorized response -

```
package com.example.springsocial.security;


import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.security.core.AuthenticationException;

import org.springframework.security.web.AuthenticationEntryPoint;


import javax.servlet.ServletException;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import java.io.IOException;


public class RestAuthenticationEntryPoint implements
AuthenticationEntryPoint {
```

```
    private static final Logger logger =
LoggerFactory.getLogger(RestAuthenticationEntryPoint.class);



    @Override

    public void commence(HttpServletRequest httpServletRequest,

                         HttpServletResponse httpServletResponse,

                         AuthenticationException e) throws
IOException, ServletException {

        logger.error("Responding with unauthorized error. Message -
{}", e.getMessage());


httpServletResponse.sendError(HttpServletResponse.SC_UNAUTHORIZED,

                e.getLocalizedMessage());

    }

}
```

UserPrincipal

The `UserPrincipal` class represents an authenticated Spring Security principal. It contains the details of the authenticated user -

```
package com.example.springsocial.security;



import com.example.springsocial.model.User;

import org.springframework.security.core.GrantedAuthority;

import
org.springframework.security.core.authority.SimpleGrantedAuthority;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.oauth2.core.user.OAuth2User;



import java.util.Collection;

import java.util.Collections;
```

```java
import java.util.List;

import java.util.Map;


public class UserPrincipal implements OAuth2User, UserDetails {

    private Long id;

    private String email;

    private String password;

    private Collection<? extends GrantedAuthority> authorities;

    private Map<String, Object> attributes;


    public UserPrincipal(Long id, String email, String password,
Collection<? extends GrantedAuthority> authorities) {

        this.id = id;

        this.email = email;

        this.password = password;

        this.authorities = authorities;

    }


    public static UserPrincipal create(User user) {

        List<GrantedAuthority> authorities = Collections.

                singletonList(new
SimpleGrantedAuthority("ROLE_USER"));


        return new UserPrincipal(

                user.getId(),

                user.getEmail(),

                user.getPassword(),

                authorities

        );

    }
```

```
    public static UserPrincipal create(User user, Map<String,
Object> attributes) {

        UserPrincipal userPrincipal = UserPrincipal.create(user);

        userPrincipal.setAttributes(attributes);

        return userPrincipal;

    }


    public Long getId() {

        return id;

    }


    public String getEmail() {

        return email;

    }


    @Override

    public String getPassword() {

        return password;

    }


    @Override

    public String getUsername() {

        return email;

    }


    @Override

    public boolean isAccountNonExpired() {

        return true;
```

```
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return authorities;
    }

    @Override
    public Map<String, Object> getAttributes() {
        return attributes;
    }

    public void setAttributes(Map<String, Object> attributes) {
        this.attributes = attributes;
```

```
    }


    @Override

    public String getName() {

        return String.valueOf(id);

    }

}
```

Current User meta annotation

This is a meta-annotation that can be used to inject the currently authenticated user principal in the controllers -

```
package com.example.springsocial.security;


import
org.springframework.security.core.annotation.AuthenticationPrincipal
;

import java.lang.annotation.*;


@Target({ElementType.PARAMETER, ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@AuthenticationPrincipal

public @interface CurrentUser {


}
```

UserController - User APIs

The `UserController` class contains a protected API to get the details of the currently authenticated user -

```
package com.example.springsocial.controller;


import com.example.springsocial.exception.ResourceNotFoundException;

import com.example.springsocial.model.User;

import com.example.springsocial.repository.UserRepository;

import com.example.springsocial.security.CurrentUser;

import com.example.springsocial.security.UserPrincipal;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.security.access.prepost.PreAuthorize;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RestController;


@RestController

public class UserController {


    @Autowired

    private UserRepository userRepository;


    @GetMapping("/user/me")

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

    public User getCurrentUser(@CurrentUser UserPrincipal
userPrincipal) {

        return userRepository.findById(userPrincipal.getId())

                .orElseThrow(() -> new
ResourceNotFoundException("User", "id", userPrincipal.getId()));

    }
```

```
}
```

Utility classes

The project uses some utility classes to perform various tasks -

CookieUtils

```
package com.example.springsocial.util;


import org.springframework.util.SerializationUtils;

import javax.servlet.http.Cookie;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import java.util.Base64;

import java.util.Optional;


public class CookieUtils {


    public static Optional<Cookie> getCookie(HttpServletRequest
request, String name) {

        Cookie[] cookies = request.getCookies();


        if (cookies != null && cookies.length > 0) {

            for (Cookie cookie : cookies) {

                if (cookie.getName().equals(name)) {

                    return Optional.of(cookie);

                }

            }

        }
```

```
        return Optional.empty();
    }

    public static void addCookie(HttpServletResponse response,
String name, String value, int maxAge) {
        Cookie cookie = new Cookie(name, value);
        cookie.setPath("/");
        cookie.setHttpOnly(true);
        cookie.setMaxAge(maxAge);
        response.addCookie(cookie);
    }

    public static void deleteCookie(HttpServletRequest request,
HttpServletResponse response, String name) {
        Cookie[] cookies = request.getCookies();
        if (cookies != null && cookies.length > 0) {
            for (Cookie cookie: cookies) {
                if (cookie.getName().equals(name)) {
                    cookie.setValue("");
                    cookie.setPath("/");
                    cookie.setMaxAge(0);
                    response.addCookie(cookie);
                }
            }
        }
    }

    public static String serialize(Object object) {
        return Base64.getUrlEncoder()
```

```
.encodeToString(SerializationUtils.serialize(object));
    }

    public static <T> T deserialize(Cookie cookie, Class<T> cls) {
        return cls.cast(SerializationUtils.deserialize(
Base64.getUrlDecoder().decode(cookie.getValue())));
    }
}
```

Request/Response Payloads

The following request/response payloads are used in our controller APIs -

1. LoginRequest

```
package com.example.springsocial.payload;

import javax.validation.constraints.Email;
import javax.validation.constraints.NotBlank;

public class LoginRequest {
    @NotBlank
    @Email
    private String email;

    @NotBlank
    private String password;

    // Getters and Setters (Omitted for brevity)
```

```
}
```

2. SignUpRequest

```
package com.example.springsocial.payload;

import javax.validation.constraints.Email;
import javax.validation.constraints.NotBlank;

public class SignUpRequest {
    @NotBlank
    private String name;

    @NotBlank
    @Email
    private String email;

    @NotBlank
    private String password;

    // Getters and Setters (Omitted for brevity)
}
```

3. AuthResponse

```
package com.example.springsocial.payload;

public class AuthResponse {
    private String accessToken;
```

```
    private String tokenType = "Bearer";

    public AuthResponse(String accessToken) {
        this.accessToken = accessToken;
    }

    // Getters and Setters (Omitted for brevity)
}
```

4. ApiResponse

```
package com.example.springsocial.payload;

public class ApiResponse {
    private boolean success;
    private String message;

    public ApiResponse(boolean success, String message) {
        this.success = success;
        this.message = message;
    }

    // Getters and Setters (Omitted for brevity)
}
```

Exception Classes

The following exception classes are used throughout the application for various error cases -

1. BadRequestException

```java
package com.example.springsocial.exception;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(HttpStatus.BAD_REQUEST)
public class BadRequestException extends RuntimeException {
    public BadRequestException(String message) {
        super(message);
    }

    public BadRequestException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

2. ResourceNotFoundException

```java
package com.example.springsocial.exception;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException {
    private String resourceName;
    private String fieldName;
    private Object fieldValue;
```

