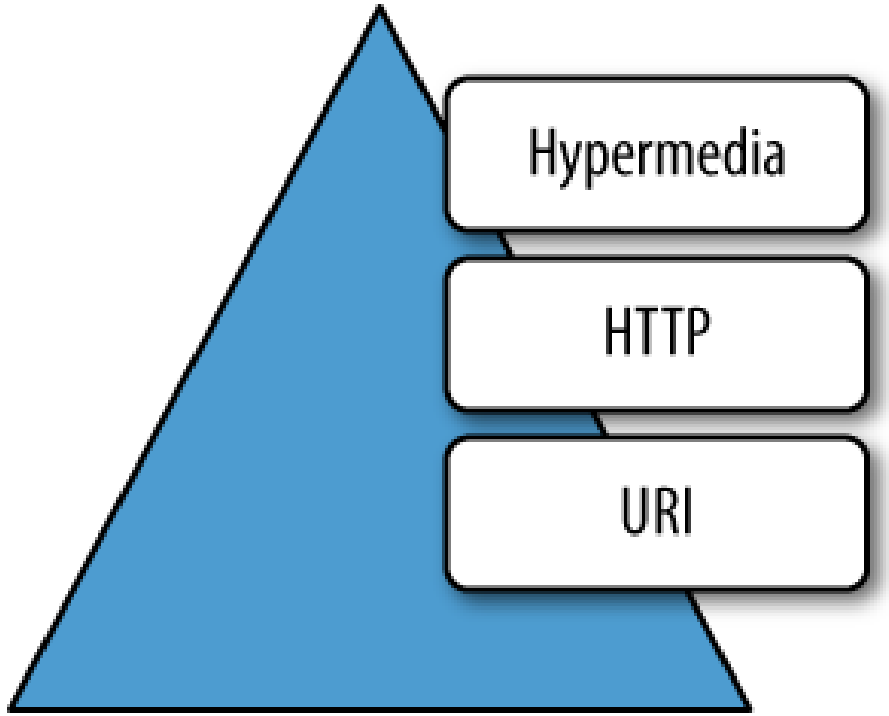
Advanced RESTful APIs

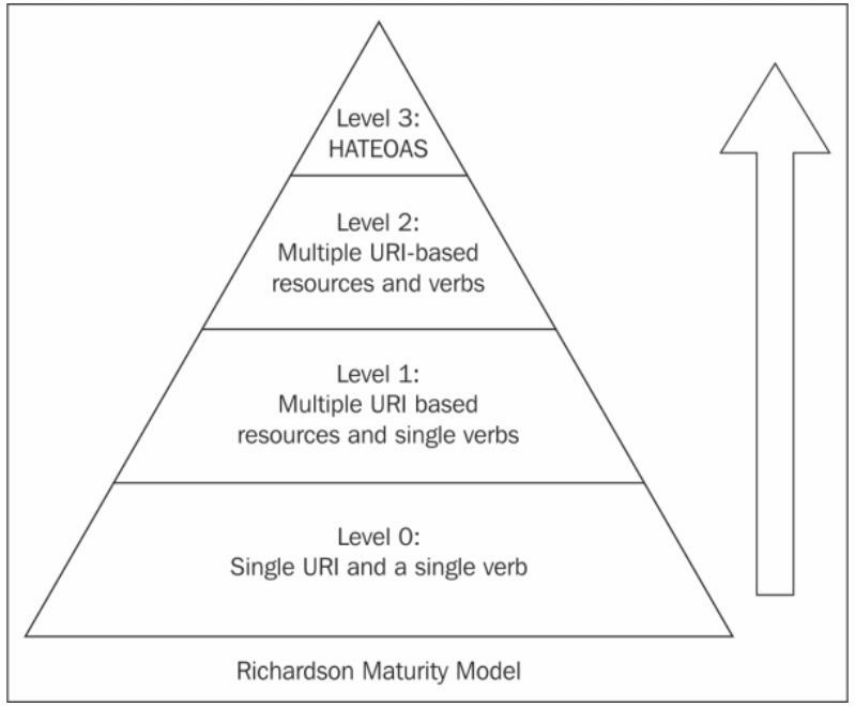
[With Microservice Concepts]

Richardson Maturity Model

Leonard Richardson analyzed a hundred different web service designs and divided them into four categories based on how much they are REST compliant.

This model of division of REST services to identify their maturity level – is called Richardson Maturity Model.





Level Zero

Level zero of maturity does not make use of any of URI, HTTP Methods, and HATEOAS capabilities.

These services have a single URI and use a single HTTP method (typically POST). For example, most Web Services (WS-*)-based services use a single URI to identify an endpoint, and HTTP POST to transfer SOAP-based payloads, effectively ignoring the rest of the HTTP verbs.

Similarly, XML-RPC based services which send data as Plain Old XML (POX). These are the most primitive way of building SOA applications with a single POST method and using XML to communicate between services.

Level One

Level one of maturity makes use of URIs out of URI, HTTP Methods, and HATEOAS.

These services employ many URIs but only a single HTTP verb – generally HTTP POST. They give each individual resource in their universe a URI. Every resource is separately identified by a unique URI – and that makes them better than level zero.

Level Two

Level two of maturity makes use of URIs and HTTP out of URI, HTTP Methods, and HATEOAS.

Level two services host numerous URI-addressable resources. Such services support several of the HTTP verbs on each exposed resource – Create, Read, Update and Delete (CRUD) services. Here the state of resources, typically representing business entities, can be manipulated over the network.

Here service designer expects people to put some effort into mastering the APIs – generally by reading the supplied documentation.

Level 2 is the good use-case of REST principles, which advocate using different verbs based on the HTTP request methods and the system can have multiple resources.

Level Three

Level three of maturity makes use of all three i.e. URIs and HTTP and HATEOAS.

This is the most mature level of Richardson’s model which encourages easy discoverability and makes it easy for the responses to be self-explanatory by using HATEOAS.

The service leads consumers through a trail of resources, causing application state transitions as a result.

HATEOAS

Hypermedia is an extension to what is known as hypertext, or the ability to open new Web pages by clicking text links on a Web browser.

Hypermedia extends upon this by allowing the user to click images, movies, graphics and other media apart from text to create a nonlinear network of information

HATEOAS (Hypermedia as the Engine of Application State) is a constraint of the REST application architecture.

A hypermedia-driven site provides information to navigate the site's REST interfaces dynamically by including hypermedia links with the responses.

Spring HATEOAS provides some APIs to ease creating REST representations that follow the HATEOAS principle when working with Spring and especially Spring MVC.

<dependency>
<groupId>org.springframework.hateoas</groupId>
<artifactId>spring-hateoas</artifactId>
<version>0.23.0.RELEASE</version> </dependency>

A HATEOAS-based response would look like this:

```
{
   "name": "Alice",
   "links": [ {
      "rel": "self",
      "href": http://localhost:8080/customer/1     } ] }
```

This response not only has the person's name, but includes the self-linking URL where that person is located.

rel means relationship. In this case, it's a self-referencing hyperlink.

More complex systems might include other relationships. For example, an order might have a "rel":"customer" relationship, linking the order to its customer (/customer/order/...)

href is a complete URL that uniquely defines the resource.

Representation models

To easily create hypermedia enriched representations, Spring HATEOAS provides a set of classes with RepresentationModel at their root. It’s basically a container for a collection of Links and has convenient methods to add those to the model.

RepresentationModel
EntityModel
CollectionModel
PagedModel

public class **EntityModel**<T>
extends RepresentationModel<EntityModel<T>>
Base class for DTOs to collect links.

public class **CollectionModel**<T>
extends RepresentationModel<CollectionModel<T>>
implements Iterable<T>
General helper to easily create a wrapper for a collection of entities.

public class **PagedModel**<T>
extends CollectionModel<T>
DTO to implement binding response representations of pageable collections.

Server-side support [Building links in Spring MVC]

Spring HATEOAS provides a WebMvcLinkBuilder that lets you create links by pointing to controller classes. The following example shows how to do so:

```
import static org.sfw.hateoas.server.mvc.WebMvcLinkBuilder.*;

Link link = linkTo(PersonController.class).withRel("people");
```

The WebMvcLinkBuilder uses Spring’s ServletUriComponentsBuilder under the hood to obtain the basic URI information from the current request

Swagger

Mostly front-end and back-end components often separate a web application.

Usually, we expose APIs as a back-end component for the front-end component or third-party app integrations.

In such a scenario, it is essential to have proper specifications for the back-end APIs.

At the same time, the API documentation should be informative, readable, and easy to follow.

Moreover, reference documentation should simultaneously describe every change in the API.

Swagger is an open-source software framework backed by a large ecosystem of tools that helps developers design, build, document, and consume RESTful web services. While most users identify Swagger by the Swagger UI tool, the Swagger toolset includes support for automated documentation, code generation, and test-case generation.

SpringFox

Automated JSON API documentation for API's built with Spring

Springfox implementation of the Swagger specification.

To add it to our Maven project, we need a dependency in the pom.xml file.

```
<dependency>
   <groupId>io.springfox</groupId>
   <artifactId>springfox-swagger2</artifactId>
   <version>2.9.2</version>
</dependency
```

Integrating Swagger 2 into the Project

The configuration of Swagger mainly around the Docket bean.

```
@Configuration
@EnableSwagger2
public class SpringFoxConfig {
   @Bean
   public Docket api() {
      return new Docket(DocumentationType.SWAGGER_2)
        .select()
        .apis(RequestHandlerSelectors.any())
        .paths(PathSelectors.any())
        .build();
   }
}
```

Swagger UI

Swagger UI is a built-in solution which makes user interaction with the Swagger-generated
API documentation much easier.

To use Swagger UI, one additional Maven dependency is required:

```
<dependency>
   <groupId>io.springfox</groupId>
   <artifactId>springfox-swagger-ui</artifactId>
   <version>2.9.2</version>
</dependency>
```

We can test it in our browser by visiting http://localhost:8080/your-app-root/swagger-ui.html

Springfox Plugin

In order to add specific features to the API specifications, we can create a Springfox plugin. A plugin can offer various features, from enriching the models and properties to the custom API listings and defaults.

Springfox supports the plugin creation through its spi module. The spi module provides a few interfaces like the ModelBuilderPlugin, ModelPropertyBuilderPlugin, and ApiListingBuilderPlugin that act as an extensibility hook to implement a custom plugin.

To demonstrate the capabilities, let's create a plugin to enrich the email property of the User model.

```
@Component
@Order(Validators.BEAN_VALIDATOR_PLUGIN_ORDER)
public class EmailAnnotationPlugin implements
ModelPropertyBuilderPlugin {
   @Override
   public boolean supports(DocumentationType delimiter) {
      return true;
   } }

@Override
public void apply(ModelPropertyContext context) {
   Optional<Email> email = annotationFromBean(context,
Email.class);
   if (email.isPresent()) {
      context.getBuilder().pattern(email.get().regexp());
      context.getBuilder().example("email@email.com");
   } }
```

So, the API specifications will show the pattern and example values of the property annotated with the @Email annotation.

Next, we'll add the @Email annotation to the User entity:

```
@Entity
public class User {
   //...
    @Email(regexp=".@.\\..*", message = "Email should be valid")
   private String email;
}
```

To enable the EmailAnnotationPlugin in the SpringFoxConfig class by registering as a bean:

```
@Import({BeanValidatorPluginsConfiguration.class})
public class SpringFoxConfig {
```

Filtering API for Swagger’s Response

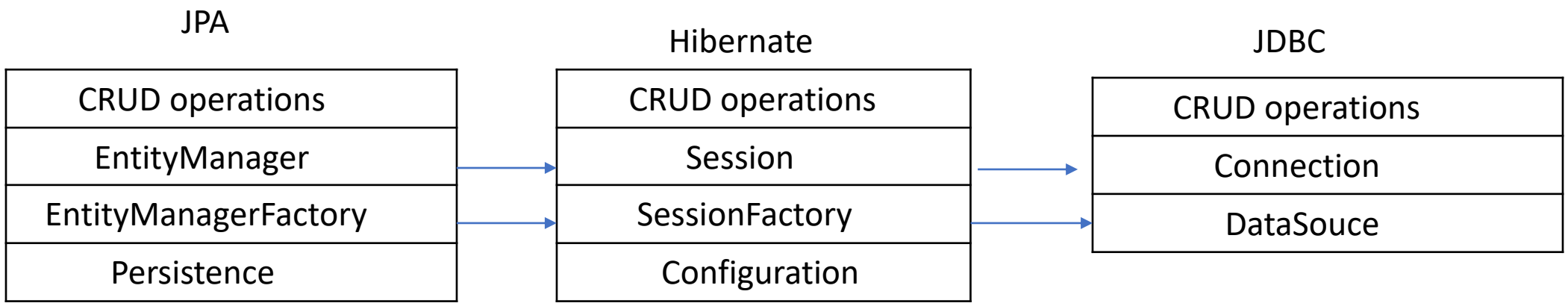
We can also restrict Swagger’s response by passing parameters to the apis() and paths() methods of the Docket class.

RequestHandlerSelectors allows using the any or none predicates, but can also be used to filter the API according to the base package, class annotation, and method annotations.

PathSelectors provides additional filtering with predicates which scan the request paths of your application. You can use any(), none(), regex(), or ant().

Data Access Layer

JPA / Hibernate / JDBC



Persistence.xml

Javax.persistence.provide = Hibernate

+

Hibernate-jpa.jar

```
@Entity
Class User
{
String name;
....
getName(){}

setName(){}
}
```

```
interface UserDao
{

List<User>  getUsers();

User getUser(long id);

void updateUser(User user);

void deleteUser(User user);

}
```

```
@Repository
Class UserDaoJpa  implements UserDao
{

@PersistenceContext
EntityManager  manager;


List<User>  getUsers()
{   return manager.createQuery("from User"); }

User getUser(long id)
{ return  manager.find(User.class, id); }

void updateUser(User user)  { ... }

void deleteUser(User user)   { ... }

}
```

Spring Data – REST

Spring Data

Spring Data's mission is to provide a familiar and consistent, Spring-based programming model for data access while still retaining the special traits of the underlying data store.

It makes it easy to use data access technologies, relational and non-relational databases, map-reduce frameworks, and cloud-based data services

Features

- Powerful repository and custom object-mapping abstractions
- Dynamic query derivation from repository method names
- Support for transparent auditing (created, last changed)
- Possibility to integrate custom repository code
- Easy Spring integration via JavaConfig and custom XML namespaces
- Advanced integration with Spring MVC controllers
- Experimental support for cross-store persistence

Main modules

Spring Data Commons - Core Spring concepts underpinning every Spring Data project.

Spring Data JPA - Makes it easy to implement JPA-based repositories.

Spring Data KeyValue - Map-based repositories and SPIs to easily build a Spring Data module for key-value stores.

Spring Data LDAP - Provides Spring Data repository support for Spring LDAP.

Spring Data MongoDB - Spring based, object-document support and repositories for MongoDB.

Spring Data REST - Exports Spring Data repositories as hypermedia-driven RESTful resources.

Spring Data for Apache Cassandra - Spring Data module for Apache Cassandra.

org.springframework.data.repository

Interface Repository<T,ID>

Type Parameters:
T - the domain type the repository manages
ID - the type of the id of the entity the repository manages

All Known Subinterfaces:
CrudRepository<T,ID>, PagingAndSortingRepository<T,ID>,
ReactiveCrudRepository<T,ID>, ReactiveSortingRepository<T,ID>,
RevisionRepository<T,ID,N>, RxJava2CrudRepository<T,ID>,
RxJava2SortingRepository<T,ID>

CrudRepository Methods:

count()
Returns the number of entities available.

delete(T entity)
Deletes a given entity.

deleteAll(Iterable<? extends T> entities)
Deletes the given entities.

deleteById(ID id)
Deletes the entity with the given id.

save(S entity)
Saves a given entity.

saveAll(Iterable<S> entities)
Saves all given entities.

org.springframework.data.repository
Interface **PagingAndSortingRepository**<T,ID>

public interface PagingAndSortingRepository<T,ID>
extends CrudRepository<T,ID>

Extension of CrudRepository to provide additional methods to retrieve entities using the pagination and sorting abstraction.

PagingAndSortingRepository methods are:

findAll(Pageable pageable)
Returns a Page of entities meeting the paging restriction provided in the Pageable object.

findAll(Sort sort)
Returns all entities sorted by the given options.

JpaRepository - which extends PagingAndSortingRepository and, in turn, the CrudRepository.

It provides JPA related methods such as flushing the persistence context and delete records in a batch.

org.springframework.data.jpa.repository
Interface JpaRepository<T,ID>

All Superinterfaces:
CrudRepository<T,ID>, PagingAndSortingRepository<T,ID>,
QueryByExampleExecutor<T>, Repository<T,ID>

All Known Implementing Classes:
QuerydslJpaRepository, SimpleJpaRepository

Methods are :

deleteAllInBatch()
Deletes all entities in a batch call.

deleteInBatch(Iterable<T> entities)
Deletes the given entities in a batch which means it will create a single Query.

flush()
Flushes all pending changes to the database.

....

Methods inherited from interface
org.springframework.data.repository.PagingAndSortingRepository

- ❑ findAll

Methods inherited from interface
org.springframework.data.repository.CrudRepository

- ❑ count, delete, deleteAll, deleteAll, deleteById, existsById, findById, save

Methods inherited from interface
org.springframework.data.repository.query.QueryByExampleExecutor

- ❑ count, exists, findAll, findOne

public @interface **RepositoryRestResource**
Annotate a Repository with this to customize export mapping and rels.

Simplifies building hypermedia-driven REST web services on top of Spring Data repositories.

Examples

The following code represents a Customer object.

```
class Customer
{
   String name;
}
```

A simple JSON presentation is traditionally rendered as:

```
{
   "name" : “Great"
}
```

The customer data is there, but the data contains nothing about its relevant links.

A HATEOAS-based response would look like this:

```
{
   "name": “Great",
   "links": [ {
      "rel": "self",
      "href": http://localhost:8080/customer/1     } ]
}
```

This response not only has the person's name, but includes the self-linking URL where that person is located.

rel means relationship. In this case, it's a self-referencing hyperlink.

More complex systems might include other relationships. For example, an order might have a "rel":"customer" relationship, linking the order to its customer.

href is a complete URL that uniquely defines the resource.