```
    public ResourceNotFoundException(String resourceName, String
fieldName, Object fieldValue) {

        super(String.format("%s not found with %s : '%s'",
resourceName, fieldName, fieldValue));

        this.resourceName = resourceName;

        this.fieldName = fieldName;

        this.fieldValue = fieldValue;

    }


    public String getResourceName() {

        return resourceName;

    }


    public String getFieldName() {

        return fieldName;

    }


    public Object getFieldValue() {

        return fieldValue;

    }

}
```

3. OAuth2AuthenticationProcessingException

```
package com.example.springsocial.exception;


import org.springframework.security.core.AuthenticationException;
```

```
public class OAuth2AuthenticationProcessingException extends
AuthenticationException {

    public OAuth2AuthenticationProcessingException(String msg,
Throwable t) {

        super(msg, t);

    }



    public OAuth2AuthenticationProcessingException(String msg) {

        super(msg);

    }

}
```

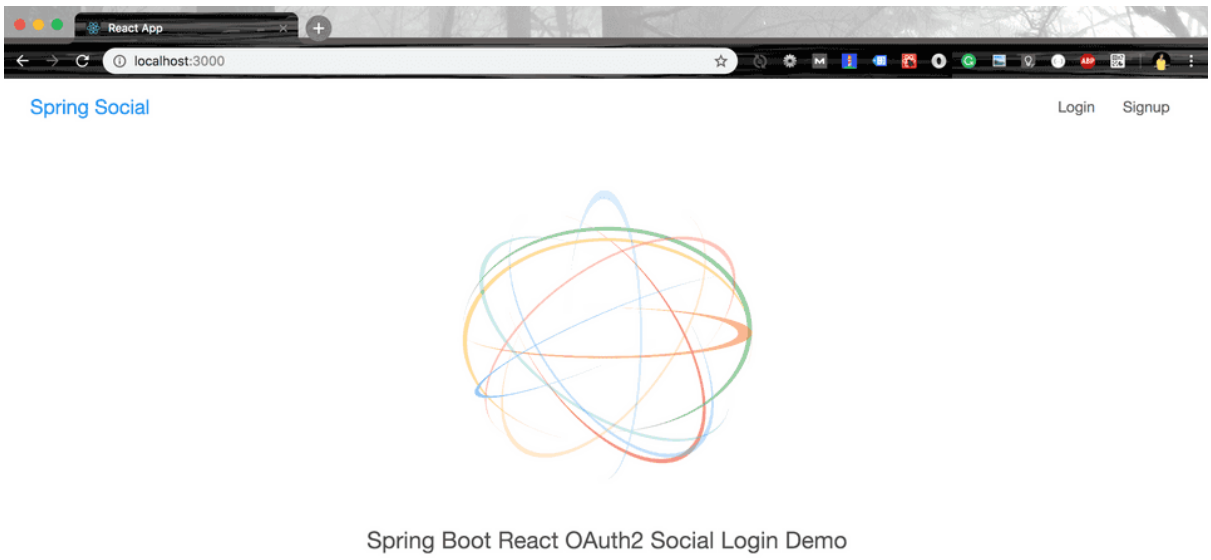
You can learn more about Spring Security's OAuth2 Login from the **official documentation**.

To explain Security implementation, we'll develop the frontend client with react.

Spring Boot OAuth2 Social Login with Google, Facebook, and Github –

Front End Implementation

The below diagram show sample REACT app for social login demo functionality.



You can find the complete source code of the application on Github.

Creating the React application

Let's create the React app using `create-react-app` CLI client. You can install `create-react-app` using `npm` like this -

```
npm install -g create-react-app
```

Now create the React app by typing the following command -

```
create-react-app react-social
```

We'll be using `react-router-dom` for client-side routing and `react-s-alert` for showing alerts. Download these dependencies by typing the following commands -

```
cd react-social

npm install react-router-dom react-s-alert --save
```

Directory Structure

Here is the directory structure of the application for your reference -

```
REACT-SOCIAL
  node_modules
  public
  src
    app
      App.css
      App.js
      App.test.js
    common
      AppHeader.css
      AppHeader.js
      LoadingIndicator.js
      NotFound.css
      NotFound.js
      PrivateRoute.js
    constants
      index.js
    home
      Home.css
      Home.js
    img
    user
      login
        Login.css
        Login.js
      oauth2
        OAuth2RedirectHandler.js
      profile
        Profile.css
        Profile.js
      signup
        Signup.css
        Signup.js
    util
      APIUtils.js
    index.css
    index.js
    logo.svg
    registerServiceWorker.js
  .gitignore
```

Understanding the frontend code

index.js

This is the entry point of our application -

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './app/App';
import registerServiceWorker from './registerServiceWorker';
import { BrowserRouter as Router } from 'react-router-dom';

ReactDOM.render(
    <Router>
        <App />
    </Router>,
    document.getElementById('root')
);

registerServiceWorker();
```

It renders the `App` component in a DOM element with id `root` (This DOM element is available inside `public/index.html` file). The `App` component is wrapped inside react-router's `Router` to enable client-side routing.

src/app/App.js

App.js is the main top-level component of our application. It defines the basic layout and the routes. It also loads the details of the currently authenticated user from backend and passes the detail to its child components.

```
import React, { Component } from 'react';
import {
```

```
  Route,
  Switch
} from 'react-router-dom';
import AppHeader from '../common/AppHeader';
import Home from '../home/Home';
import Login from '../user/login/Login';
import Signup from '../user/signup/Signup';
import Profile from '../user/profile/Profile';
import OAuth2RedirectHandler from
'../user/oauth2/OAuth2RedirectHandler';
import NotFound from '../common/NotFound';
import LoadingIndicator from '../common/LoadingIndicator';
import { getCurrentUser } from '../util/APIUtils';
import { ACCESS_TOKEN } from '../constants';
import PrivateRoute from '../common/PrivateRoute';
import Alert from 'react-s-alert';
import 'react-s-alert/dist/s-alert-default.css';
import 'react-s-alert/dist/s-alert-css-effects/slide.css';
import './App.css';

class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      authenticated: false,
      currentUser: null,
      loading: true
    }
```

```
    this.loadCurrentlyLoggedInUser =
this.loadCurrentlyLoggedInUser.bind(this);

    this.handleLogout = this.handleLogout.bind(this);

  }


  loadCurrentlyLoggedInUser() {

    getCurrentUser()

    .then(response => {

      this.setState({

        currentUser: response,

        authenticated: true,

        loading: false

      });

    }).catch(error => {

      this.setState({

        loading: false

      });

    });

  }


  handleLogout() {

    localStorage.removeItem(ACCESS_TOKEN);

    this.setState({

      authenticated: false,

      currentUser: null

    });

    Alert.success("You're safely logged out!");

  }
```

```
  componentDidMount() {
    this.loadCurrentlyLoggedInUser();
  }

  render() {
    if(this.state.loading) {
      return <LoadingIndicator />
    }

    return (
      <div className="app">
        <div className="app-top-box">
          <AppHeader authenticated={this.state.authenticated}
onLogout={this.handleLogout} />
        </div>
        <div className="app-body">
          <Switch>
            <Route exact path="/" component={Home}></Route>
            <PrivateRoute path="/profile"
authenticated={this.state.authenticated}
currentUser={this.state.currentUser}
              component={Profile}></PrivateRoute>
            <Route path="/login"
              render={(props) => <Login
authenticated={this.state.authenticated} {...props} />}></Route>
            <Route path="/signup"
              render={(props) => <Signup
authenticated={this.state.authenticated} {...props} />}></Route>
            <Route path="/oauth2/redirect"
component={OAuth2RedirectHandler}></Route>
            <Route component={NotFound}></Route>
```

```
            </Switch>
          </div>
          <Alert stack={{limit: 3}}
            timeout = {3000}
            position='top-right' effect='slide' offset={65} />
        </div>
      );
    }
  }

export default App;
```