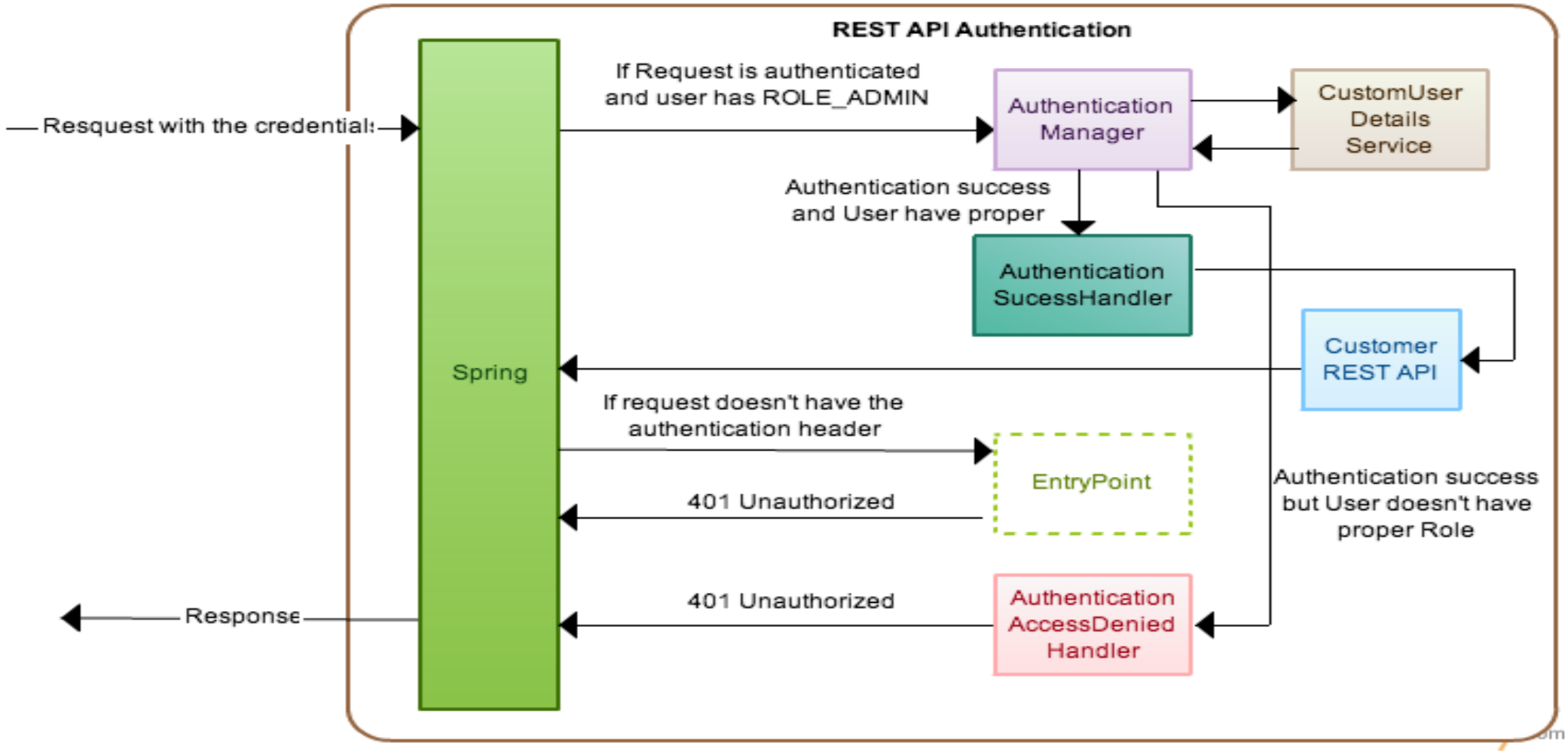
Note : XML is also accepted as a standard response format

SPRING security

The two dimensions of System Security

- Authentication
  - Identifying the user as an authorized user
  - Limiting the time the user can operate in the system before re-identifying himself
  - Securing that all requests for the duration of the session come from the identified user
- Authorization
  - Constraining the operations the logged in user can perform in accordance with his assigned roles

The SPRING Security Model



In web.xml

```
<filter>
<filter-name>springSecurityFilterChain</filter-name>
<filter-class>
org.springframework.web.filter.DelegatingFilterProxy
</filter-class>
</filter>


<filter-mapping>
<filter-name>springSecurityFilterChain</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

Java Configuration – Replaces web.xml

```
public class SecurityWebInitializer
        extends AbstractSecurityWebApplicationInitializer
{
// optionally override methods
}
```

Java Configuration – WebSecurityConfig

```
@Configuration
@EnableWebMvcSecurity
public class WebSecurityConfig
        extends WebSecurityConfigurerAdapter
{
...
}
```

Spring Security Configuration

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

        @Autowired
        public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
          auth.inMemoryAuthentication().withUser(“test").password("123456").roles("USER");
          auth.inMemoryAuthentication().withUser("admin").password("123456").roles("ADMIN");
          auth.inMemoryAuthentication().withUser("dba").password("123456").roles("DBA");
        }

        @Override
        protected void configure(HttpSecurity http) throws Exception {

          http.authorizeRequests()
                  .antMatchers("/admin/**").access("hasRole('ROLE_ADMIN')")
                  .antMatchers("/dba/**").access("hasRole('ROLE_ADMIN') or hasRole('ROLE_DBA')")
                  .and().formLogin();

        }
}
```

The equivalent of the Spring Security xml file :

```
<http auto-config="true">
        <intercept-url pattern="/admin**" access="ROLE_ADMIN" />
        <intercept-url pattern="/dba**" access="ROLE_ADMIN,ROLE_DBA" />
        <form-login />
</http>

<authentication-manager>
  <authentication-provider>
    <user-service>
        <user name=“test" password="123456" authorities="ROLE_USER" />
        <user name="admin" password="123456" authorities="ROLE_ADMIN" />
        <user name="dba" password="123456" authorities="ROLE_DBA" />
    </user-service>
  </authentication-provider>
</authentication-manager>
```

Example:

```
protected void configure(HttpSecurity http) throws Exception
{
    http.authorizeRequests()
      .anyRequest().authenticated()
      .and().httpBasic();
}
```

The above default configuration makes sure any request to the application is authenticated with form based login or HTTP basic authentication.

Also, it is exactly similar to the following XML configuration:

```
<http>
    <intercept-url pattern="/**" access="authenticated"/>
    <form-login />
    <http-basic />
</http>
```

org.springframework.util

Class AntPathMatcher

java.lang.Object
    org.springframework.util.AntPathMatcher

All Implemented Interfaces:

`PathMatcher`

```
public class AntPathMatcher
extends java.lang.Object
implements PathMatcher
```

`PathMatcher` implementation for Ant-style path patterns.

Part of this mapping code has been kindly borrowed from Apache Ant.

The mapping matches URLs using the following rules:

- `?` matches one character
- `*` matches zero or more characters
- `**` matches zero or more *directories* in a path
- `{spring:[a-z]+}` matches the regexp `[a-z]+` as a path variable named "spring"

Examples

com/t?st.jsp — matches com/test.jsp but also com/tast.jsp or com/txst.jsp

com/*.jsp — matches all .jsp files in the com directory

com/**/test.jsp — matches all test.jsp files underneath the com path

org/springframework/**/*.jsp — matches all .jsp files underneath the org/springframework path

org/**/servlet/bla.jsp — matches org/springframework/servlet/bla.jsp but also org/springframework/testing/servlet/bla.jsp and org/servlet/bla.jsp

The Spring 5.0 release added a very easy to use URI variable syntax: {*foo} to capture any number of path segments at the end of the pattern.

```
@GetMapping("/spring5/{*id}")
public String URIVariableHandler(@PathVariable String id) {
    return id;
}
```

Ex:

```
uri("/spring5/sample/tutorial")
uri("/spring5/sample")
```

```
protected void configure(HttpSecurity http) throws Exception {
        http
                .authorizeRequests()
                        .anyRequest().authenticated()
                        .and()
                .formLogin();
                        }
```

The default configuration above:

- ❑ Ensures that any request to our application requires the user to be authenticated
- ❑ Allows users to authenticate with form based login

It is similar the XML Namespace configuration:

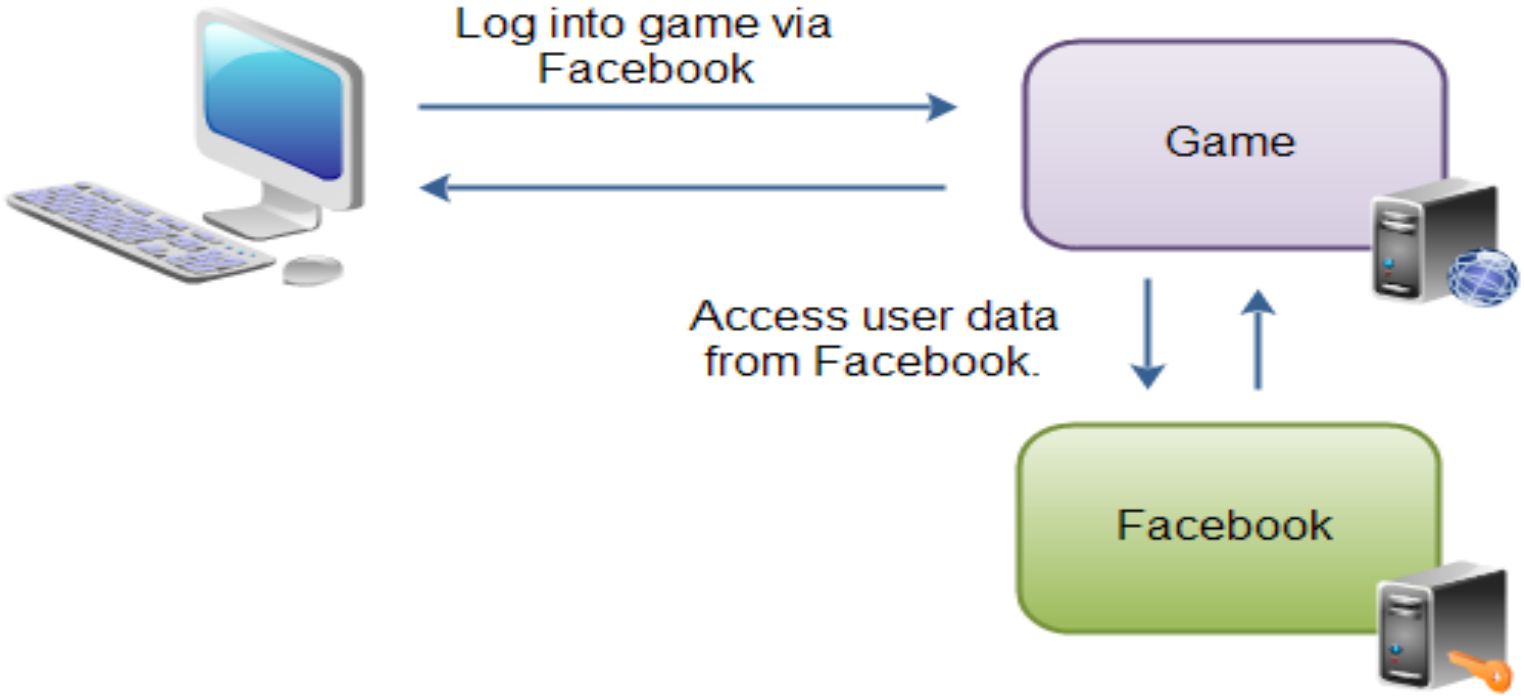
```
<http>
        <intercept-url pattern="/**" access="authenticated"/>
        <form-login />
</http>
```

OAUTH 2.0 OVERVIEW

OAuth 2.0 is an open authorization protocol specification defined by IETF OAuth WG (Working Group) which enables applications to access each other’s data.