src/user/login/Login.js

The `Login` component allows users to login using an OAuth2 provider or an email and password.

```
import React, { Component } from 'react';
import './Login.css';
import { GOOGLE_AUTH_URL, FACEBOOK_AUTH_URL, GITHUB_AUTH_URL,
ACCESS_TOKEN } from '../../constants';
import { login } from '../../util/APIUtils';
import { Link, Redirect } from 'react-router-dom'
import fbLogo from '../../img/fb-logo.png';
import googleLogo from '../../img/google-logo.png';
import githubLogo from '../../img/github-logo.png';
import Alert from 'react-s-alert';

class Login extends Component {
    componentDidMount() {
```

```
        // If the OAuth2 login encounters an error, the user is
redirected to the /login page with an error

        // Here we display the error and then remove the error query
parameter from the location.

        if(this.props.location.state &&
this.props.location.state.error) {

            setTimeout(() => {

                Alert.error(this.props.location.state.error, {

                    timeout: 5000

                });

                this.props.history.replace({

                    pathname: this.props.location.pathname,

                    state: {}

                });

            }, 100);

        }

    }


    render() {

        if(this.props.authenticated) {

            return <Redirect

                to={{

                pathname: "/",

                state: { from: this.props.location }

            }}/>;

        }


        return (

            <div className="login-container">

                <div className="login-content">
```

```
                        <h1 className="login-title">Login to
SpringSocial</h1>

                        <SocialLogin />

                        <div className="or-separator">

                            <span className="or-text">OR</span>

                        </div>

                        <LoginForm {...this.props} />

                        <span className="signup-link">New user? <Link
to="/signup">Sign up!</Link></span>

                    </div>

                </div>

            );

        }

}


class SocialLogin extends Component {

    render() {

        return (

            <div className="social-login">

                <a className="btn btn-block social-btn google"
href={GOOGLE_AUTH_URL}>

                        <img src={googleLogo} alt="Google" /> Log in
with Google</a>

                <a className="btn btn-block social-btn facebook"
href={FACEBOOK_AUTH_URL}>

                        <img src={fbLogo} alt="Facebook" /> Log in with
Facebook</a>

                <a className="btn btn-block social-btn github"
href={GITHUB_AUTH_URL}>

                        <img src={githubLogo} alt="Github" /> Log in
with Github</a>

            </div>
```

```
        );
    }
}



class LoginForm extends Component {
    constructor(props) {
        super(props);
        this.state = {
            email: '',
            password: ''
        };
        this.handleInputChange = this.handleInputChange.bind(this);
        this.handleSubmit = this.handleSubmit.bind(this);
    }

    handleInputChange(event) {
        const target = event.target;
        const inputName = target.name;
        const inputValue = target.value;

        this.setState({
            [inputName] : inputValue
        });
    }

    handleSubmit(event) {
        event.preventDefault();
```

```
        const loginRequest = Object.assign({}, this.state);


        login(loginRequest)

        .then(response => {

            localStorage.setItem(ACCESS_TOKEN,
response.accessToken);

            Alert.success("You're successfully logged in!");

            this.props.history.push("/");

        }).catch(error => {

            Alert.error((error && error.message) || 'Oops! Something
went wrong. Please try again!');

        });

    }


    render() {

        return (

            <form onSubmit={this.handleSubmit}>

                <div className="form-item">

                    <input type="email" name="email"

                        className="form-control" placeholder="Email"

                        value={this.state.email}
onChange={this.handleInputChange} required/>

                </div>

                <div className="form-item">

                    <input type="password" name="password"

                        className="form-control"
placeholder="Password"

                        value={this.state.password}
onChange={this.handleInputChange} required/>

                </div>

                <div className="form-item">
```

```
                        <button type="submit" className="btn btn-block
btn-primary">Login</button>

                    </div>

                </form>

            );

        }

    }



export default Login
```

OAuth2RedirectHandler.js

This component is loaded when the user has completed the OAuth2 authentication flow with the server. The server redirects the user to this page with an access token if the authentication was successful, or an error if it failed.

```
import React, { Component } from 'react';

import { ACCESS_TOKEN } from '../../constants';

import { Redirect } from 'react-router-dom'


class OAuth2RedirectHandler extends Component {

    getUrlParameter(name) {

        name = name.replace(/[\[]/, '\\[').replace(/[\]]/, '\\]');

        var regex = new RegExp('[\\?&]' + name + '=([^&#]*)');


        var results = regex.exec(this.props.location.search);

        return results === null ? '' :
decodeURIComponent(results[1].replace(/\+/g, ' '));

    };


    render() {
```

```
        const token = this.getUrlParameter('token');
        const error = this.getUrlParameter('error');

        if(token) {
            localStorage.setItem(ACCESS_TOKEN, token);
            return <Redirect to={{
                pathname: "/profile",
                state: { from: this.props.location }
            }}/>;
        } else {
            return <Redirect to={{
                pathname: "/login",
                state: {
                    from: this.props.location,
                    error: error
                }
            }}/>;
        }
    }
}

export default OAuth2RedirectHandler;
```

On successful authentication, this component reads the token from the query string, saves it in localStorage and redirects the user to the `/profile` page.

If the authentication failed, then this component redirects the user to the `/login` page with an error -

Running the react app

You can run the react-social app using `npm` like this -

```
cd react-social

npm install && npm start
```

The above command will install any missing dependencies and start the app on port `3000`.

If you just want to build the app, then type the following command -

```
npm run build
```

The above command will create an optimized production build in a directory named `build/`.

Conclusion

Social as well as email and password based login to your spring boot application.

Appendix – A

Usage of Google OUATH Identity platform.

Pricing table

Pricing for Identity Platform is divided into different tiers based on the authentication method used.

Tier 1 providers

- Email
- Phone
- Anonymous
- Social

Monthly Active Users (MAU)	Price per MAU ($)
0 - 49,999	0
50,000 - 99,999	0.0055
100,000 - 999,999	0.0046

Monthly Active Users (MAU)	Price per MAU ($)
1,000,000 - 9,999,999	0.0032
10,000,000 +	0.0025

If you pay in a currency other than USD, the prices listed in your currency on Cloud Platform SKUs apply.

Tier 2 providers

- OpenID Connect (OIDC)
- Security Assertion Markup Language (SAML)

Monthly Active Users (MAU)	Price per MAU ($)
0 - 49	0
50 +	0.015

If you pay in a currency other than USD, the prices listed in your currency on Cloud Platform SKUs apply.

Phone authentication

Country	Price per verification ($)
First 10,000 successful verifications	Free
US, Canada, India	0.01
All other countries	0.06

If you pay in a currency other than USD, the prices listed in your currency on Cloud Platform SKUs apply.

Multi-factor authentication

Country	Price per verification ($)
First 100 successful verifications	Free
US, Canada, India	0.01
All other countries	0.06

If you pay in a currency other than USD, the prices listed in your currency on Cloud Platform SKUs apply.

Cloud Functions

Any functions you create are billed at the normal Cloud Functions rate. See the pricing information for Cloud Functions to learn more.

Viewing usage

To view your current Identity Platform usage:

1. Open the **Billing** page in the console.
   GO TO THE BILLING PAGE
2. Open the **Reports** tab.
3. Under **Filters**, select **Identity Platform** and **Firebase Authentication** from the **Products** dropdown menu.

The chart shows your current billing amount. If your usage is below the free tier allowance, the graph will show a flat line.

The table breaks down costs by authentication method. Charges from tier 1 and 2 providers are listed as such. Phone and multi-factor charges are listed as Firebase Authentication.

Pricing examples

The following table includes example Identity Platform usage patterns for three variations of applications and services, and the potential cost per month.

Authentication Type	Example Application or Service		
	Consumer App	Enterprise SaaS	Hybrid Service
Anonymous users	5,000	0	1,000
Email users with or without password	70,000	45,000	60,000
Social users from Google, Facebook, etc.	150,000	0	70,000
Sub-total	225,000 MAU 50,000 * 0 = $0 50,000 * 0.0055 = $275 125,000 * 0.0046 = $575 Sub-total: $850	45,000 MAU 45,000 * 0 = $0 Sub-total: $0	131,000 MAU 50,000 * 0 = $0 50,000 * 0.0055 = $275 31,000 * 0.0046 = $143 Sub-total: $418
Phone / SMS verifications	11,000 to Australia	0	12,000 to Canada
Sub-total	11,000 phone verifications 10,000 * Free = $0 1,000 * 0.06 = $60 Sub-total: $60	$0	12,000 phone verifications 10,000 * Free = $0 2,000 * 0.01 = $20 Sub-total: $20
Federated SAML users	0	2,000	1,000

Authentication Type	Example Application or Service		
	Consumer App	Enterprise SaaS	Hybrid Service
Federated OIDC users	0	75,000	9,000
Sub-total	$0	77,000 MAU 50 * 0 = $0 76,950 * 0.015 = $1154 Sub-total: $1154	10,000 MAU 50 * 0 = $0 9,950 * 0.015 = $149 Sub-total: $149
Total cost (monthly)	$910	$1154	$587

The above examples demonstrate the following common usage trends:

- **Consumer Applications** usually rely on users that sign up and sign in with credentials from social providers, or users created directly in Identity Platform using phone and email authentication. You might also want to make use of anonymous users who are testing your application and can be upgraded to full user accounts to maintain their state and user identifier. The above example also uses an SMS one-time password to sign in to the application.
- **Enterprise SaaS Services** usually use federation to sign in to the service because organizations want to maintain centralized and organizational control of their identities. The above example still has some users who sign up for accounts within the Identity Platform service, but there are a large number of users who are using OIDC for federation. There are also some users who are using SAML federation.
- **Hybrid Services** are a blend of customers, partners, employees, and anyone else who needs access to an application or service. The example above has a broad mix of sign-in methods by the creation of local accounts and the use of enterprise federation technologies like SAML and OIDC.

Appendix - B

How to get a Facebook application id and secret key in 2022?

Facebook has changed its developer portal user interfaces often. The Steps illustrated is for Facebook application id and secret key as per latest developer portal steps.

If you ever need to integrate Facebook Login or use Facebook graph API,

How To Get An App ID and Secret Key From Facebook

You have to register for a new Facebook App to get a Facebook App ID and Secret Key. It is very easy to set up an account and it is free of cost where your application does not need to do anything. For the further process, we only need the keys.

In case you have a Facebook App, you can directly use its App ID and Secret Key without any registration to a new account.

- First, you have to navigate your browser to the Facebook Developers page. For the same, you have to login into your Facebook account.
- Next, you need to Click the "Add a New App" link located in the top right "My Apps" menu.

Login Process for Facebook API:

To use the Facebook API, like the Login with Facebook or Facebook Graph API, you need to create a Facebook App. When you make a Facebook App, that app will have an App ID and an App Secret. You need these credentials to do almost anything with Facebook, including going through the OAuth authorization flow and working with Facebook's Graph API.

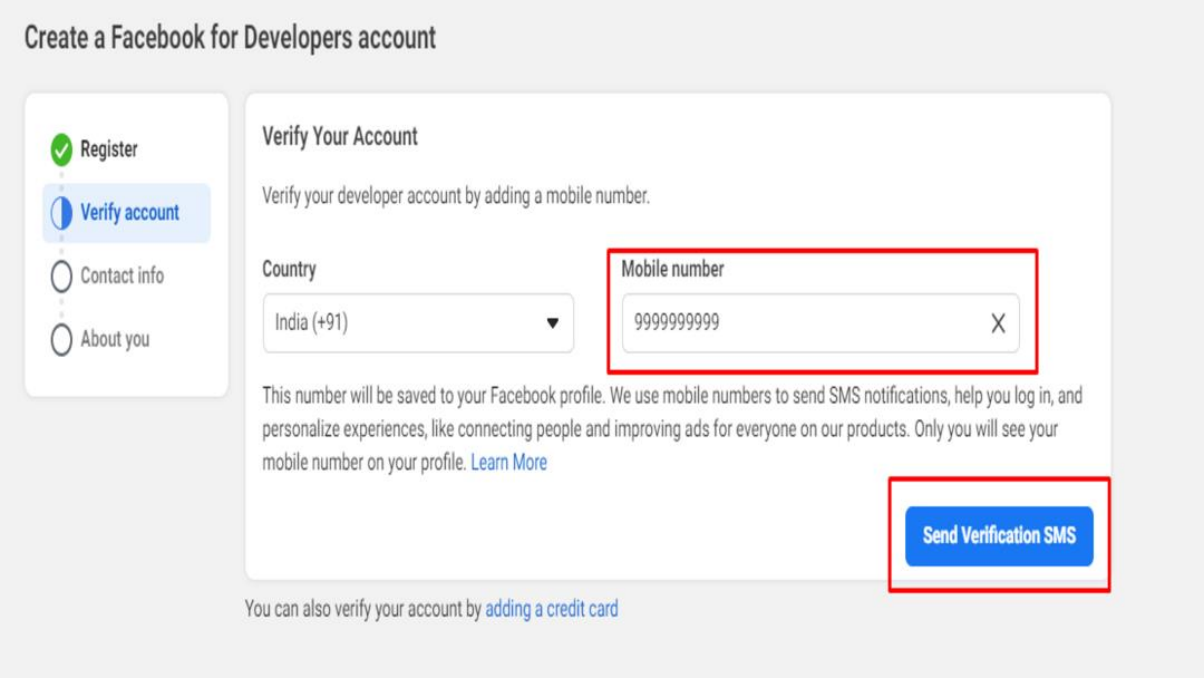
With the App ID, you can send API requests to Facebook for data. The Facebook App Secret can be used to decode encrypted data.

Follow the below steps to create Facebook application:

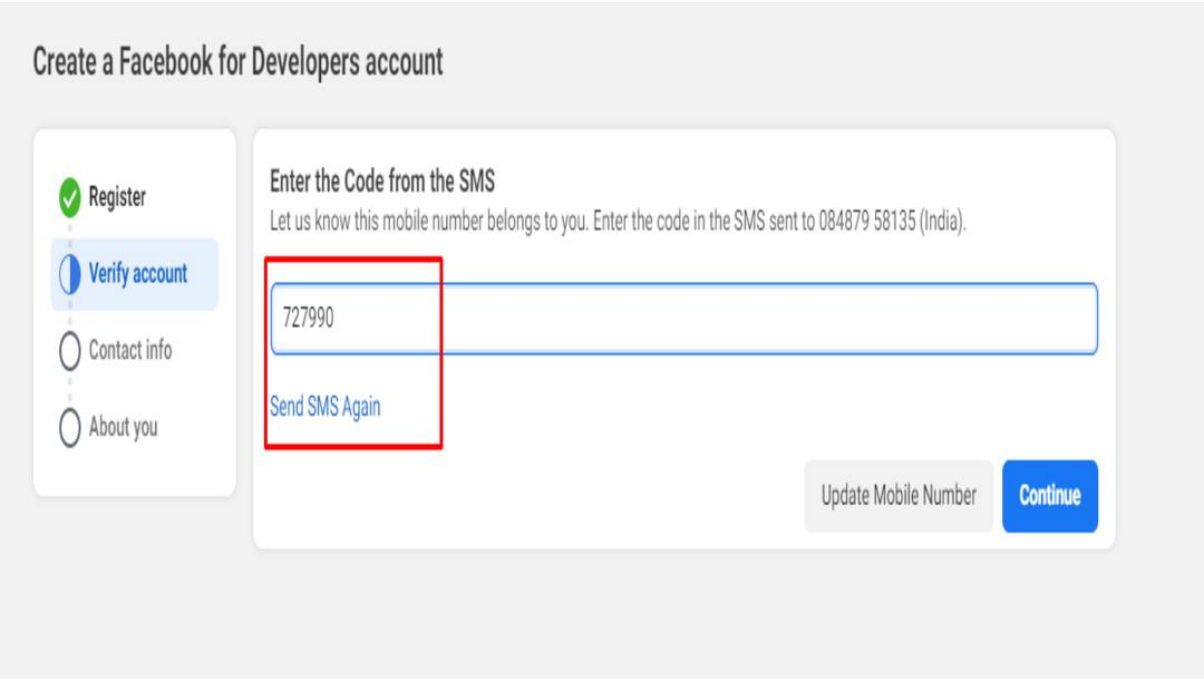
1. Login to your Facebook account and navigate to **https://developers.facebook.com**

2. If you already have an account registered for a Facebook developer account, you can skip this step and directly go to step 3. If you are not registered on a Facebook developer account, then you need to follow these steps:

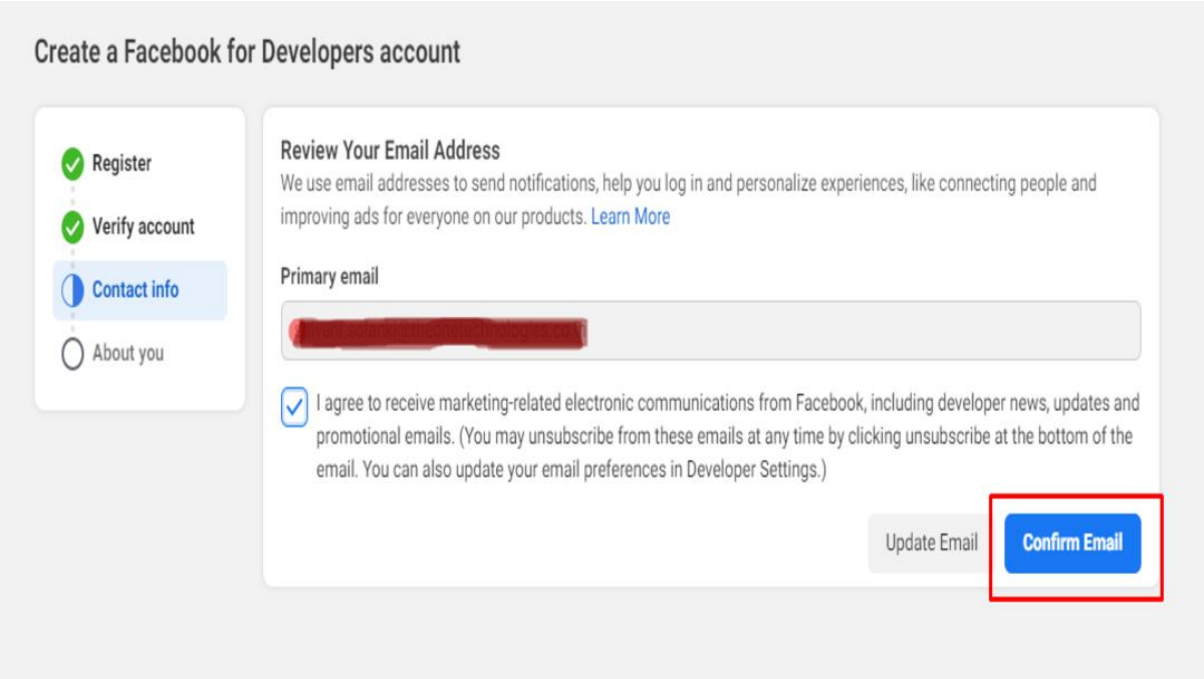
- Click on Get Started button from the top right corner.
- Now, you need to click on the “Next button” from the popup.



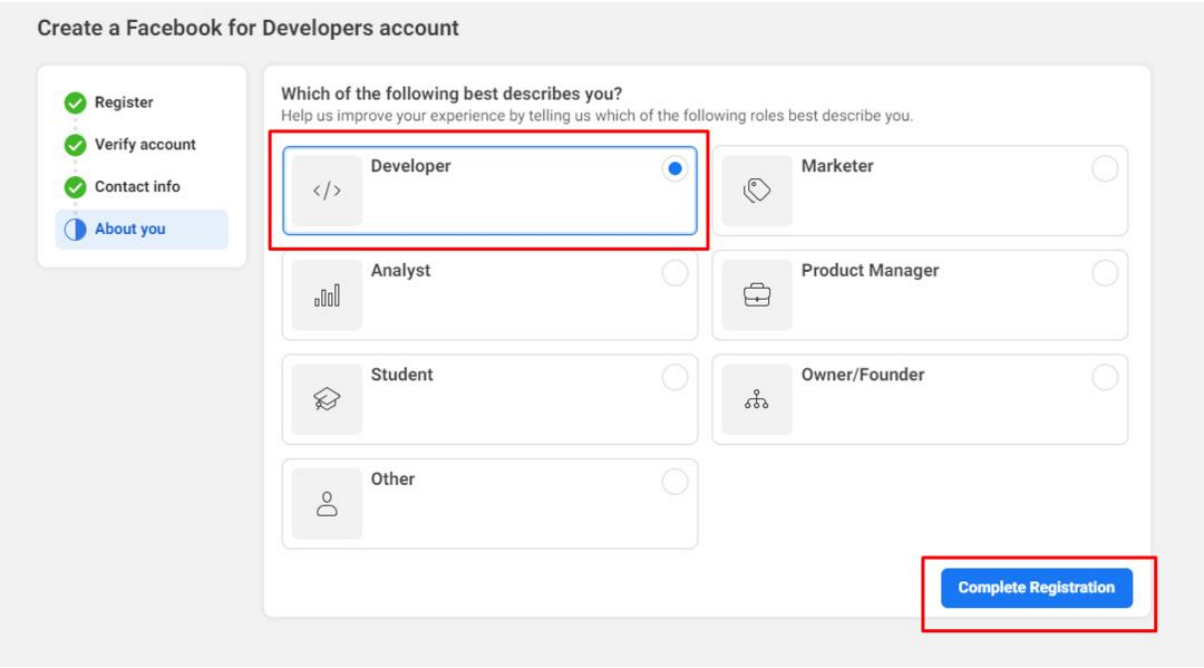
- Enter your verification code to verify your account through your mail ID.