The prime focus of this protocol is to define a standard where an application, say gaming site, can access the user’s data maintained by another application like facebook, google or other resource server.

OAuth 2.0 is a replacement for OAuth 1.0, which was more complicated. OAuth 1.0 involved certificates etc. OAuth 2.0 is more simple. It requires no certificates at all, just SSL / TLS.



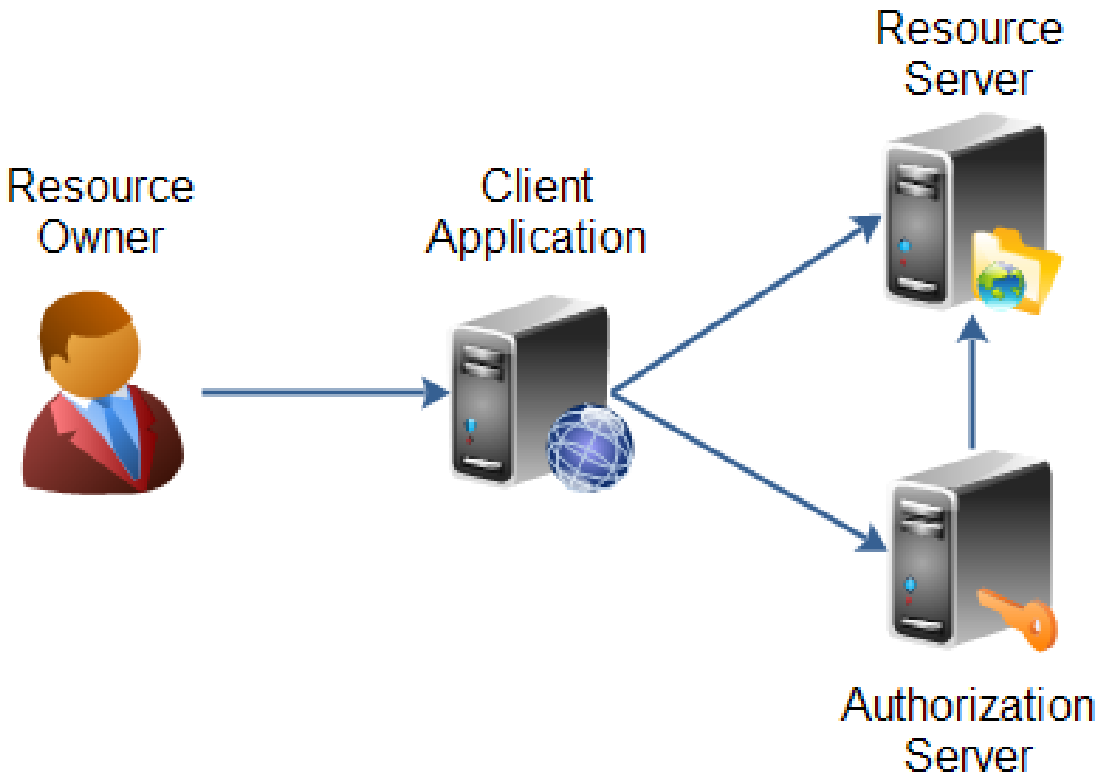
The user accesses the game web application.

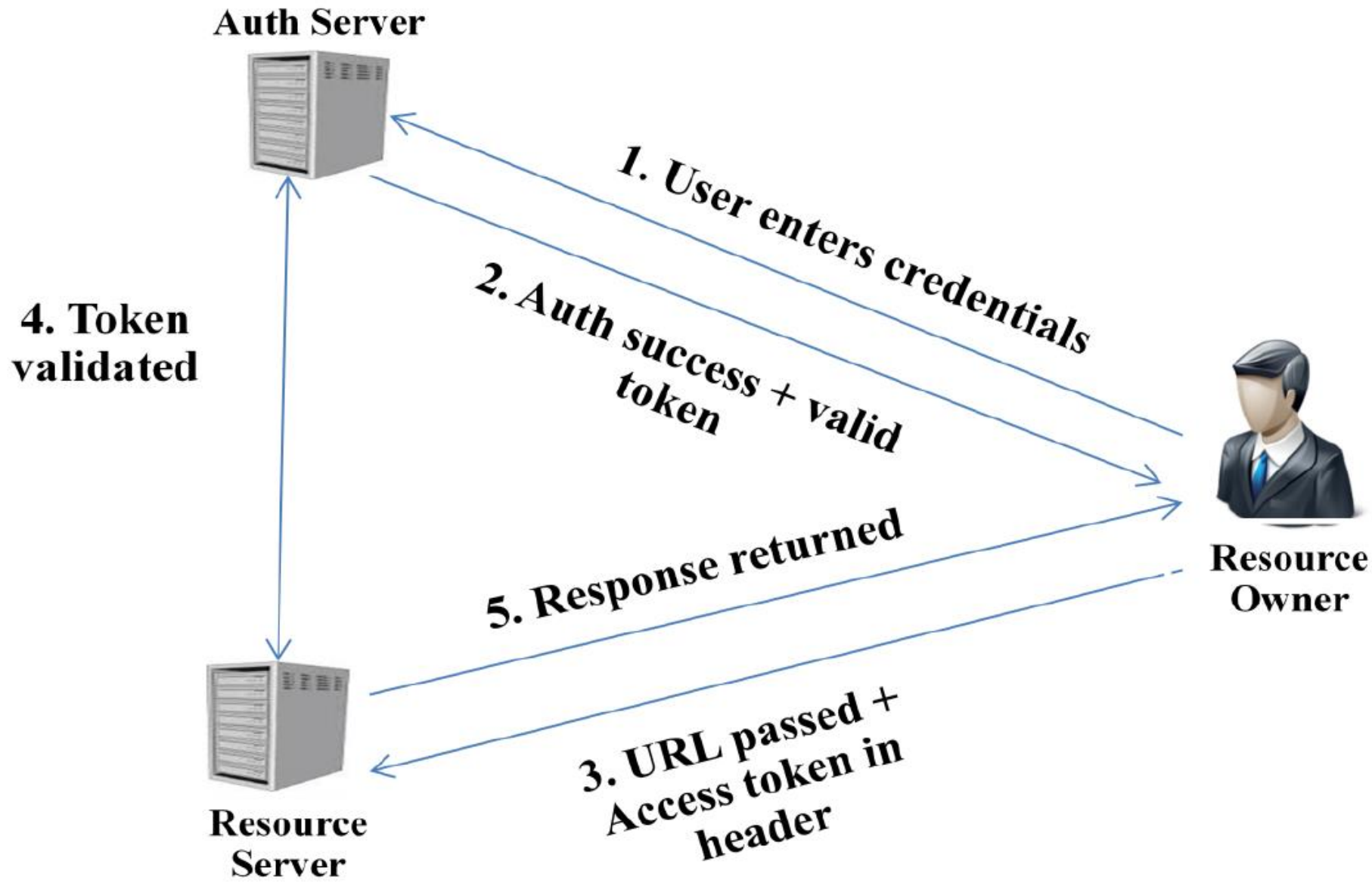
The game web application asks the user to login to the game via Facebook.

The user logs into Facebook, and is sent back to the game. The game can now access the users data in Facebook, and call functions in Facebook on behalf of the user (e.g. posting status updates).

OAuth 2.0 defines the following roles of users and applications:

- Resource Owner
- Resource Server
- Client Application
- Authorization Server


Resource
Server
Resource
Owner
Client
Application
Authorization
Server


Auth Server
1. User enters credentials
2. Auth success + valid token
4. Token validated
5. Response returned
3. URL passed + Access token in header
Resource Owner
Resource Server

- The resource owner is the person or application that owns the data that is to be shared.

  For instance, a user on Facebook or Google could be a resource owner. The resource they own is their data. The resource owner could also be an application. The OAuth 2.0 specification mentions both possibilities.

- The resource server is the server hosting the resources. For instance, Facebook or Google is a resource server (or has a resource server).

- The client application is the application requesting access to the resources stored on the resource server. The resources, which are owned by the resource owner. A client application could be a game requesting access to a users Facebook account.

- The authorization server is the server authorizing the client app to access the resources of the resource owner. The authorization server and the resource server can be the same server, but it doesn't have to.

Steps involved in User Authentication

1. User enters credentials which are passed over to Authorization Server in Http Authentication header in encrypted form. The communication channel is secured with SSL.

2. Authorization server authenticates the user with the credentials passed and generates a token for limited time and finally returns it in response.

3. The client application calls API to resource server, passing the token in http header or as a query string.

4. Resource server extracts the token and authorizes it with Authorization server.

5. Once the authorization is successful, a valid response is sent to the caller.

Client ID, Client Secret and Redirect URI

Before a client application can request access to resources on a resource server, the client application must first register with the authorization server associated with the resource server.

The registration is typically a one-time task. Once registered, the registration remains valid, unless the client app registration is revoked.

At registration the client application is assigned a client ID and a client secret (password) by the authorization server. The client ID and secret is unique to the client application on that authorization server.

Authorization Grant

The authorization grant is given to a client application by the resource owner, in cooperation with the authorization server associated with the resource server.

The OAuth 2.0 specification lists four different types of authorization grants. Each type has different security characteristics. The authorization grant types are:

- ❑ Authorization Code
- ❑ Implicit
- ❑ Resource Owner Password Credentials
- ❑ Client Credentials

Authorization Code

An authorization grant using an authorization code works like this

1) The resource owner (user) accesses the client application.

2) The client application tells the user to login to the client application via an authorization server (e.g. Facebook, Twitter, Google etc.).

3) To login via the authorization server, the user is redirected to the authorization server by the client application. The client application sends its client ID along to the authorization server, so the authorization server knows which application is trying to access the protected resources.

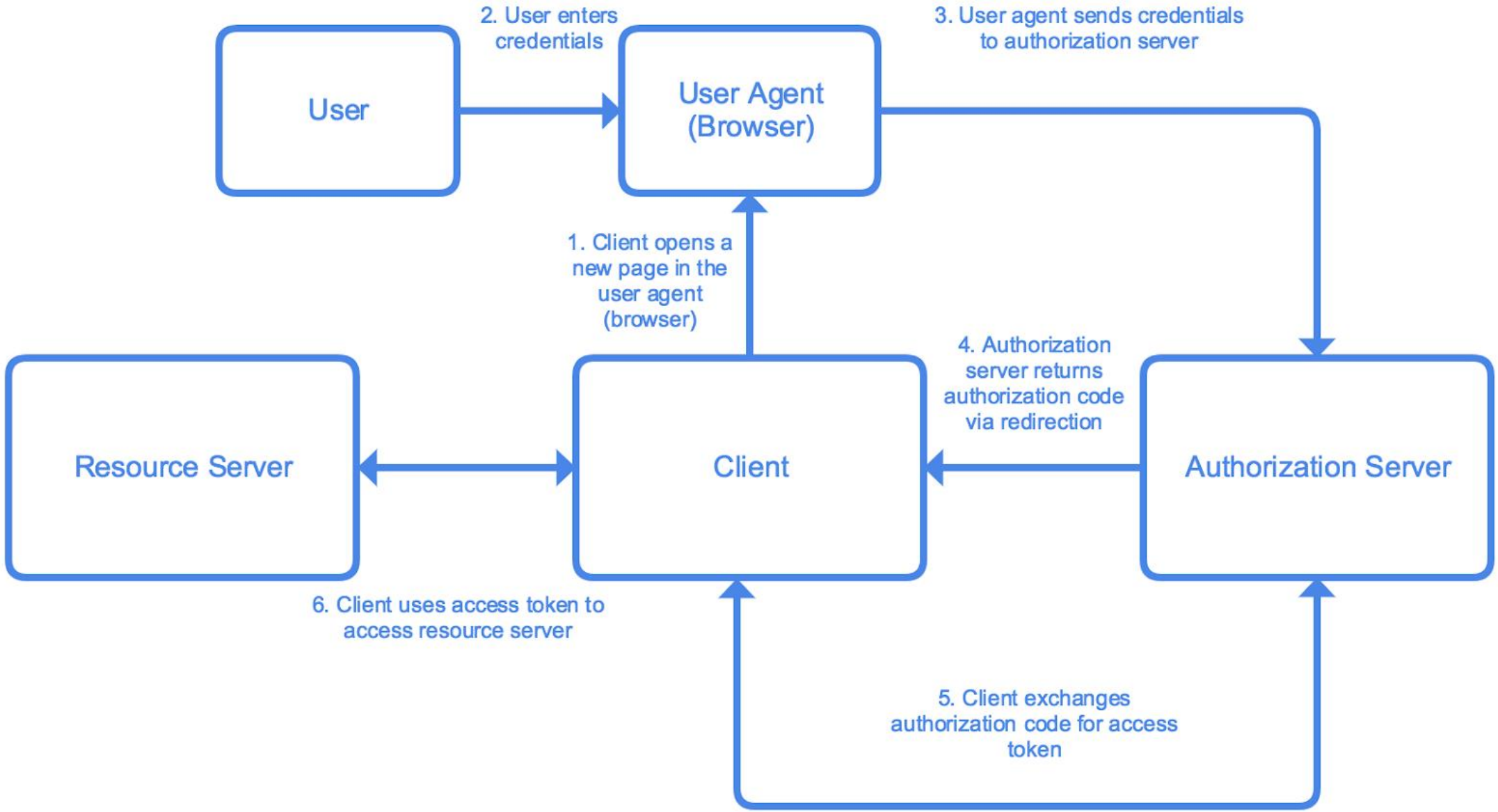
4) The user logs in via the authorization server. After successful login the user is asked if he wants to grant access to his resources to the client application. If the user accepts, the user is redirected back to the client application.

5) When redirected back to the client application, the authorization server sends the user to a specific redirect URI, which the client application has registered with the authorization server ahead of time. Along with the redirection, the authorization server sends an authorization code, representing the authorization.

6) When the redirect URI in the client application is accessed, the client application connects directly to the authorization server. The client application sends the authorization code along with its own client ID and and client secret.

7) If the authorization server can accept these values, the authorization server sends back an access token.

10) The client application can now use the access token to request resources from the resource server. The access token serves as both authentication of the client, resource owner (user) and authorization to access the resources.


User
2. User enters credentials
User Agent (Browser)
3. User agent sends credentials to authorization server
1. Client opens a new page in the user agent (browser)
Resource Server
Client
4. Authorization server returns authorization code via redirection
Authorization Server
6. Client uses access token to access resource server
5. Client exchanges authorization code for access token

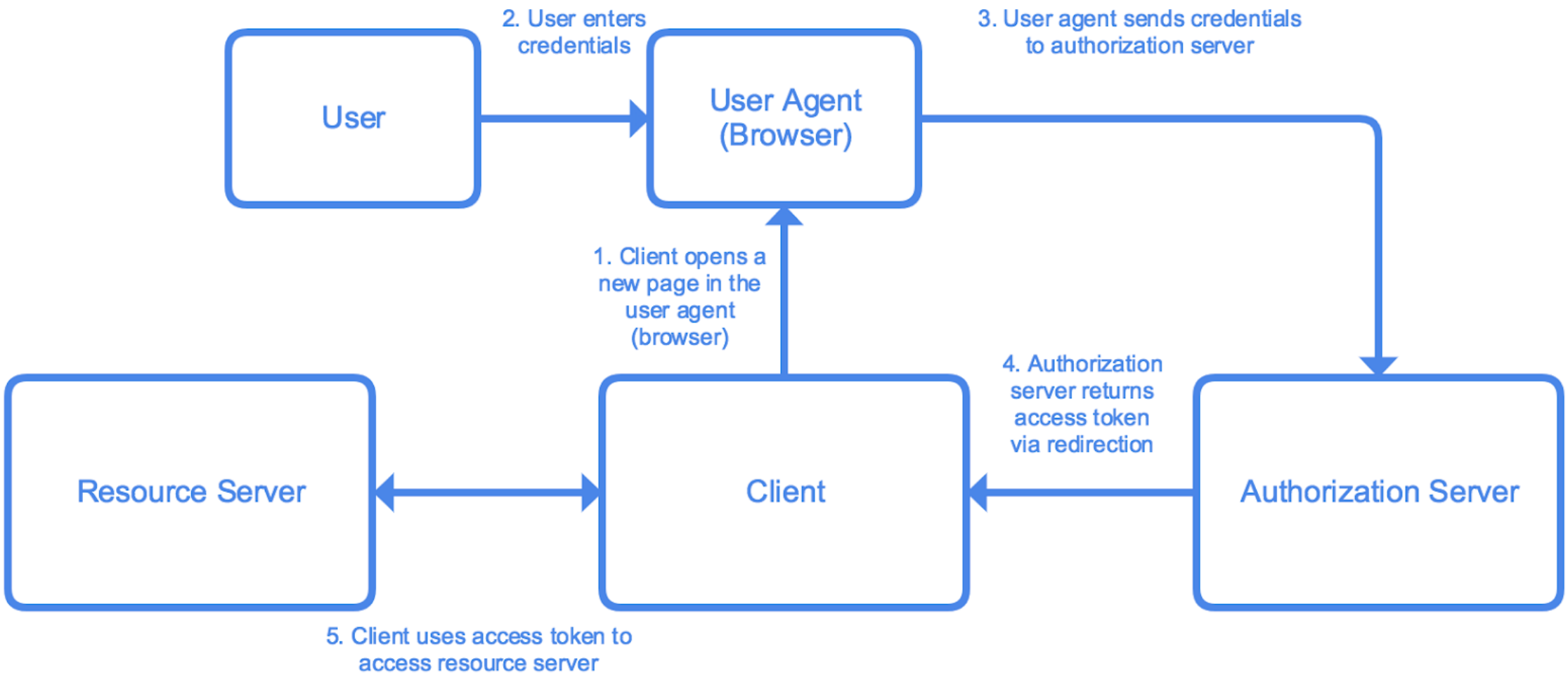
Implicit

An implicit authorization grant is similar to an authorization code grant, except the access token is returned to the client application already after the user has finished the authorization. The access token is thus returned when the user agent is redirected to the redirect URI.

This of course means that the access token is accessible in the user agent, or native application participating in the implicit authorization grant. The access token is not stored securely on a web server.

Furthermore, the client application can only send its client ID to the authorization server. If the client were to send its client secret too, the client secret would have to be stored in the user agent or native application too. That would make it vulnerable to hacking.

Implicit authorization grant is mostly used in a user agent or native client application. The user agent or native application would receive the access token from the authorization server.