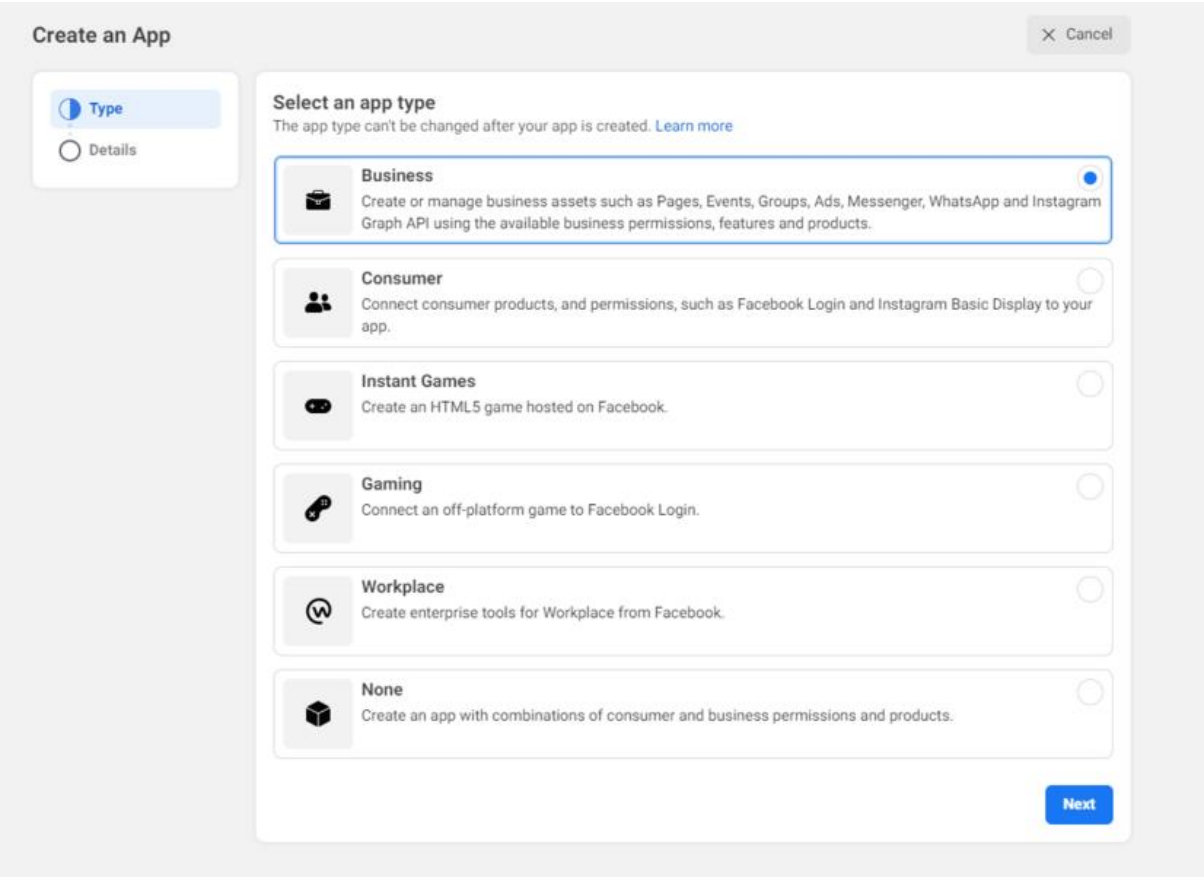
- It may be needed to Re Verify your account by your mail ID.



- On the next screen, after verifying your account, 3rd step is to tell you about it. Select the Developer option here.

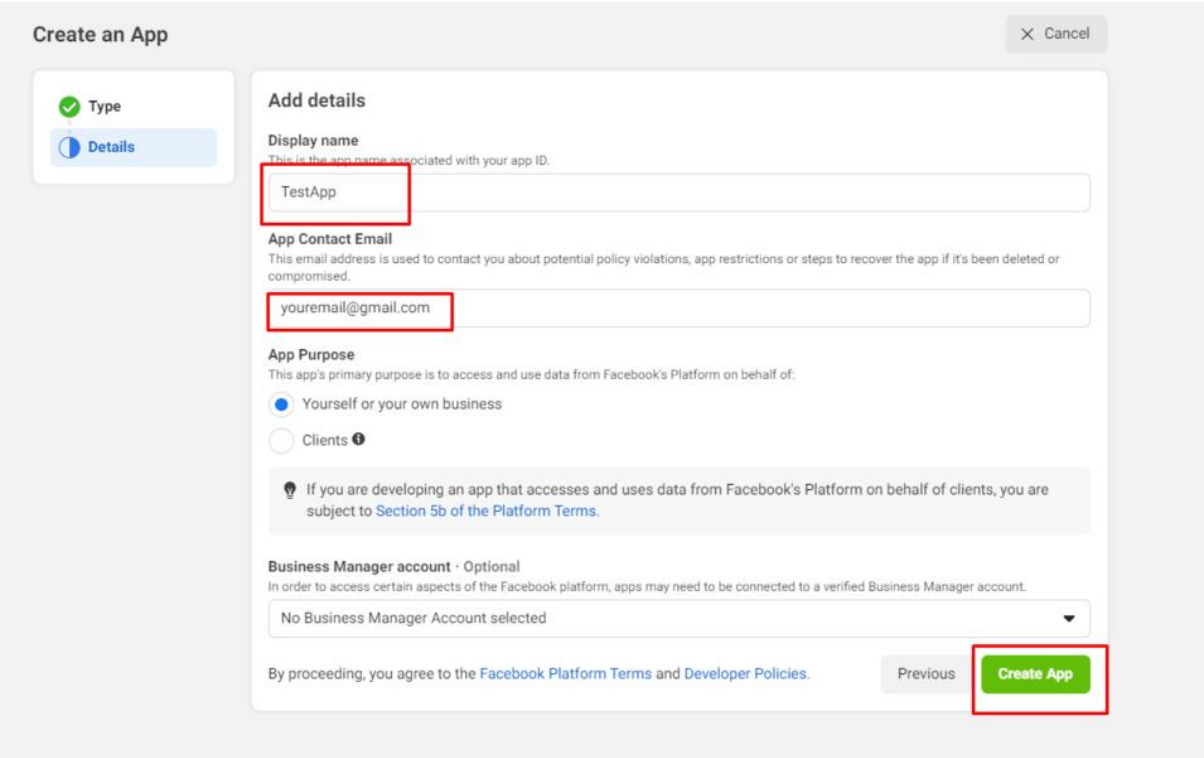


- On welcome screen, click on Create First App button. Here you'll be asked for Display Name and Contact Email of your New App ID. This app name will be shown to end-users when they will try to “Login with Facebook” and they are redirected to the Facebook website for login permission.
- So, make sure you give a meaningful name here which can identify your website.