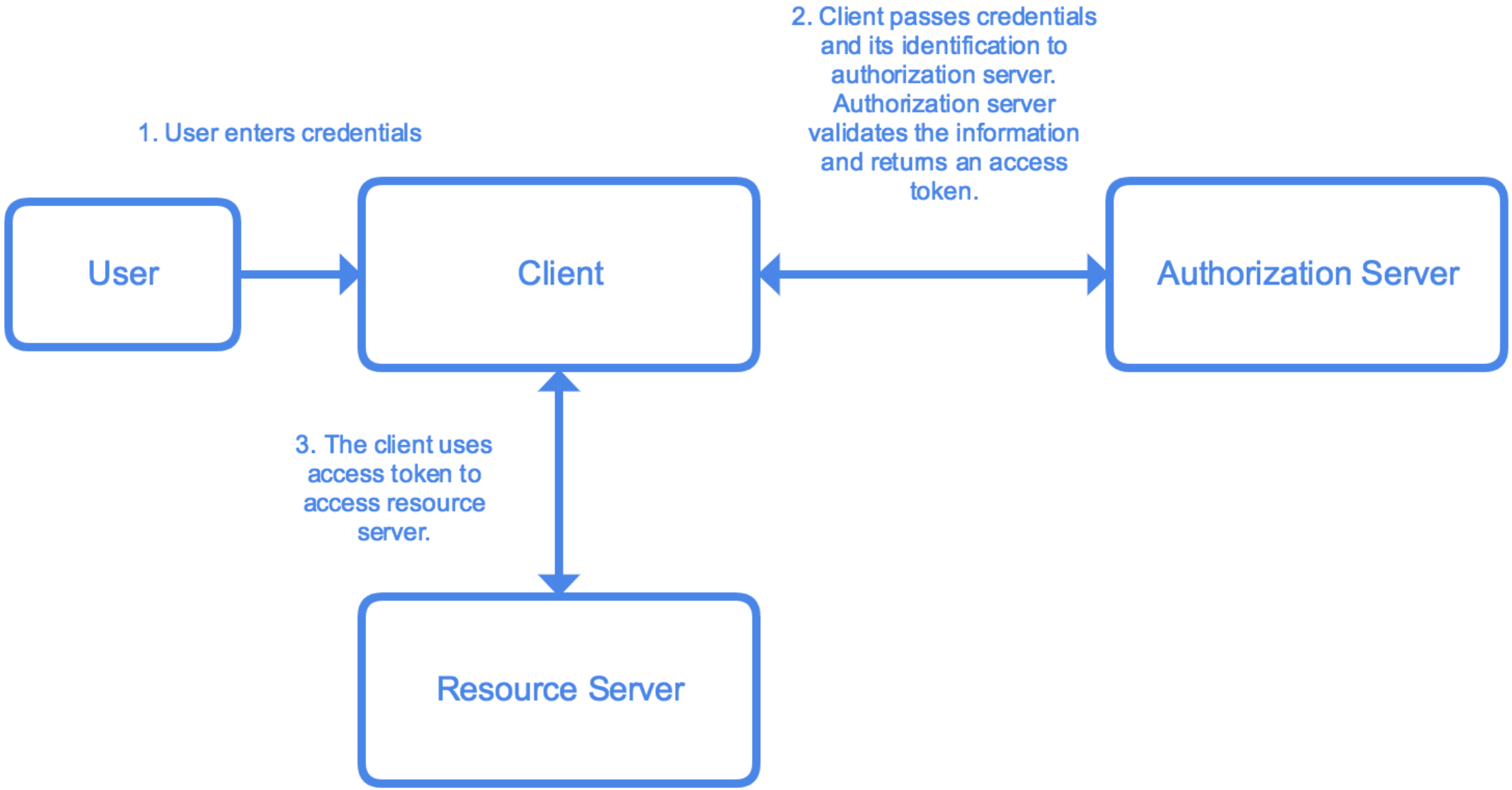

User
2. User enters credentials
User Agent (Browser)
3. User agent sends credentials to authorization server
1. Client opens a new page in the user agent (browser)
Resource Server
Client
4. Authorization server returns access token via redirection
Authorization Server
5. Client uses access token to access resource server

Resource Owner Password Credentials

The resource owner password credentials authorization grant method works by giving the client application access to the resource owners credentials. For instance, a user could type his Twitter user name and password (credentials) into the client application. The client application could then use the user name and password to access resources in Twitter.

Using the resource owner password credentials requires a lot of trust in the client application. You do not want to type your credentials into an application you suspect might abuse it.

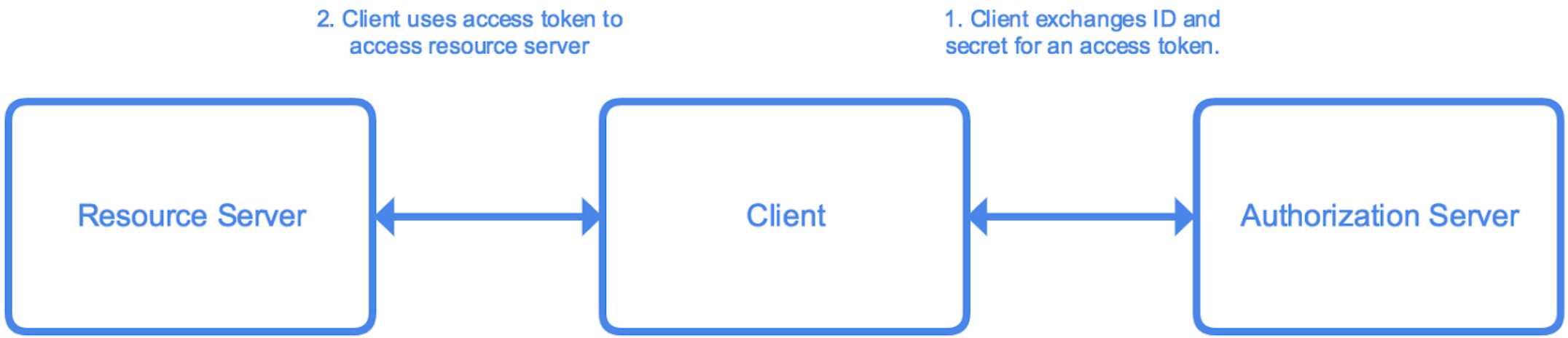
The resource owner password credentials would normally be used by user agent client applications, or native client applications.


1. User enters credentials
2. Client passes credentials and its identification to authorization server. Authorization server validates the information and returns an access token.
User
Client
Authorization Server
3. The client uses access token to access resource server.
Resource Server

Client Credentials

Client credential authorization is for the situations where the client application needs to access resources or call functions in the resource server, which are not related to a specific resource owner (e.g. user).

For instance, obtaining a list of venues from Foursquare. This does not necessary have anything to do with a specific Foursquare user.


2. Client uses access token to access resource server
1. Client exchanges ID and secret for an access token.
Resource Server
Client
Authorization Server

Implicit and Client Credentials are flows typically reserved for special types of clients. More specifically,

Client Type	Flow
Single-page Javascript Web Applications (for example, Google Fonts)	Implicit
Non-interactive programs for machine-to-machine communications (for example, background services and daemons)	Client Credentials

As for other clients, depending on their trustworthiness, they can use the following flows:

Client Type	Flow
Highly trusted apps (first-party apps)	Authorization Code or Resource Owner Password Credentials
Less trusted apps (third-party apps requesting access to your platform)	Authorization Code

OAuth 2.0 Authorization Code Requests and Responses

The authorization code grant consists of 2 requests and 2 responses in total.

An authorization request + response, and a token request + response.

Authorization Request

The authorization request is sent to the authorization endpoint to obtain an authorization code. Here are the parameters used in the request

response_type	Required. Must be set to code
client_id	Required. The client identifier as assigned by the authorization server, when the client was registered.
redirect_uri	Optional. The redirect URI registered by the client.
scope	Optional. The possible scope of the request.
state	Optional (recommended). Any client state that needs to be passed on to the client request URI.

Authorization Response

The authorization response contains the authorization code needed to obtain an access token.

code	Required. The authorization code.
state	Required, if present in request. The same value as sent by the client in the state parameter, if any.

Authorization Error Response

If an error occurs during authorization, two situations can occur.

The first is, that the client is not authenticated or recognized. For instance, a wrong redirect URI was sent in the request. In that case the authorization server must not redirect the resource owner to the redirect URI. Instead it should inform the resource owner of the error.

The second situation is that client is authenticated correctly, but that something else failed.

error	Required. Must be one of a set of predefined error codes. See the specification for the codes and their meaning.
error_description	Optional. A human-readable UTF-8 encoded text describing the error. Intended for a developer, not an end user.
error_uri	Optional. A URI pointing to a human-readable web page with information about the error.
state	Required, if present in authorization request. The same value as sent in the state parameter in the request.

Token Request

Once an authorization code is obtained, the client can use that code to obtain an access token. Here is the access token request parameters:

client_id	Required. The client application's id.
client_secret	Required. The client application's client secret .
grant_type	Required. Must be set to authorization_code .
code	Required. The authorization code received by the authorization server.
redirect_uri	Required, if the request URI was included in the authorization request. Must be identical then.

Token Response

The response to the access token request is a JSON string containing the access token plus some more information:

```
{ "access_token" : "...", "token_type" : "...", "expires_in" : "...",
"refresh_token" : "...", }
```

The access_token property is the access token as assigned by the authorization server.

The token_type property is a type of token assigned by the authorization server.

The expires_in property is a number of seconds after which the access token expires, and is no longer valid. Expiration of access tokens is optional.