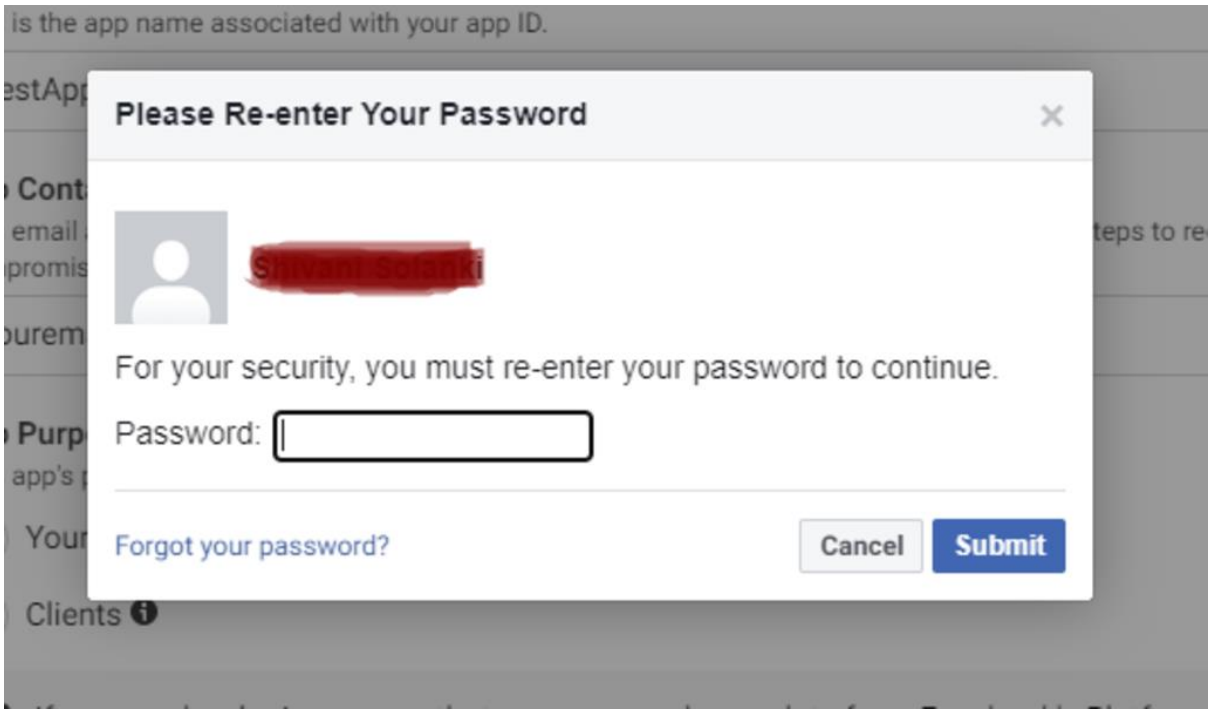


3. If you are already registered with Facebook developer perform below steps:

- Click on “My App” >> “Add a New App”

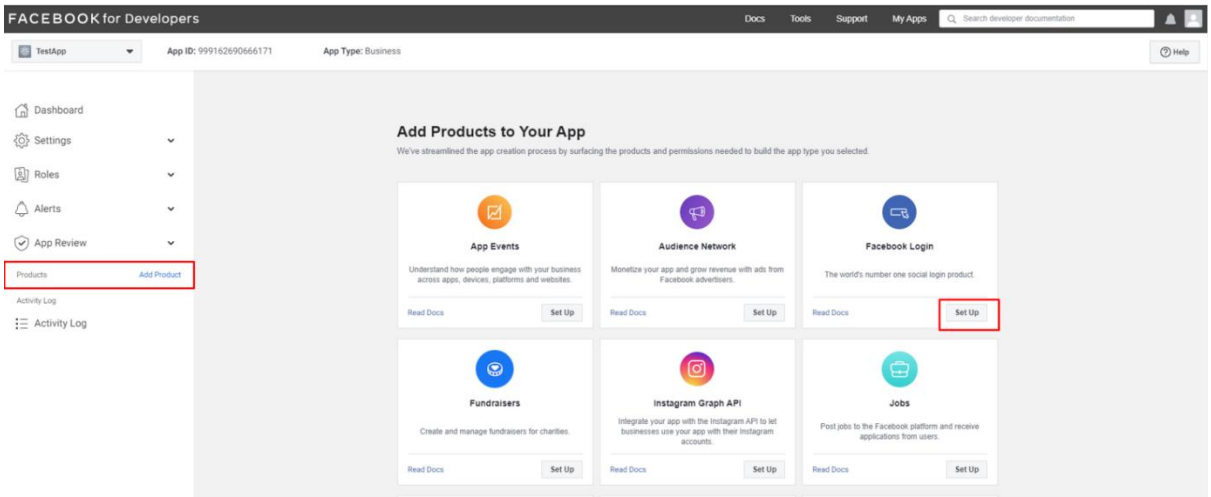


- From the popup enter “Display Name” of your new application and "Contact Email”, then click on the “Create App Id” Button.

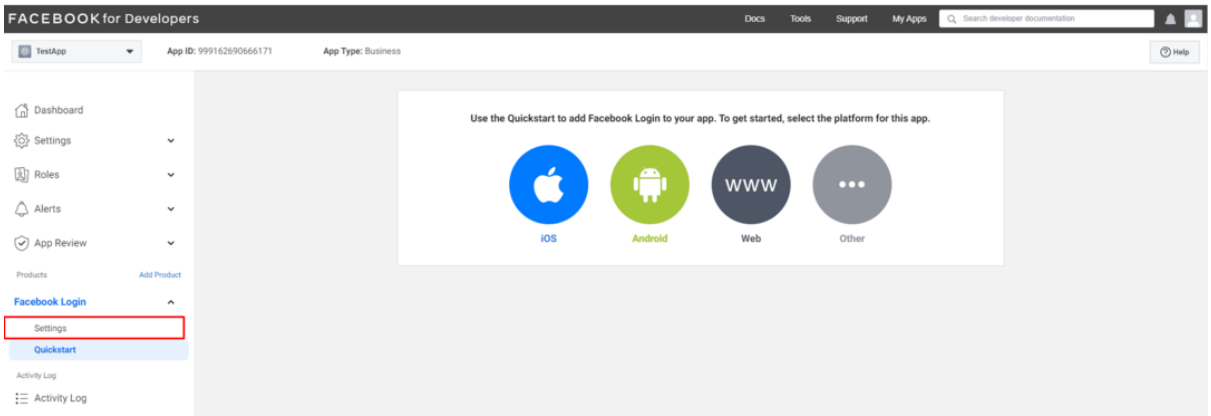


- Complete the Security Check entering the password and click on “Submit button” and it will open the product list page.

4. From the Product list page click on “Set Up” button from product with name “Facebook Login”.

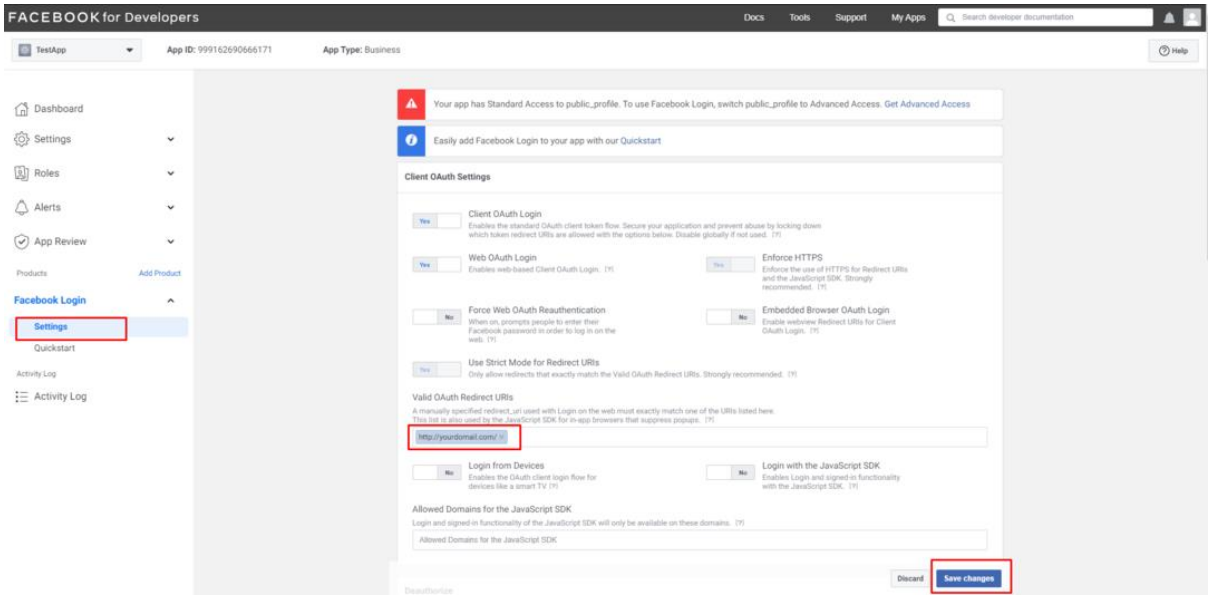


5. In the menu at the left corner, click on the “Facebook Login” link to expand the sub menu. Next, you need to Click on “Settings” from the sub-menu.

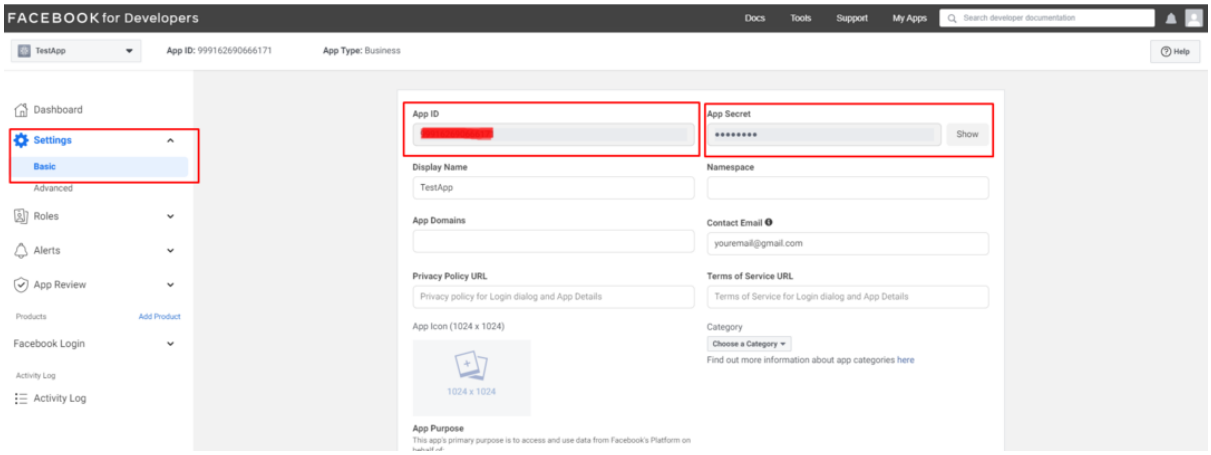


6. Now, you will see the details where you have to enter your website URL in “Valid OAuth redirect URIs”.

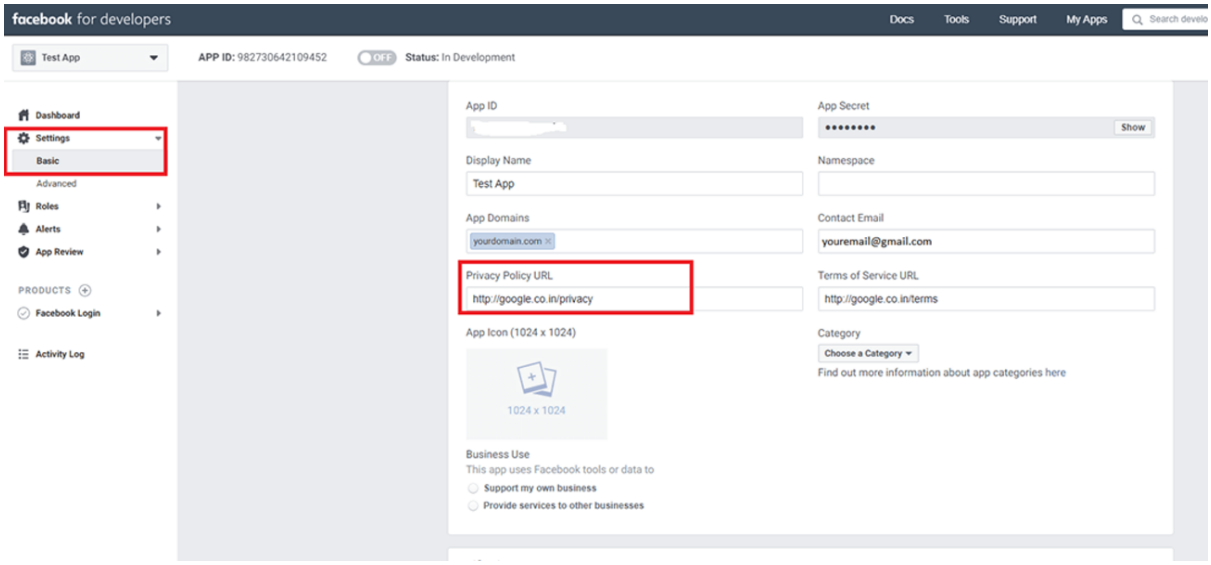
For example, my website domain is yourdomain.com so I entered https://yourdomain.com



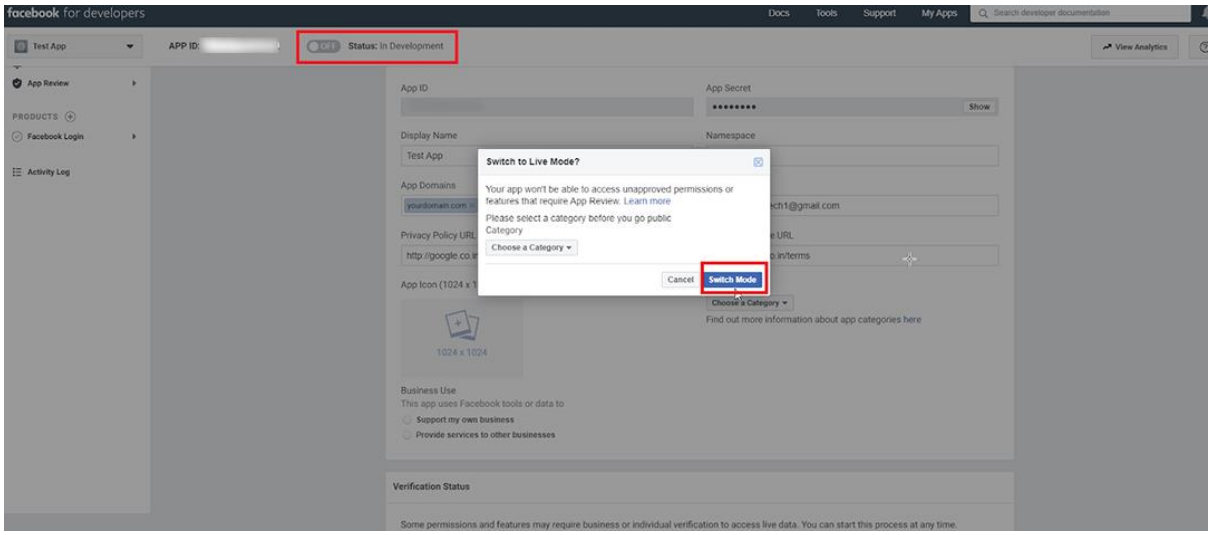
7. Now expand the Setting menu and select Basic. Here you can find the App ID and App Secret. Then click on the “Show” button in the “App Secret” text box. You can copy the “App Id” and “App Secret” which you can use for your Facebook API calls.



8. By default, when you create Facebook application, it is private and available only to you for testing purpose. End users can only use it after you make the app live. For the live app you must enter a Privacy Policy URL at setting > Basic.



9. For Switching the mode from Development to Live, click on the status switch. After that, it will show a popup to switch the mode. Please select the category of your app and click on the “Switch Mode” button to make your app public.



In a nutshell, It is an easy process to get a Facebook app ID and Facebook secret codes from Facebook. Using Facebook OAuth settings while following above mentioned steps can help you get a Facebook application id and secret key in 2022. You can set it live from the development version by just switching the mode in one click.