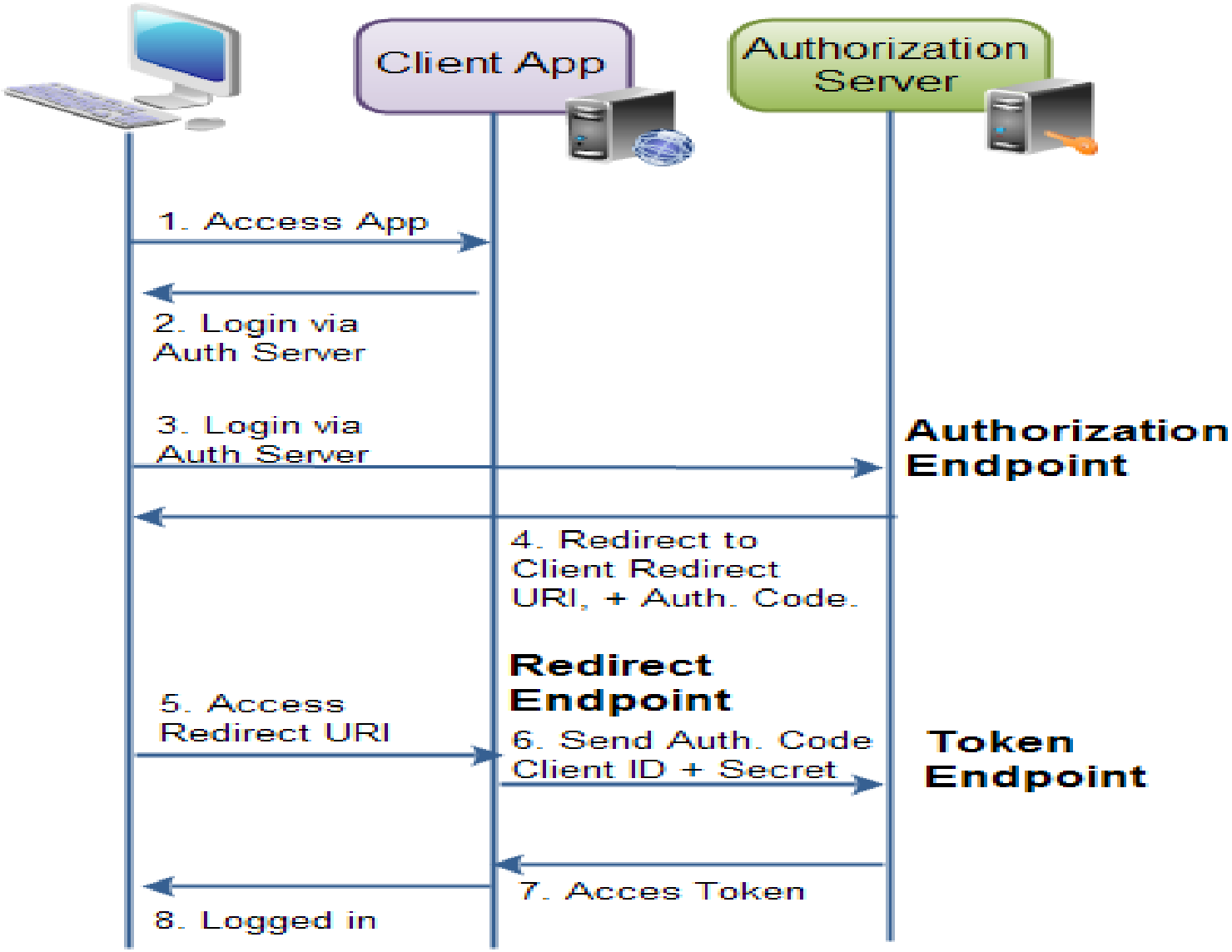
The refresh_token property contains a refresh token in case the access token can expire. The refresh token is used to obtain a new access token once the one returned in this response is no longer valid.

OAuth 2.0 Endpoints

OAuth 2.0 defines a set of endpoints. An endpoint is typically a URI on a web server. For instance, the address of a Java servlet, JSP page, PHP page, ASP.NET page etc.

The endpoints defined are:
•Authorization Endpoint
•Token Endpoint
•Redirection Endpoint

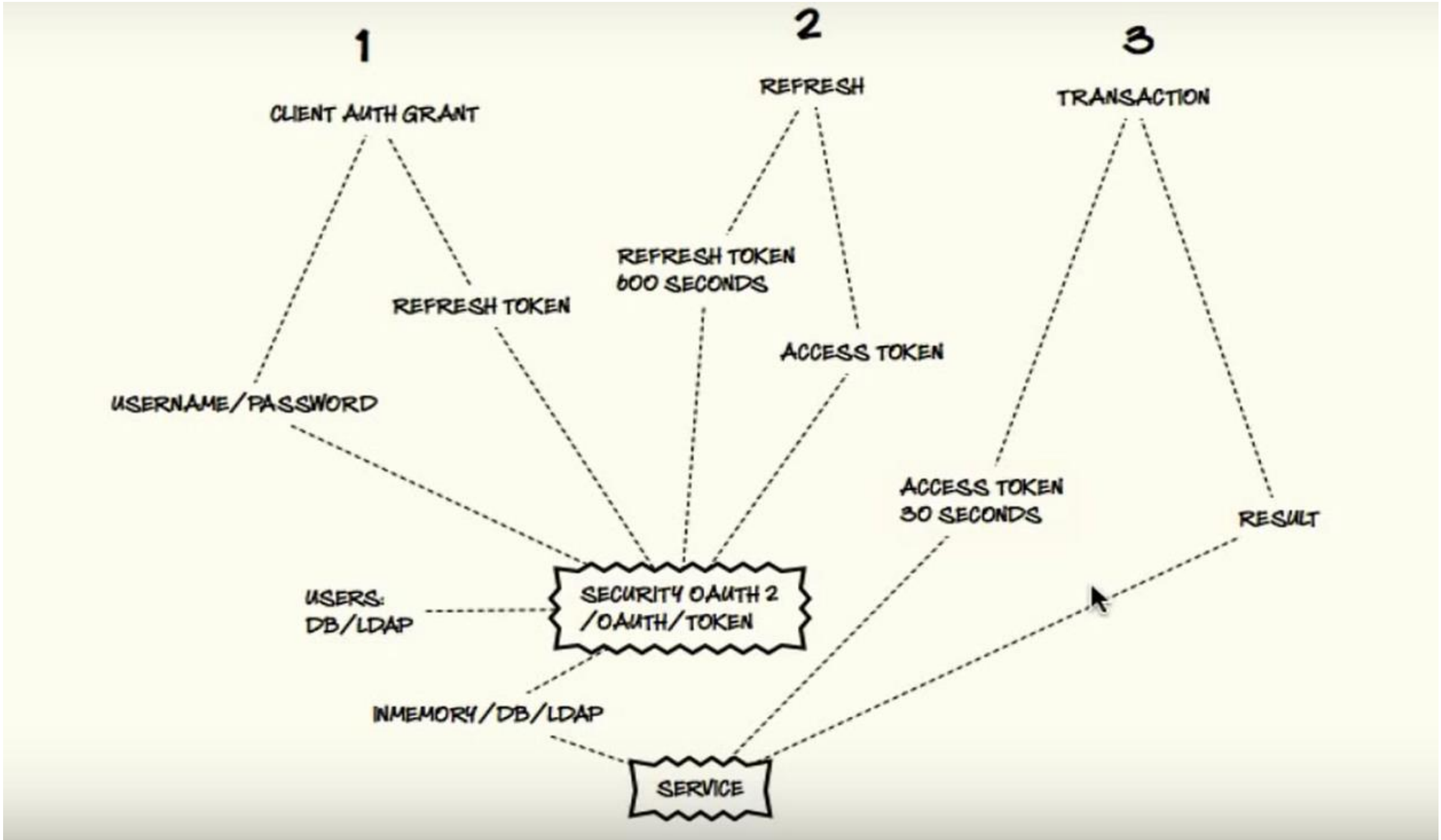
The authorization endpoint and token endpoint are both located on the authorization server. The redirection endpoint is located in the client application.


Client App
Authorization Server
1. Access App
2. Login via Auth Server
3. Login via Auth Server
Authorization Endpoint
4. Redirect to Client Redirect URI, + Auth. Code.
Redirect Endpoint
5. Access Redirect URI
6. Send Auth. Code Client ID + Secret
Token Endpoint
7. Acces Token
8. Logged in

Authorization Endpoint
The authorization endpoint is the endpoint on the authorization server where the resource owner logs in, and grants authorization to the client application.

Token Endpoint
The token endpoint is the endpoint on the authorization server where the client application exchanges the authorization code, client ID and client secret, for an access token.

Redirect Endpoint
The redirect endpoint is the endpoint in the client application where the resource owner is redirected to, after having granted authorization at the authorization endpoint.


1
2
3
CLIENT AUTH GRANT
REFRESH
TRANSACTION
REFRESH TOKEN
600 SECONDS
REFRESH TOKEN
ACCESS TOKEN
USERNAME/PASSWORD
ACCESS TOKEN
30 SECONDS
RESULT
USERS:
DB/LDAP
SECURITY OAUTH 2
/OAUTH/TOKEN
INMEMORY/DB/LDAP
SERVICE

A Refresh Token is a special kind of token that can be used to obtain a renewed access token

request of a new Access Token/Refresh Token pair

$ curl -X POST -H 'Authorization: Basic dGVzdGNsaWVudDpzZWNyZXQ=' -d 'grant_type=password&username=test&password=test' localhost:3000/oauth/token


```
{
   "token_type":"bearer",
   "access_token":"eyJ0eXAiOiJKV1QiLCJhQ3_DYKxxP2rFnD37Ip4",
   "expires_in":20,
   "refresh_token":"fdb8fdbecf1d03ce5e6125c067733c0d51de209c"
}
```


We can use the Refresh Token to get a new Access Token by using the token endpoint

curl -X POST -H 'Authorization: Basic dGVzdGNsaWVudDpzZWNyZXQ=' -d 'refresh_token=fdb8fdbecf1d03ce5e6125c067733c0d51de209c&grant_type=refresh_token' localhost:3000/oauth/token

grant_type=authorization_code

This is example of code request:
http://localhost:8081/spring-security-oauth-server/oauth/authorize?response_type=code&client_id=myclient&redirect_uri=http://localhost:8080/auth

And this is example of token request inside implementation of rest /auth

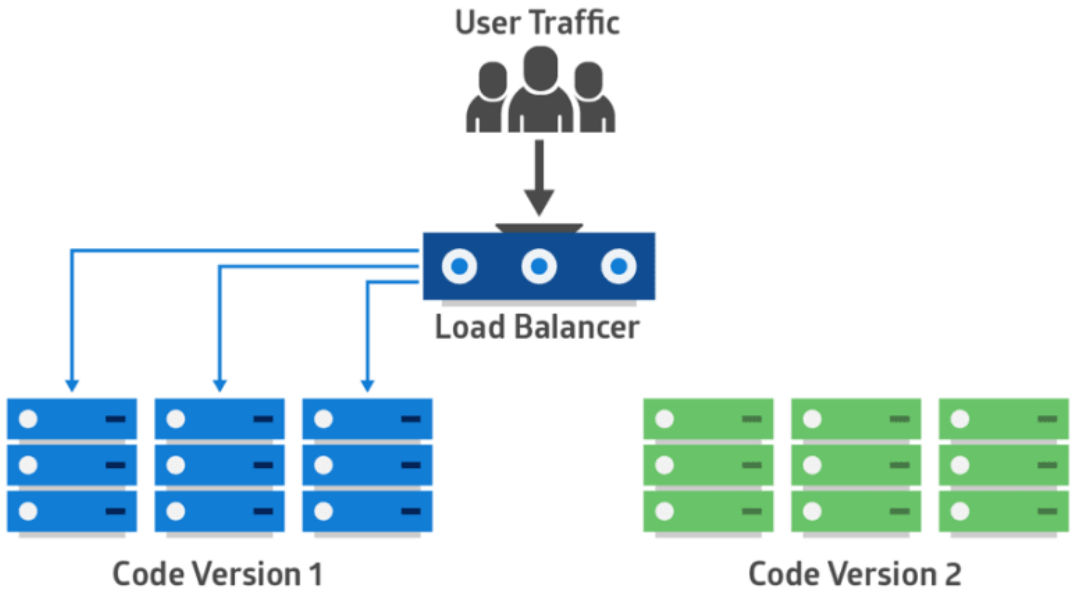
http://localhost:8081/spring-security-oauth-server/oauth/token?client_id=myclient&client_secret=123&grant_type=authorization_code&code=byZc1r&redirect_uri=http://localhost:8080/auth

Deployment Strategies

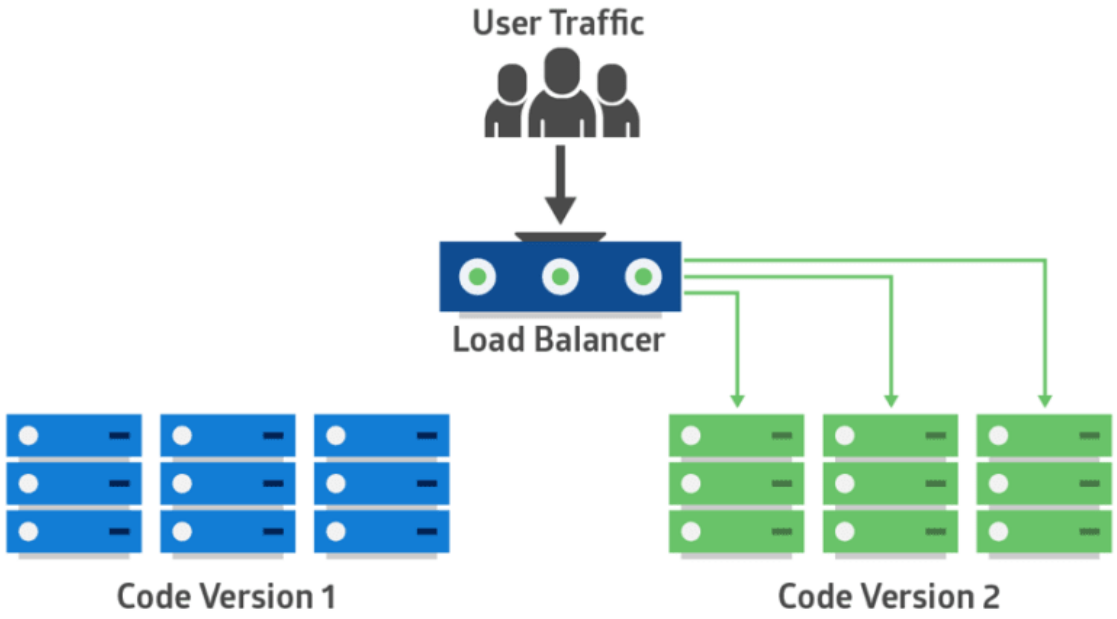
Blue-Green Deployment

This is fail-safe process. In this method, two identical production environments work in parallel.

One is the currently-running production environment receiving all user traffic (Blue). The other is a clone of it, but idle (Green). Both use the same database back-end and app configuration:



The new version of the application is deployed in the green environment and tested for functionality and performance. Once the testing results are successful, application traffic is routed from blue to green. Green then becomes the new production.



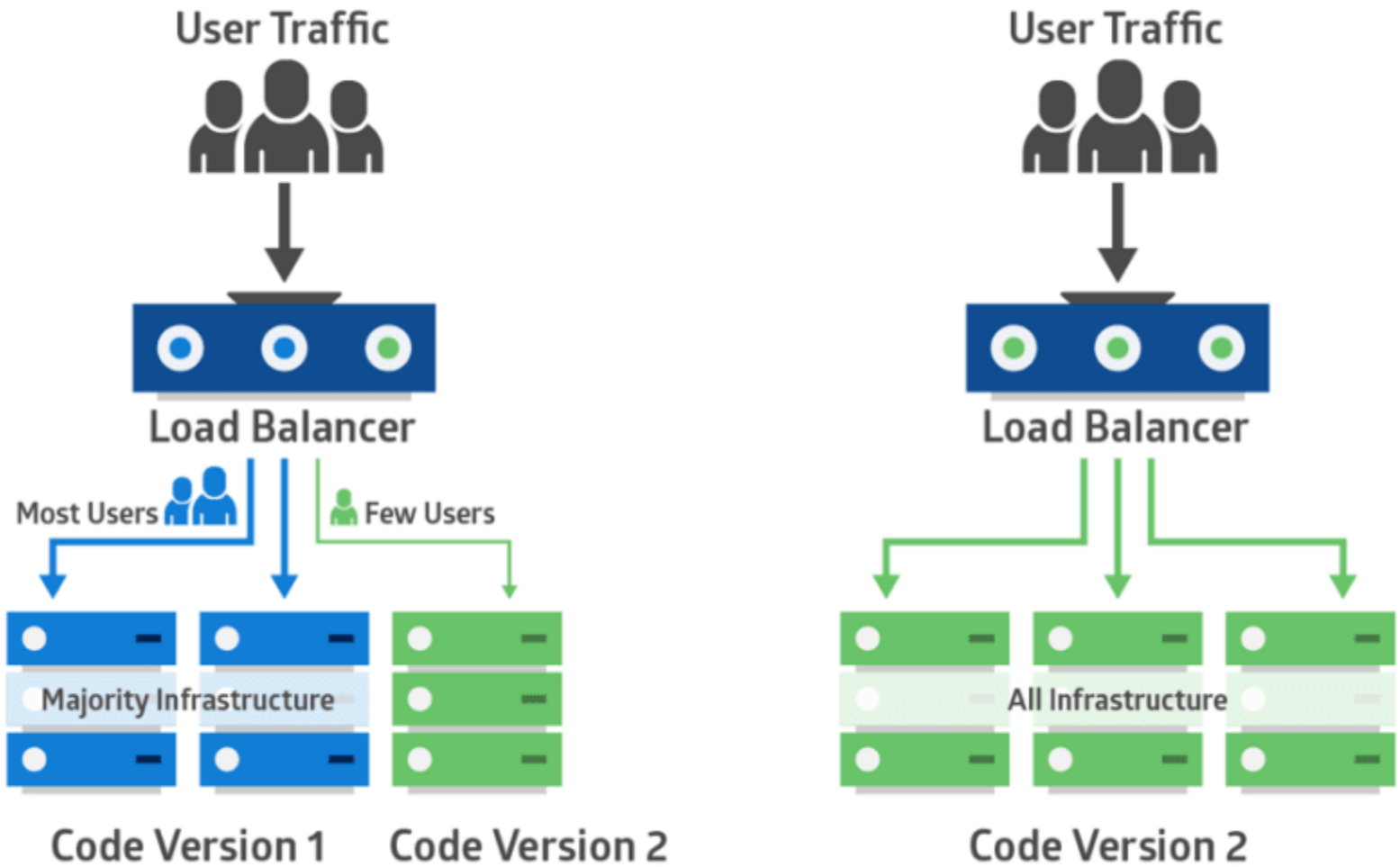
Canary Deployment

Canary deployment is like blue-green.

Instead of switching from blue to green in one step, we use a phased approach.

With canary deployment, we deploy a new application code in a small part of the production infrastructure. Once the application is signed off for release, only a few users are routed to it. This minimizes any impact.

With no errors reported, the new version can gradually roll out to the rest of the infrastructure. The image below demonstrates canary deployment:


User Traffic
Load Balancer
Most Users
Few Users
Majority Infrastructure
Code Version 1
Code Version 2
User Traffic
Load Balancer
All Infrastructure
Code Version 2